

CSHP01D.book	2
CSHP02D.book	70
CSHP03D.book	166
CSHP04D.book	268
CSHP05D.book	348
CSHP06D.book	430
CSHP07D.book	518
CSHP08D.book	590
CSHP09D.book	656
CSHP10D.book	732
CSHP11D.book	808
CSHP12D.book	886
CSHP13D.book	960
CSHP14D.book	1040
CSHP15D.book	1106
CSHP16D.book	1188
CSHP17D.book	1254
CSHP18D.book	1312
CSHP19D.book	1386
CSHP20D.book	1460
CSHP21D.book	1546
CSHP22D.book	1620
CSHP23D.book	1718
CSHP24D.book	1788

Objektorientierte Software-Entwicklung mit C#

Installation und erste Schritte mit
Visual Studio Community 2019

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0919N01

CSHP01D

Objektorientierte Software-Entwicklung mit C#

Installation und erste Schritte mit Visual Studio Community 2019

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Installation und erste Schritte mit Visual Studio Community 2019

Inhaltsverzeichnis

Einleitung	1
1 C# und das .NET Framework	3
1.1 Aufgaben einer höheren Programmiersprache	3
1.2 Geschichte von C#	4
1.3 CLI und .NET	5
Zusammenfassung	6
2 Installation von Visual Studio Community 2019	7
2.1 Installation durchführen	7
2.2 Der Desktop von Visual Studio Community 2019	10
2.3 Den Desktop anpassen	14
2.4 Bei Visual Studio anmelden	17
Zusammenfassung	19
3 „Hallo Welt“-Programm erstellen	21
3.1 Neues Projekt erstellen	21
3.2 Quelltext erfassen	25
3.3 Projekt speichern und öffnen	26
3.4 Kompilieren und Ausführen	28
3.5 Fehlersuche und -behebung	30
Zusammenfassung	37
4 Die Hilfefunktionen von Visual Studio Community 2019	39
4.1 Suchen in der Hilfe	42
4.2 Weitere nützliche Hilfefunktionen	43
Zusammenfassung	48

Schlussbetrachtung	49
---------------------------------	----

Anhang

A. Lösungen der Aufgaben zur Selbstüberprüfung	50
B. Glossar	51
C. Literaturverzeichnis	54
D. Abbildungsverzeichnis	55
E. Codeverzeichnis	57
F. Medienverzeichnis	58
G. Sachwortverzeichnis	59
H. Einsendeaufgabe	61

Einleitung

Herzlich willkommen!

Dieser Lehrgang beschäftigt sich mit der Software-Entwicklung mit C#. Sie werden lernen, wie Sie Software-Systeme mit Methoden, Techniken und Werkzeugen des **Software-Engineering¹** erstellen. Als Programmiersprache setzen wir dabei C# und das .NET Framework ein.

Neben einigen kleineren Projekten, die Sie mit der Programmierung vertraut machen, werden wir in diesem Lehrgang auch ein größeres Projekt mit allen wichtigen Entwicklungsschritten umsetzen. Dabei verwenden wir unter anderem die Modellierungssprache **UML** (*Unified Modeling Language²*).

In diesem Studienheft beschäftigen wir uns zunächst mit den Aufgaben einer höheren Programmiersprache wie C#. Außerdem erfahren Sie etwas zur Geschichte von C#.

Anschließend lernen Sie, wie Sie Visual Studio Community 2019 auf Ihrem Rechner installieren und wie Sie das Programm bedienen. Dabei zeigen wir Ihnen, wie Sie Quelltexte eingeben, speichern, kompilieren und ausführen. Außerdem stellen wir Ihnen die Hilfe von Visual Studio vor.

Wenn Sie dieses Studienheft bearbeitet haben, werden Sie Ihr erstes eigenes C#-Programm erstellt und ausgeführt haben.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie die Programmiersprache C# entstanden ist,
- was sich hinter Begriffen wie CLI und .NET Framework verbirgt,
- wie Sie Visual Studio Community 2019 installieren,
- wie Sie die Arbeitsoberfläche von Visual Studio an Ihre Bedürfnisse anpassen,
- wie Sie ein neues Projekt erstellen,
- wie Sie einen Quelltext eingeben,
- wie Sie ein Programm kompilieren und ausführen,
- wie Sie Fehler in einem Programm erkennen und korrigieren und
- wie Sie mit der Hilfe von Visual Studio arbeiten.

Damit Sie die Bedienung von Visual Studio Community 2019 leichter nachvollziehen können, finden Sie auf der Online-Lernplattform einige Videos zum Thema.

Außerdem finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform die Quelltexte für die Studienhefte. Die Namen der Projekte entsprechen dabei der Nummerierung der Codes in den Studienheften. Den Code 2.1 aus dem Studienheft CSHP02D könnten Sie zum Beispiel laden, indem Sie das Projekt **Cshp02d_02_01** in der Download-Datei für das Studienheft CSHP02D öffnen.

Christoph Siebeck

1. Engineering bedeutet übersetzt so viel wie „Ingenieurswesen“.

2. Übersetzt bedeutet *Unified Modeling Language* so viel wie „Vereinheitlichte Modellierungssprache“.

1 C# und das .NET Framework

In diesem Kapitel beschäftigen wir uns mit den Aufgaben einer höheren Programmiersprache und der Geschichte von C#. Außerdem stellen wir Ihnen das .NET Framework vor.

1.1 Aufgaben einer höheren Programmiersprache

Wie Sie vielleicht bereits wissen, arbeitet ein Computer intern nur mit Einsen und Nullen. Über diese beiden Ziffern werden die beiden Zustände **Strom fließt** beziehungsweise **Strom fließt nicht** dargestellt. Dieses binäre System mit zwei eindeutigen Zuständen ist die Grundlage sämtlicher Funktionen, die ein herkömmlicher Computer ausführt. Auch ein Programm – also eine Reihe von Anweisungen, die der Computer ausführt – besteht auf der untersten – der ausführbaren – Ebene aus einer langen Kombination von Einsen und Nullen.

In den Anfangszeiten wurden Computer direkt über diese Kombinationen von Einsen und Nullen programmiert. Da kein unmittelbar nachvollziehbarer Zusammenhang zwischen den gewünschten Aktionen und der Kombinationen aus Einsen und Nullen bestand, war diese direkte Art der Programmierung extrem zeitaufwendig und auch sehr fehleranfällig.

Um dieses Problem zu umgehen, wurden **Assembler** eingesetzt. Bei einem Assembler erfolgt die Programmierung über **Mnemonics**, kurze mehr oder weniger sprechende Befehle, die leichter zu merken sind als eine Kombination aus Einsen und Nullen. Ein typischer Assembler-Befehl ist zum Beispiel `mov` für *move* (verschieben). Mit diesem Befehl werden unter anderem Werte gesetzt.

Assembler-Programme lassen sich nicht direkt ausführen, da die Mnemonics nicht vom Computer verarbeitet werden können. Vor der Abarbeitung eines Assembler-Programms müssen die Mnemonics erst wieder in Kombinationen aus Einsen und Nullen umgewandelt werden. Diese Aufgabe übernimmt ebenfalls der Assembler.

Da auch die Programmierung in Assembler recht kompliziert war, wurden **Hochsprachen** wie Basic, Pascal, C++ und C# entwickelt. Die Befehle dieser Hochsprachen orientieren sich zum Teil sehr eng an der menschlichen Sprache und lassen sich daher sehr viel leichter merken als die Mnemonics eines Assemblers. In der Programmiersprache COBOL können Sie zum Beispiel Befehle wie `MOVE A TO B` benutzen (übersetzt etwa: Bewege A nach B).

Außerdem unterstützen viele Hochsprachen Strukturen wie zum Beispiel Schleifen und Methoden oder Objekte, die die Programmierung erheblich vereinfachen. Was sich genau hinter diesen Begriffen verbirgt, erfahren Sie im weiteren Verlauf des Lehrgangs.

Die Anweisungen einer Hochsprache müssen vor der Ausführung ebenfalls in ein Format übersetzt werden, das der Computer direkt verarbeiten kann. Dazu werden die Anweisungen des Programms – der **Quelltext** – in der Regel durch einen **Compiler**³ und einen **Linker**⁴ umgesetzt. Erst das so entstandene Programm kann dann ausgeführt werden.

3. *To compile* bedeutet übersetzt so viel wie „zusammentragen“.

4. *To link* bedeutet übersetzt so viel wie „verbinden“.



Ein Compiler ist für die eigentliche Übersetzung des Quelltextes zuständig. Ein Linker fügt mehrere Dateien zu einem ausführbaren Programm zusammen.

Neben den Hochsprachen, die mit einem Compiler arbeiten, gibt es auch einige Hochsprachen, die das Programm durch einen **Interpreter**⁵ verarbeiten. Hier wird der ausführbare Code nicht vor der Ausführung des Programms erzeugt, sondern zur Laufzeit des Programms. Das heißt, die einzelnen Anweisungen werden erst unmittelbar vor der Ausführung einzeln übersetzt. Beispiele für Programmiersprachen, die mit einem Interpreter arbeiten, sind ältere BASIC-Versionen und JavaScript für Programmieraufgaben im Internet-Bereich.

1.2 Geschichte von C#

Nach diesem kurzen Überblick über die Entstehung und die Aufgaben von Hochsprachen wollen wir uns mit der Geschichte von C# beschäftigen.



Der Name C# (C sharp) beziehungsweise die Schreibweise leitet sich aus der Musik ab. Dort steht C# für den Notenwert Cis, der einen halben Ton höher ist als der Notenwert C.

Wie viele andere höhere Programmiersprachen auch, ist C# aus mehreren anderen Programmiersprachen entstanden. Diese Entwicklung lässt sich damit erklären, dass im Laufe der Zeit häufig die typischen Vorteile bestimmter Programmiersprachen in anderen, neueren Programmiersprachen zusammengefasst wurden.

Einen Stammbaum der Programmiersprache C# mit den wichtigsten Entwicklungen sehen Sie in der folgenden Abbildung.

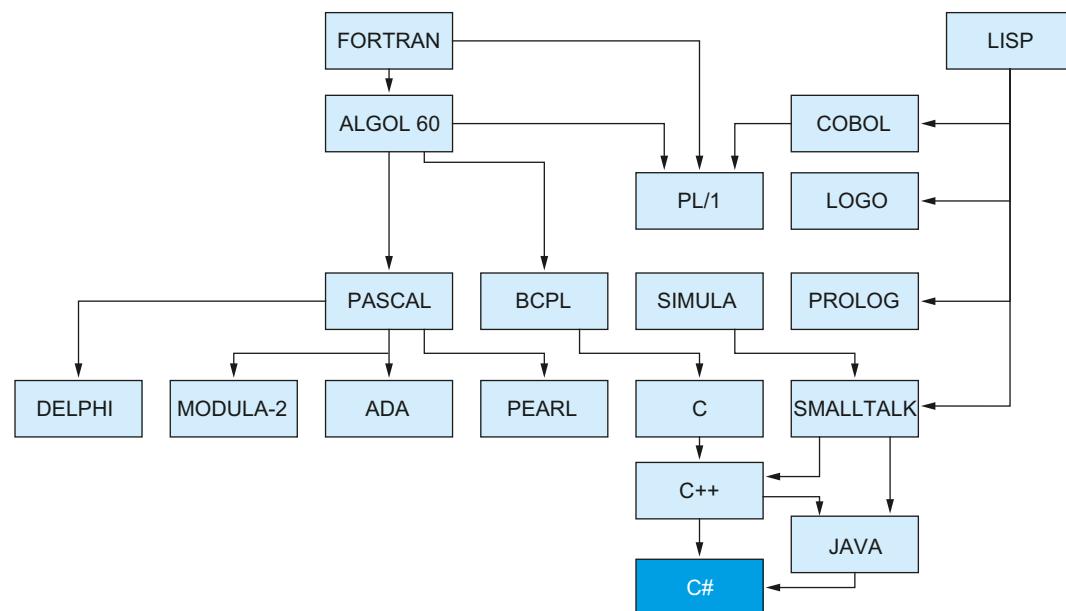


Abb. 1.1: Stammbaum von C#

5. *To interpret* bedeutet übersetzt so viel wie „auslegen“ oder „dolmetschen“.

1.3 CLI und .NET

Wenn Sie sich den Stammbaum von C# in der vorherigen Abbildung etwas genauer ansehen, ahnen Sie vielleicht schon ein typisches Problem beim Erstellen von Computerprogrammen: Es gibt unzählige Programmiersprachen, die nicht zueinanderpassen. So können Sie in der Regel Teile eines Programms, das mit C# erstellt wurde, nicht mit Teilen eines Programms zusammenbauen, das zum Beispiel in Java programmiert wurde. Ihnen bleibt dann nichts weiter übrig, als die Programmteile, die Sie benötigen, noch einmal neu zu programmieren.

Einen Ausweg aus diesem Dilemma bietet die *Common Language Infrastructure*⁶ – kurz CLI. Sie schafft eine gemeinsame Basis für das Erstellen von Computerprogrammen – unabhängig von einer bestimmten Programmiersprache und auch von einer bestimmten Plattform. Das heißt: Sie können theoretisch mit einer x-beliebigen Programmiersprache ein Programm unter dem Betriebssystem Windows erstellen und dieses Programm zum Beispiel mit einer anderen Programmiersprache unter dem Betriebssystem Linux weiterentwickeln. Ausführen lässt sich das Programm dann theoretisch unter jedem beliebigen Betriebssystem.

Das Ganze funktioniert allerdings nur, wenn bestimmte Rahmenbedingungen geschaffen werden – zum Beispiel durch das .NET Framework⁷. Es sorgt dafür, dass Programme zunächst in eine Zwischensprache – die *Common Intermediate Language*⁸ (CIL) – umgesetzt werden. Diese Zwischensprache wird dann auf jedem einzelnen Computer durch einen *Just-in-time-Compiler*⁹ (JIT) in ein ausführbares Programm umgewandelt. Der *Just-in-time-Compiler* selbst ist Bestandteil einer Laufzeitumgebung – der *Common Language Runtime* (CLR). Diese Laufzeitumgebung muss auf jedem Rechner vorhanden sein, der ein Programm, das für das .NET Framework erstellt wurde, ausführen soll.

C# ist fest mit dem .NET Framework verbunden und vollständig in das Framework integriert.



Neben dieser ganzen Technik bietet das .NET Framework aber noch einen weiteren Aspekt, der für Sie als Programmierer im Moment vielleicht der interessanteste ist: Das Framework stellt Ihnen zahlreiche vorgefertigte Komponenten und Funktionen zur Verfügung, die Sie sehr einfach in Ihren eigenen Programmen wiederverwenden können. So können Sie zum Beispiel auch anspruchsvollere Windows-Anwendungen komfortabel mit einigen wenigen Mausklicks erstellen. Wie das genau geht, erfahren Sie detailliert im weiteren Verlauf.

Das Zusammenstellen von Anwendungen aus vorgefertigten Komponenten wird auch *Rapid Application Development (RAD)* genannt. Übersetzt bedeutet *Rapid Application Development* so viel wie „schnelle Anwendungsentwicklung“.



6. Übersetzt bedeutet *common language infrastructure* so viel wie „allgemeine Sprachinfrastruktur“ oder „gemeinsame Sprachinfrastruktur“.
7. Der „Punkt“ wird ausgesprochen *dot.net* bedeutet übersetzt „Netz“. Framework bedeutet übersetzt so viel wie „Rahmen“.
8. *Intermediate Language* bedeutet übersetzt so viel wie „Zwischensprache“.
9. *Just in time* bedeutet übersetzt so viel wie „genau rechtzeitig“.

Wenn Ihnen jetzt der Kopf brummt vor lauter Fachbegriffen und Abkürzungen, ein Trost: Wie die Umsetzung genau erfolgt und welche Teile dabei was erledigen, müssen Sie nicht im Detail wissen. Interessant ist jetzt vor allem eins: Sie haben mit Visual Studio ein Programmierwerkzeug in Händen, das das .NET Framework in der Version 4.7 unterstützt. Damit sind Sie in der Lage, in vergleichsweise kurzer Zeit auch anspruchsvolle Windows-Programme zu erstellen.

Zusammenfassung

Ein ausführbares Programm besteht aus einer langen Kombination von Nullen und Einsen.

Der Quelltext eines Programms muss vor der Ausführung in ein Format übersetzt werden, das der Computer verarbeiten kann. Die Übersetzung kann zum Beispiel durch einen Compiler und einen Linker erfolgen.

Die Common Language Infrastructure (CLI) schafft eine gemeinsame Basis für das Erstellen von Computerprogrammen – unabhängig von einer Programmiersprache und der Plattform. Dazu werden bestimmte Rahmenbedingungen benötigt, die zum Beispiel durch das .NET Framework geschaffen werden.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Wie bezeichnet man die Programmiersprachen, zu denen zum Beispiel C, C# und Pascal gehören?

- 1.2 Welche beiden Programmiersprachen waren wichtige Vorläufer der Programmiersprache C#?

- 1.3 Beschreiben Sie kurz die zwei wesentlichen Vorteile des .NET Frameworks.

2 Installation von Visual Studio Community 2019

In diesem Kapitel beschäftigen wir uns zuerst mit der Installation von Microsoft Visual Studio Community 2019. Danach stellen wir Ihnen den Desktop von Visual Studio vor.

Bei Visual Studio handelt es sich um eine integrierte Entwicklungsumgebung, die verschiedene Programme zusammenfasst, die Sie für die Entwicklung benötigen. Dazu gehören zum Beispiel:

- ein Texteditor für das Erfassen der Quelltexte,
- ein Compiler zum Übersetzen der Quelltexte und
- verschiedene weitere Hilfsprogramme.

Außerdem umfasst Visual Studio Community 2019 noch das .NET Framework.

2.1 Installation durchführen

Sehen wir uns die Installation von Visual Studio an. Wir zeigen Ihnen dabei die Installation unter dem Betriebssystem Windows 10. Bei anderen Windows-Versionen läuft die Installation nahezu identisch ab. Lediglich das Aussehen der Oberfläche und der Fenster kann sich von den Abbildungen in diesem Studienheft unterscheiden.

Bitte überprüfen Sie zunächst, ob Ihr Rechner folgende Mindestanforderungen erfüllt:

- Windows 7 mit Service Pack 1, Windows 8.1 oder Windows 10,
- Prozessor mit 1,8 Gigahertz,
- 2 Gigabyte Arbeitsspeicher und
- circa 50 Gigabyte freier Platz auf der Festplatte.

Außerdem benötigen Sie eine Internet-Verbindung, da die Installation online erfolgt.

Hinweise:

Es handelt sich um Empfehlungen. Sie können Visual Studio Community 2019 auch mit weniger Arbeitsspeicher oder einem langsameren Prozessor einsetzen. Sie benötigen für unsere Beispielinstallation allerdings mindestens 25 Gigabyte freien Platz auf der Festplatte.

Bitte beachten Sie, dass Sie nicht mit allen Betriebssystemen alle Anwendungstypen erstellen können. Das betrifft vor allem mobile Anwendungen wie Universal Windows Apps. Die größte Auswahl haben Sie, wenn Sie Windows 10 als Betriebssystem verwenden.

Für die Installation benötigen Sie administrative Rechte.

An die Grafikkarte und die Maus werden keine besonderen Anforderungen gestellt.

Rufen Sie in Ihrem Browser die Seite <https://visualstudio.microsoft.com/de/downloads/> auf. Laden Sie dort die Installationsdatei für Visual Studio Community 2019 herunter und lassen Sie sie ausführen.



Bitte achten Sie unbedingt darauf, dass Sie Installationsdateien für Visual Studio Community 2019 herunterladen. Die Versionen Professional und Enterprise-Versionen von Visual Studio können Sie nicht unbegrenzt kostenlos nutzen.

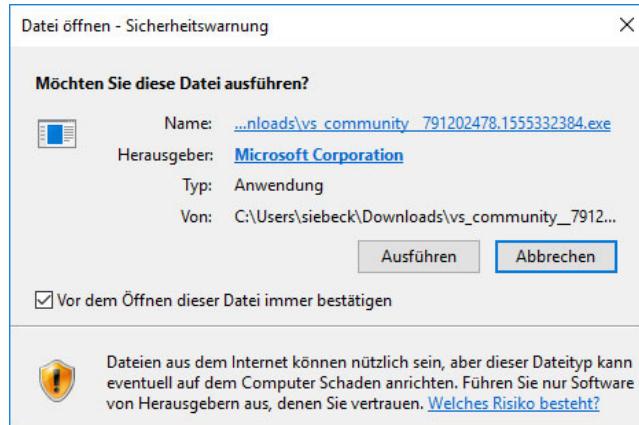


Abb. 2.1: Die Sicherheitswarnung beim Ausführen

Lassen Sie danach das Ausführen ausdrücklich zu, und bestätigen Sie auch die Abfrage der Benutzerkontensteuerung mit Ja.

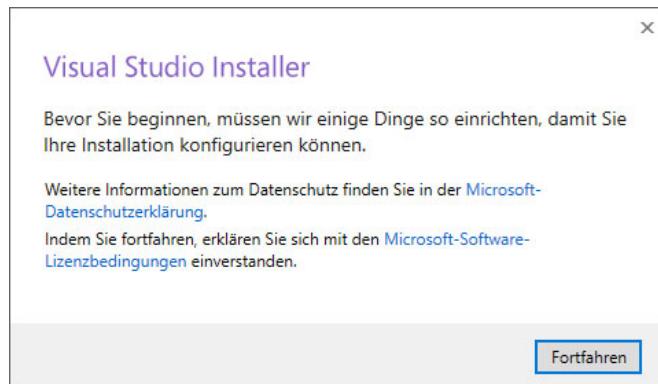


Abb. 2.2: Der erste Schritt des Installers

Im ersten Schritt des Installers erhalten Sie lediglich einen Hinweis, dass einige Vorbereitungen getroffen werden müssen. Außerdem können Sie hier die Datenschutzerklärung und die Lizenzbedingungen nachlesen.

Klicken Sie für den nächsten Schritt auf die Schaltfläche Fortfahren.

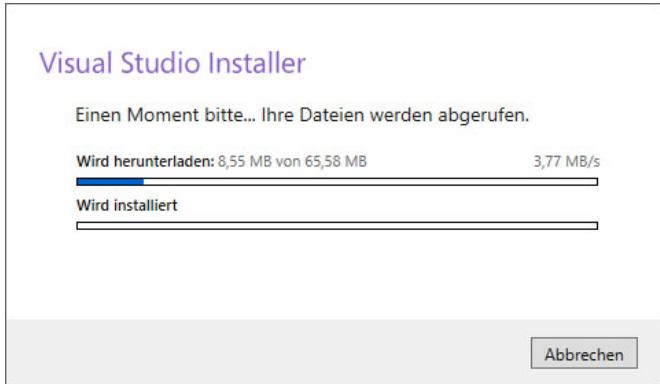


Abb. 2.3: Installation der ersten Dateien

Danach werden einige Dateien aus dem Internet heruntergeladen und installiert.

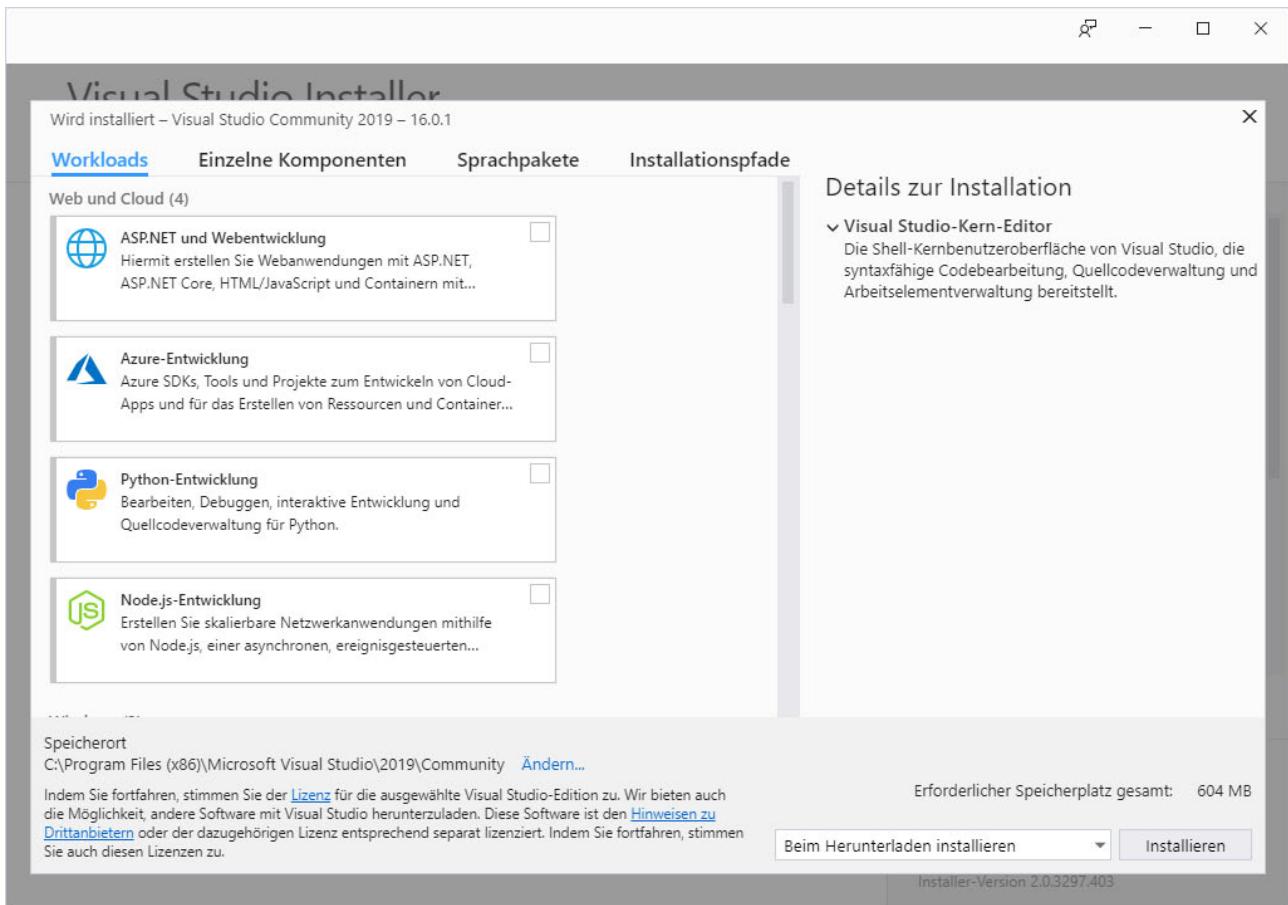


Abb. 2.4: Die Auswahl der Workloads

Anschließend können Sie die Workloads auswählen, die Sie installieren möchten. Wir benötigen die Workloads **.NET-Desktopentwicklung** sowie **Entwicklung für die universelle Windows-Plattform**. Sie finden beide im Bereich **Windows** in der Mitte des Fensters.

Markieren Sie bitte die beiden Workloads und klicken Sie auf **Installieren** unten rechts im Fenster. Danach beginnt die eigentliche Installation. Sie kann durchaus einige Zeit dauern.

Anschließend können Sie die Entwicklungsumgebung durch einen Mausklick auf die Schaltfläche **Starten** zum ersten Mal starten. Das Installationsprogramm erstellt aber auch Einträge im Startmenü.

Hinweis:

Das Installationsprogramm startet Visual Studio in den Standardeinstellungen automatisch nach der Installation. Allerdings liegt das Fenster unter Umständen im Hintergrund und ist nicht zu sehen.

Rufen Sie Visual Studio jetzt bitte auf, falls es nicht automatisch gestartet wurde. Unter Windows 10 klicken Sie dazu im Startmenü auf den Eintrag **Visual Studio 2019**.

2.2 Der Desktop von Visual Studio Community 2019



Abb. 2.5: Die Anmeldung

Nach dem ersten Start können Sie sich bei Visual Studio anmelden. Da wir uns jetzt aber erst einmal den Desktop ansehen wollen, holen wir das später nach. Klicken Sie daher auf den Link **Jetzt nicht, vielleicht später** unterhalb der Schaltfläche **Anmelden**.

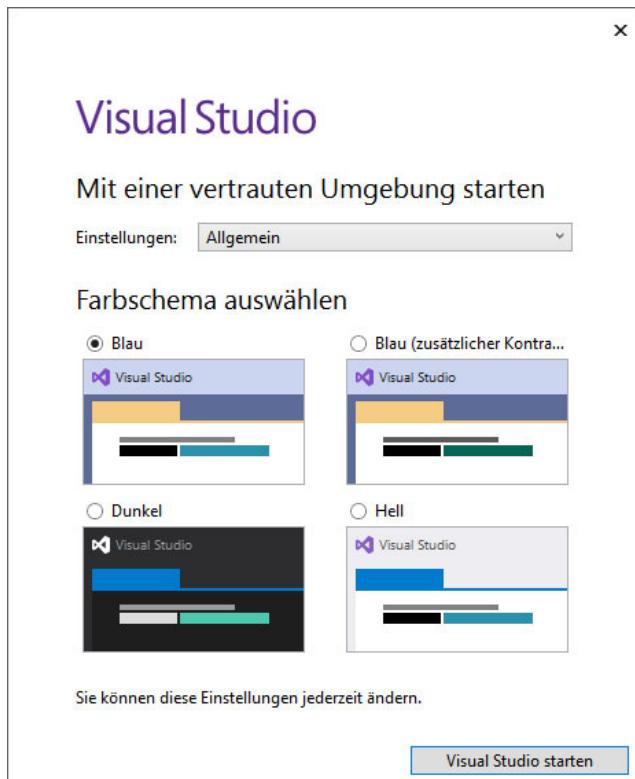


Abb. 2.6: Die Auswahl des Farbschemas

Danach können Sie ein Farbschema auswählen. Wir übernehmen hier den Vorschlag **Blau**. Klicken Sie bitte auf die Schaltfläche **Visual Studio starten** unten rechts im Fenster.

Anschließend richtet Visual Studio zunächst einmal die Arbeitsumgebung ein. Danach erscheint dann die Startseite. Hier können Sie vorhandene Projekte öffnen oder neue erstellen.

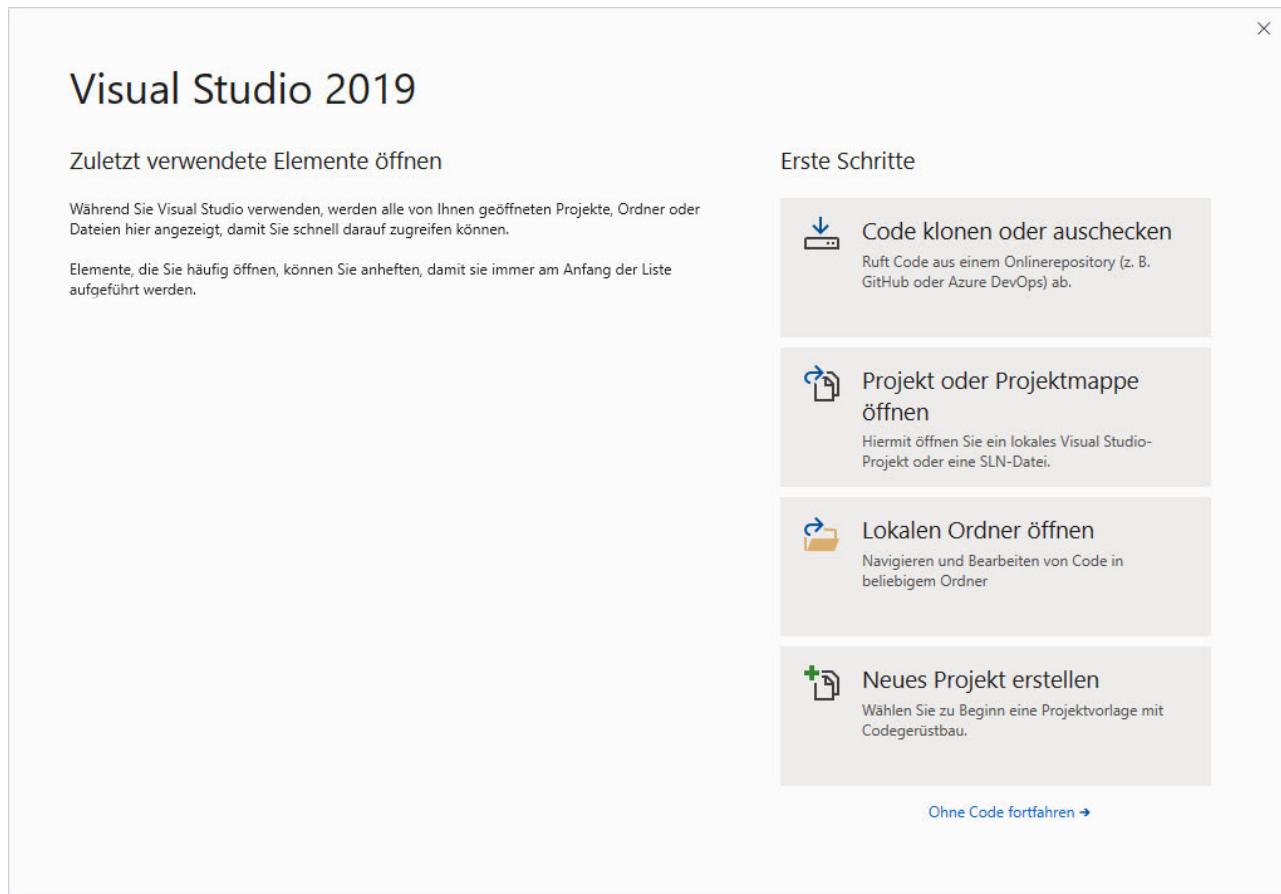


Abb. 2.7: Die Startseite

Wir wollen uns aber zunächst einfach einmal den Desktop ansehen. Klicken Sie daher auf den Link **Ohne Code fortfahren** unten rechts.

Der Desktop untergliedert sich in folgende Elemente:

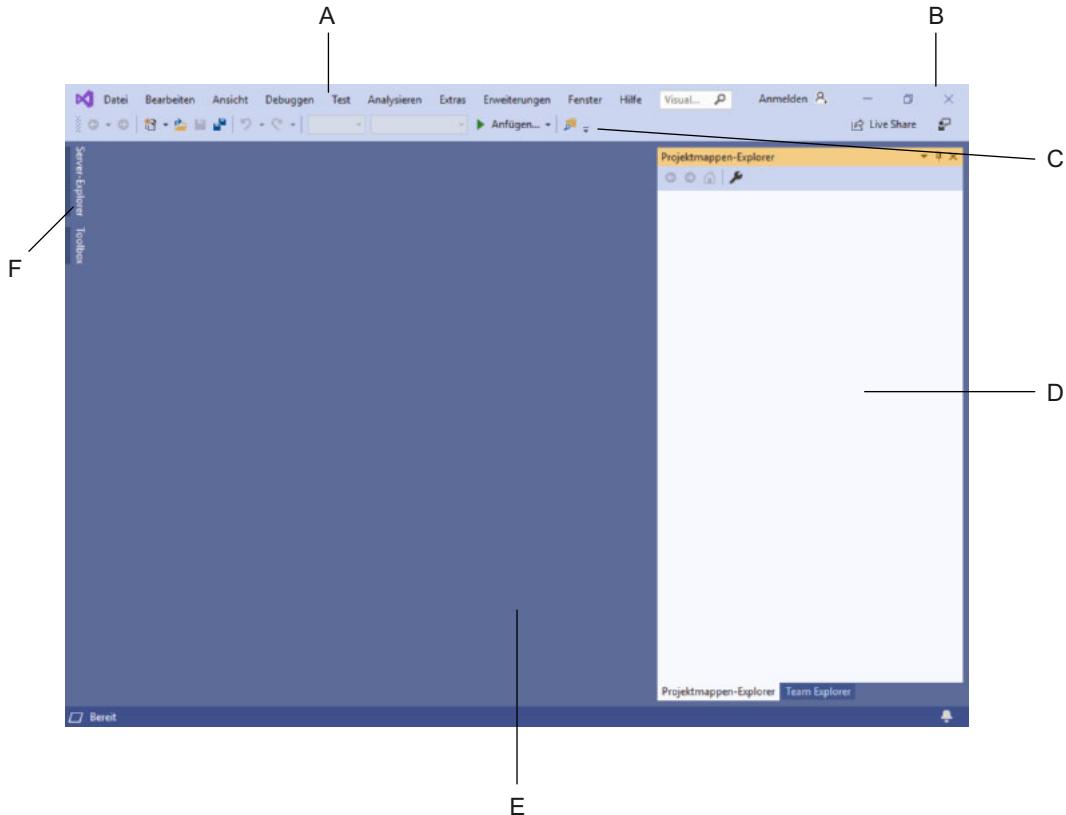


Abb. 2.8: Der Desktop von Visual Studio Community 2019

- A die Menüleiste
die Standard-Windows-Schaltflächen zum Minimieren und Maximieren des Fensters sowie zum Beenden der Anwendung
- B die Symbolleisten
Welche Symbole hier angezeigt werden, hängt unter anderem von Ihrer aktuellen Bildschirmauflösung ab.
- C den Projektmappen-Explorer
Er zeigt Ihnen alle wichtigen Dateien in einem Projekt in einer hierarchischen Struktur. In unserem Beispiel ist er noch leer, da wir kein Projekt geöffnet haben.
- D den eigentlichen Arbeitsbereich
Hier erfassen Sie Quelltexte und entwerfen Formulare. Er ist in unserem Beispiel ebenfalls noch leer, da wir kein Projekt geöffnet haben.
- E die Registerzunge zum Einblenden der Toolbox und des Server-Explorers
Was sich dahinter genau verbirgt, erfahren Sie im weiteren Verlauf der Studienhefte.

2.3 Den Desktop anpassen

Die einzelnen Bereiche beziehungsweise Fenster können Sie gezielt aus- und wieder einblenden. Zum Ausblenden klicken Sie auf das Symbol **Schließen**  oben rechts im Bereich. Probieren Sie das einmal aus. Schließen Sie bitte den Projektexplorer. Der Desktop sollte nun ungefähr so aussehen wie in der folgenden Abbildung.

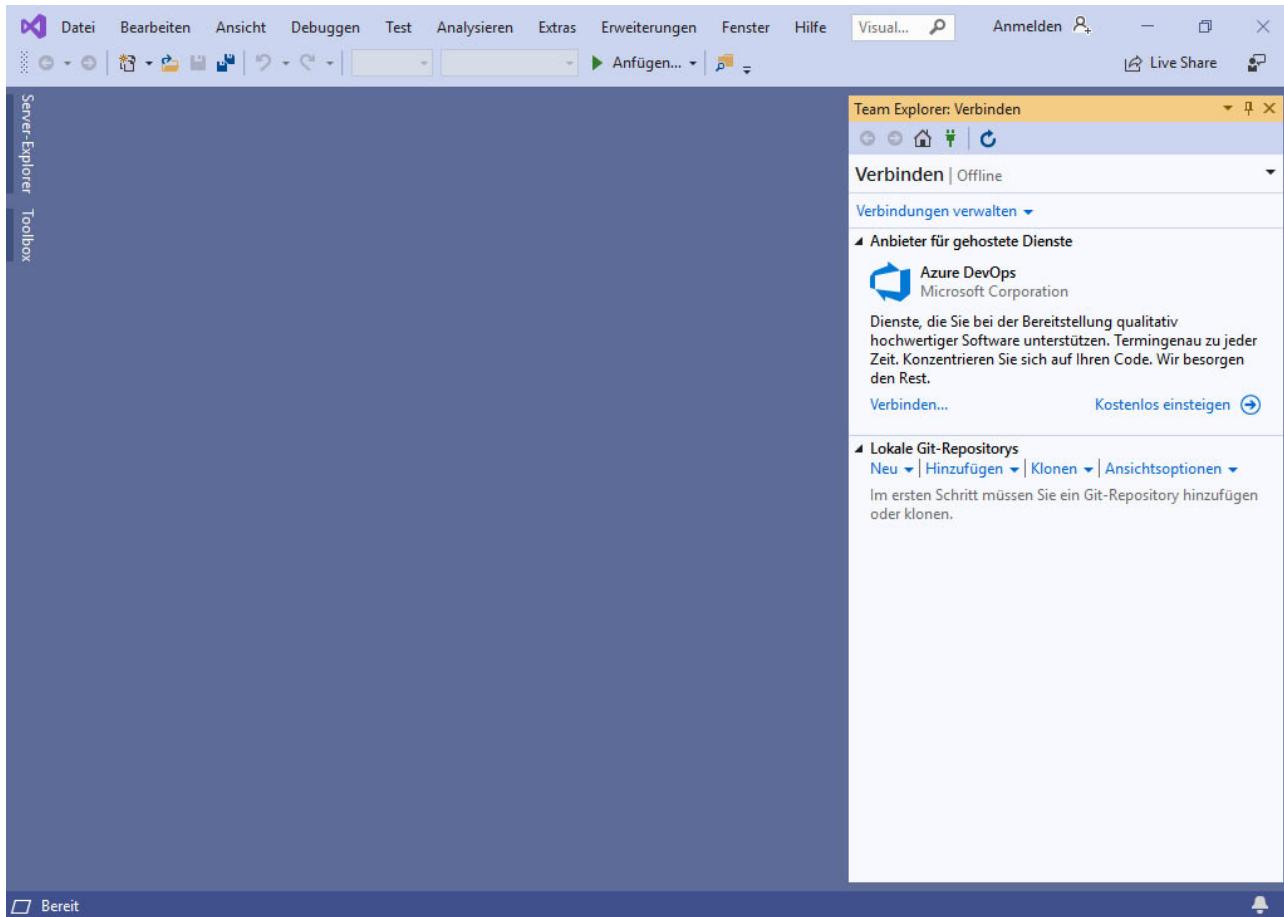


Abb. 2.9: Der Desktop ohne den Projektexplorer

Über das Menü **Ansicht** können Sie einen ausgeblendeten Bereich wieder einblenden. Dazu öffnen Sie das Menü und klicken dann auf den Eintrag des gewünschten Bereichs. Blenden Sie den Projektexplorer jetzt wieder ein.

Wenn Sie einen Bereich nur zeitweise ausblenden wollen, können Sie ihn mit dem Symbol **Automatisch im Hintergrund**  oben rechts in dem Bereich auch in den Hintergrund setzen. Der Bereich wird dann verkleinert am Rand dargestellt. Um ihn wieder vollständig anzuzeigen, klicken Sie mit der Maus auf die Registerzunge. Wenn Sie danach an eine beliebige Stelle außerhalb des Bereichs klicken, wird er automatisch wieder verkleinert.

Probieren Sie auch das bitte aus. Setzen Sie den Projektmappen-Explorer bitte in den Hintergrund. Der Desktop sollte nun so aussehen:

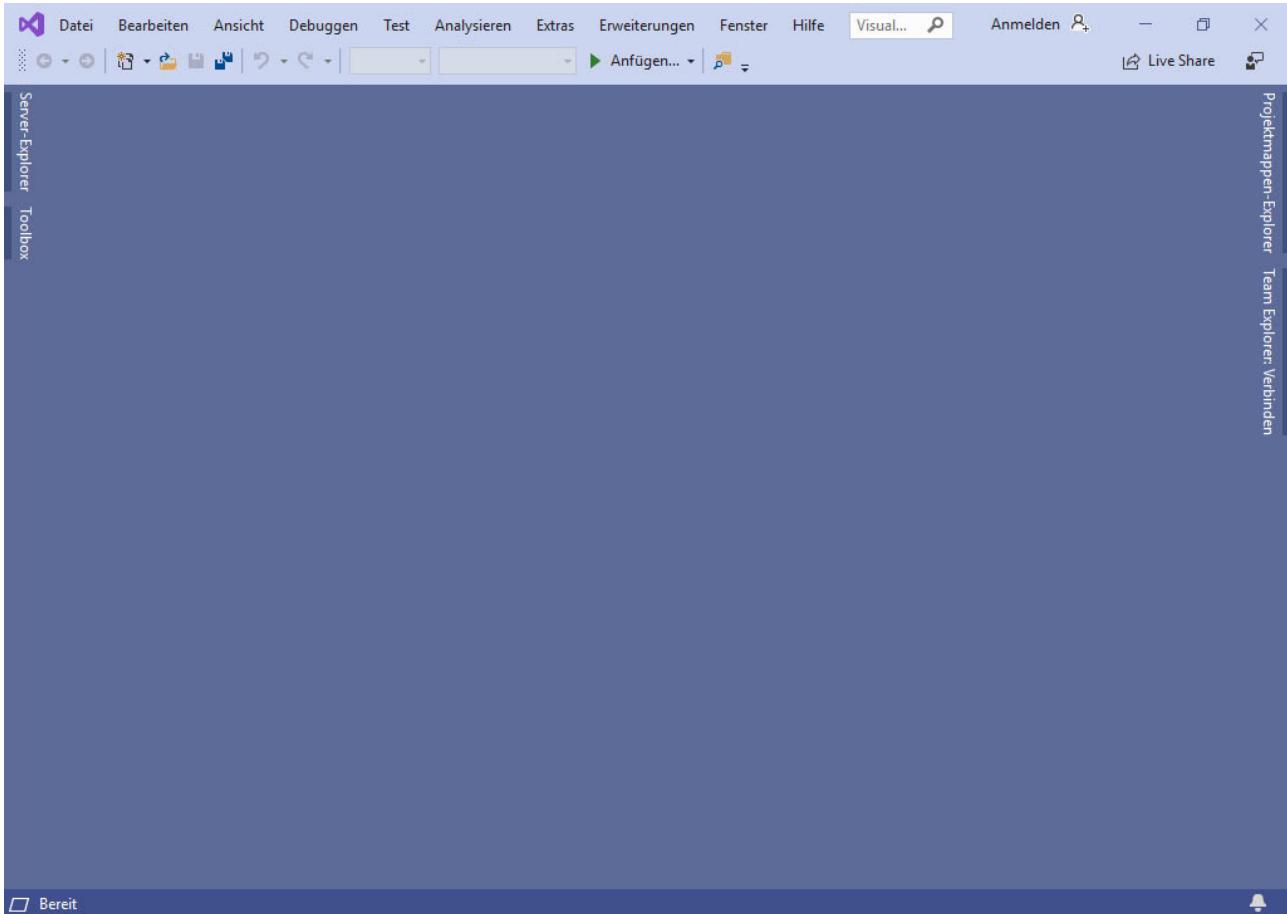


Abb. 2.10: Der Projektmappen-Explorer im Hintergrund

Wie Sie sehen, werden nun einige Registerzungen am rechten Rand des Bildschirms angezeigt. Über diese Registerzungen können Sie jetzt den Bereich wieder in den Vordergrund holen.

Um einen Bereich, den Sie in den Hintergrund gestellt haben, wieder ständig anzuzeigen, klicken Sie noch einmal auf das Symbol **Automatisch im Hintergrund** .

Sie können auch die Breite und die Höhe der einzelnen Bereiche nahezu beliebig verändern. Stellen Sie dazu den Mauszeiger auf den Rand des Bereichs und achten Sie darauf, dass er als Doppelpfeil dargestellt wird. Ziehen Sie dann mit gedrückter linker Maustaste am Rand des Bereichs, bis er die gewünschte Größe erreicht hat.

Auch die Position der verschiedenen Bereiche lässt sich verändern. Dazu stellen Sie den Mauszeiger auf die Titelleiste des Bereichs, den Sie verschieben wollen. Ziehen Sie dann den Bereich mit gedrückter linker Maustaste in die gewünschte Richtung. Nach kurzer Zeit erscheinen über dem Desktop zahlreiche Symbole, die Ihnen Hinweise auf die neue Position geben.

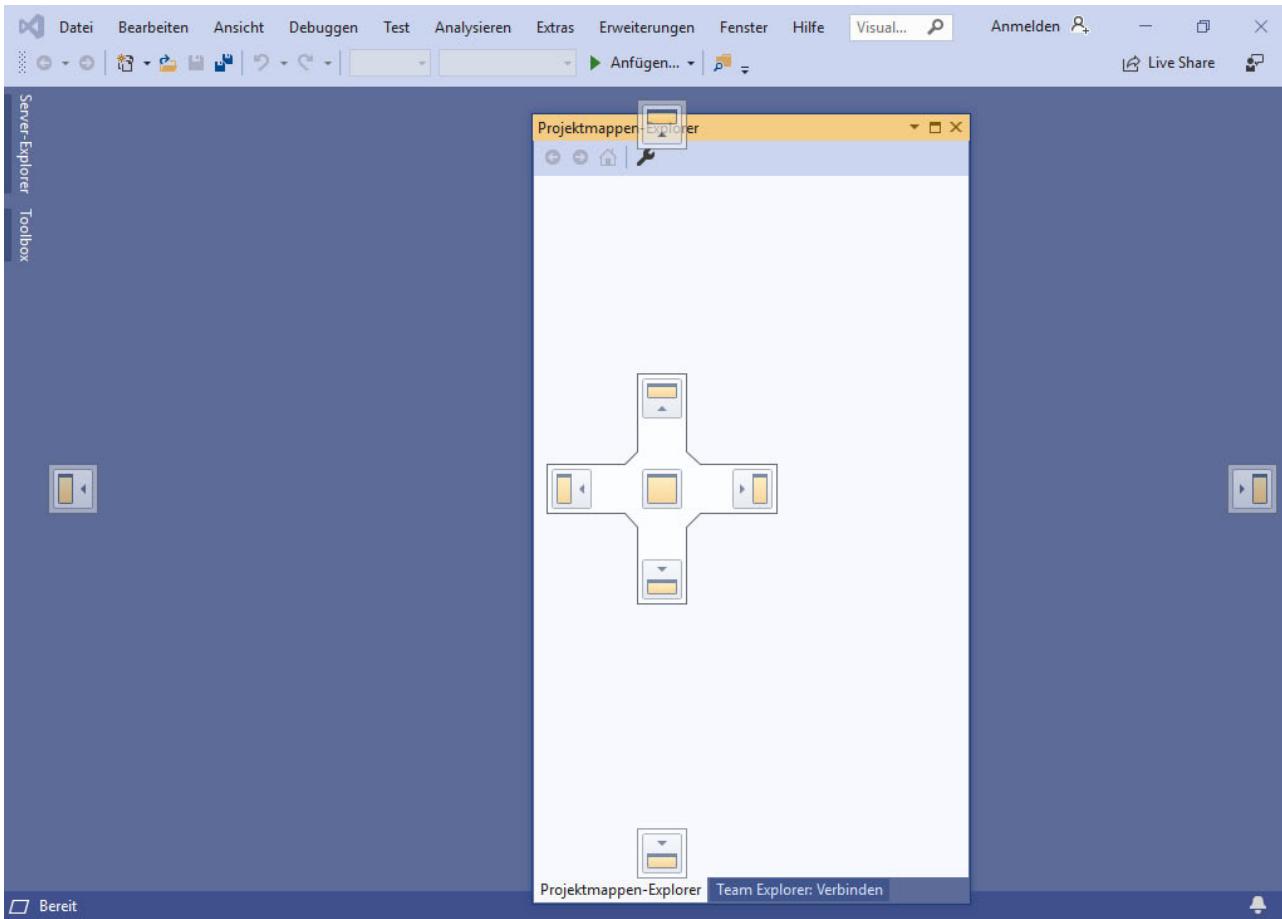


Abb. 2.11: Das Positionieren eines Bereichs auf dem Desktop

Wenn Sie den Mauszeiger auf eins dieser Symbole stellen und dann die linke Maustaste loslassen, wird der Bereich an der entsprechenden Position abgelegt. Wo genau der Bereich abgelegt wird, zeigt Ihnen Visual Studio auch durch eine blaue Markierung auf dem Bildschirm.

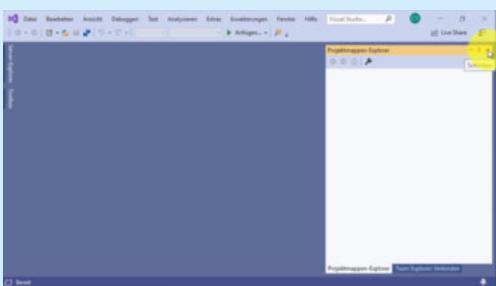
Sie können die Bereiche aber auch an beliebigen Positionen mitten auf dem Bildschirm ablegen. Dazu ziehen Sie den Bereich einfach mit gedrückter linker Maustaste an die gewünschte Stelle und lassen die Maustaste wieder los.

Probieren Sie die verschiedenen Möglichkeiten in Ruhe aus und richten Sie sich den Desktop so ein, wie Sie am besten damit arbeiten können.

In der Standardeinstellung speichert Visual Studio Ihre Desktop-Einstellungen übrigens automatisch. Das heißt, beim nächsten Start finden Sie den Desktop genau so wieder vor, wie Sie ihn verlassen haben.

Mit der Funktion **Extras/Einstellungen importieren/exportieren...** können Sie aber auch eine bestimmte Einstellung speichern und später wieder abrufen. Außerdem lassen sich hier auch die Standardeinstellungen wiederherstellen. Da sowohl das Speichern, das Abrufen als auch das Wiederherstellen durch einen Assistenten unterstützt werden, wollen wir Ihnen das Vorgehen hier nicht weiter vorstellen.

Für Ihren Lehrgang stellen wir Ihnen Lernvideos bereit, die die Inhalte noch einmal vertiefend darstellen. Wenn Sie das Heft als PDF- oder EPUB-Datei betrachten, können Sie das Video durch Klick auf das Vorschaubild oder den angegebenen Link aufrufen. Sollten Sie das Heft jedoch in gedruckter Form vor sich liegen haben, können Sie entweder den angegebenen Link in einem Internetbrowser eingeben oder aber mithilfe Ihres Smartphones und einer QR-App den abgedruckten QR-Code scannen, um das Video anzusehen.



In diesem Video stellen wir Ihnen den Desktop von Visual Studio und verschiedene Anpassungsmöglichkeiten vor.



www.dfz.media/uypeek

Video 2.1: Der Desktop von Visual Studio

So viel zum Desktop.

2.4 Bei Visual Studio anmelden

Sie können Visual Studio 2019 Community nach der Installation 30 Tage lang nutzen. Danach müssen Sie sich kostenlos bei Visual Studio anmelden. Damit Sie später keine Probleme bei der Nutzung bekommen, sollten Sie die Anmeldung am besten sofort nach der Installation durchführen. Sie benötigen dazu ein Microsoft-Konto und Zugang zum Internet.

Um die Anmeldung beziehungsweise Registrierung zu starten, klicken Sie im Menü Hilfe auf den Eintrag **Produkt registrieren**.



Abb. 2.12: Das Fenster zur Anmeldung

Im folgenden Fenster klicken Sie auf die Schaltfläche **Anmelden**.

Hinweis:

Wenn Sie noch kein Konto haben, können Sie es über den Link **Erstellen Sie ein Konto!** anlegen.

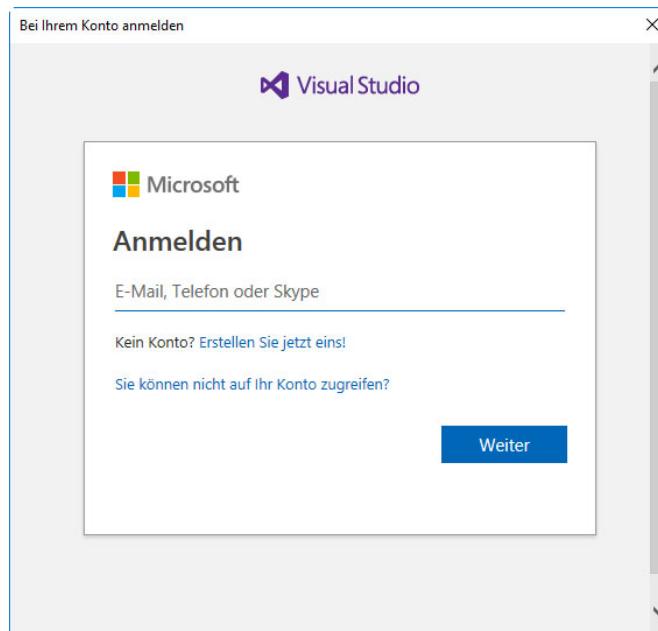


Abb. 2.13: Die Eingabe für das Konto

Danach geben Sie entweder die E-Mail-Adresse oder die Telefonnummer des Kontos ein, das Sie für die Anmeldung benutzen wollen. Klicken Sie danach auf **Weiter**.

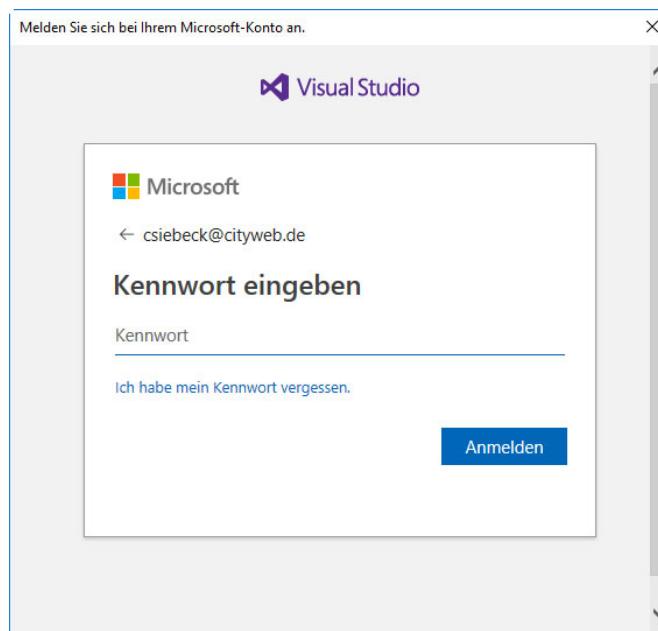


Abb. 2.14: Die Anmeldung über das Microsoft-Konto

Danach können Sie sich dann mit Ihrem Konto anmelden.

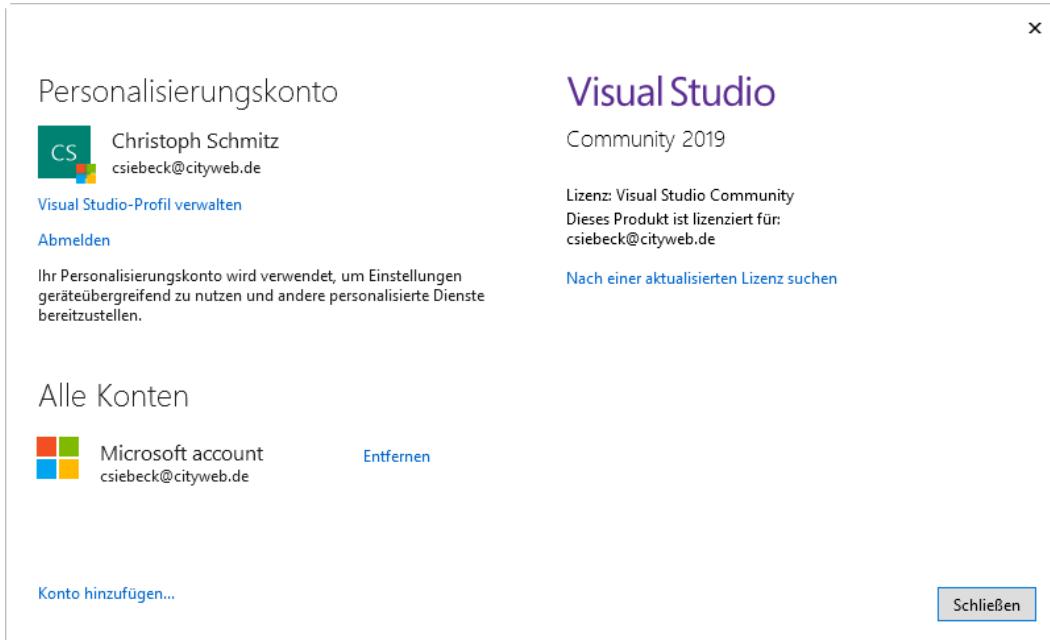


Abb. 2.15: Die aktualisierten Informationen

Wenn alles geklappt hat, wird das Konto kurze Zeit später angezeigt. Auch die Beschränkung auf 30 Tage sollte aufgehoben worden sein.

Im nächsten Kapitel werden Sie Ihr erstes Programm erstellen.

Zusammenfassung

Visual Studio fasst verschiedene Programme zusammen, die für die Entwicklung von Anwendungen benötigt werden.

Bei der Installation von Visual Studio werden Sie von einem Assistenten geführt und unterstützt.

Der Desktop von Visual Studio untergliedert sich in verschiedene Bereiche.

Das Aussehen des Desktops können Sie an Ihre eigenen Anforderungen anpassen.

Sie können Visual Studio 30 Tage nutzen. Danach ist eine Registrierung erforderlich.

Aufgaben zur Selbstüberprüfung

- 2.1 Nennen Sie zwei wichtige Programme, die zur integrierten Entwicklungsumgebung von Visual Studio gehören.

- 2.2 Sie möchten, dass ein Bereich auf dem Desktop von Visual Studio automatisch in den Hintergrund gestellt wird. Wie können Sie das erreichen?

3 „Hallo Welt“-Programm erstellen

Nachdem wir einen ersten Blick auf den Desktop von Visual Studio Community 2019 geworfen und die Registrierung durchgeführt haben, wollen wir jetzt das erste Programm erstellen.

Hinweis:

Da das Erstellen von Programmen mit grafischer Oberfläche einiges Vorwissen benötigt, erstellen wir in den ersten Studienheften ausschließlich Konsolenprogramme. Diese Konsolenprogramme laufen unter der Eingabeaufforderung von Windows.

Die Eingabeaufforderung starten Sie über **Start/Alle Programme/Zubehör/Eingabeaufforderung** beziehungsweise **Windows-System/Eingabeaufforderung** oder über die Kachel **Eingabeaufforderung**. Im folgenden Fenster können Sie dann über die Tastatur Befehle eingeben.

Die Konsolenprogramme, die wir in diesem Lehrgang erstellen, werden automatisch unter der Eingabeaufforderung gestartet. Sie müssen die Eingabeaufforderung also nicht selbst öffnen, um ein Beispielprogramm zu testen.

3.1 Neues Projekt erstellen

Das Erstellen von Programmen mit Visual Studio erfolgt über **Projekte**. Diese Projekte enthalten alle nötigen Einstellungen, um einen Quelltext zu übersetzen und auszuführen.

Schauen wir uns an, wie Sie ein Projekt für ein Konsolenprogramm mit C# erstellen. Als Beispiel benutzen wir das Programm „Hallo Welt“. Es gibt den Text `Hallo Welt` auf dem Bildschirm aus und wird traditionell für die ersten Programmierversuche genutzt.

Falls Sie Visual Studio nicht mehr geöffnet haben, starten Sie das Programm bitte. Klicken Sie dann rechts in der Startseite auf den Eintrag **Neues Projekt erstellen**. Alternativ können Sie ein neues Projekt auch auf dem Desktop von Visual Studio über das Symbol **Neues Projekt**  links in der Symbolleiste oder mit dem Eintrag **Neu/Projekt...** im Menü **Datei** anlegen.

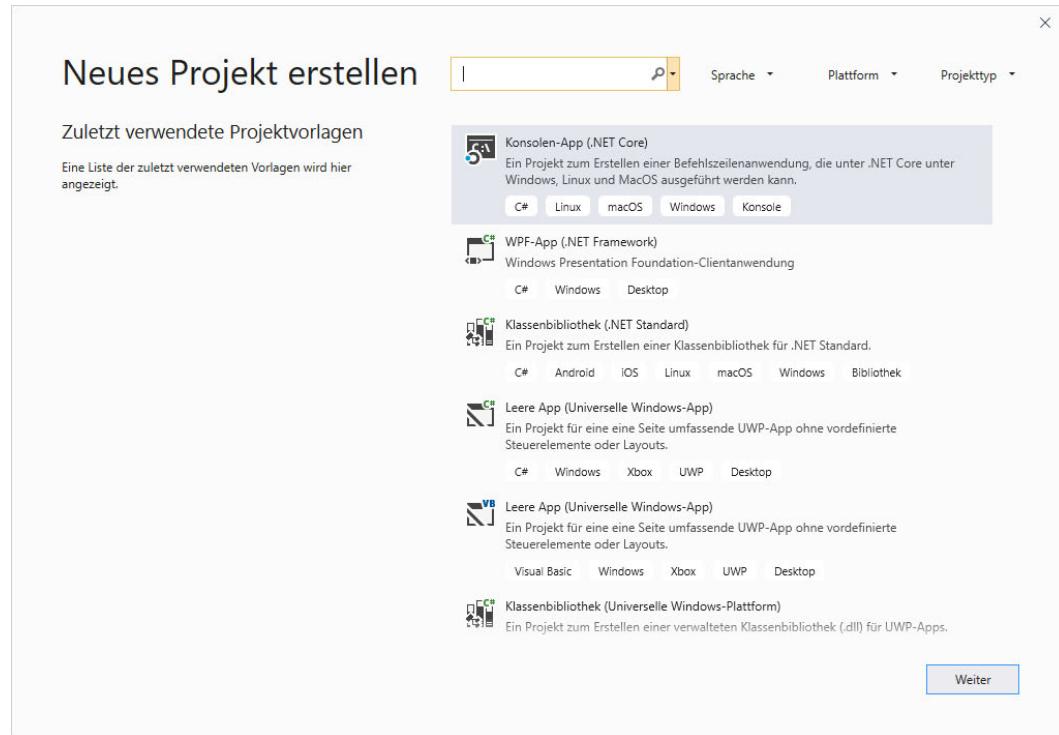


Abb. 3.1: Das Fenster Neues Projekt erstellen

Im Fenster **Neues Projekt erstellen** können Sie jetzt unter verschiedenen Projekttypen für verschiedene Programmiersprachen auswählen. Für ein Konsolenprogramm benötigen wir den Typ **Konsolen-App (.NET Framework)** für die Programmiersprache C#. Sie finden ihn in der Mitte der Liste im rechten Bereich. Markieren Sie den Eintrag bitte.

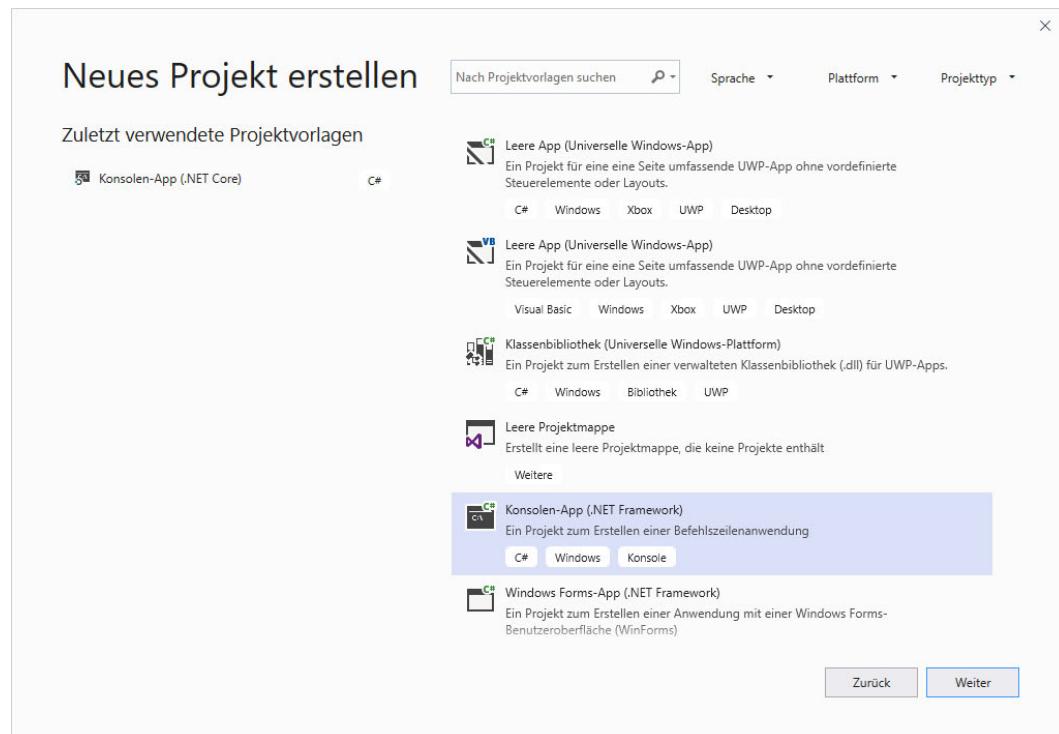


Abb. 3.2: Die Auswahl des Projekttyps

Bitte beachten Sie, dass es mehrere Einträge für Konsolenanwendungen gibt. Wir benötigen eine Konsolenanwendung für C#. Die Programmiersprache wird immer unterhalb des Typs angegeben.



Klicken Sie dann auf **Weiter** unten rechts im Fenster.

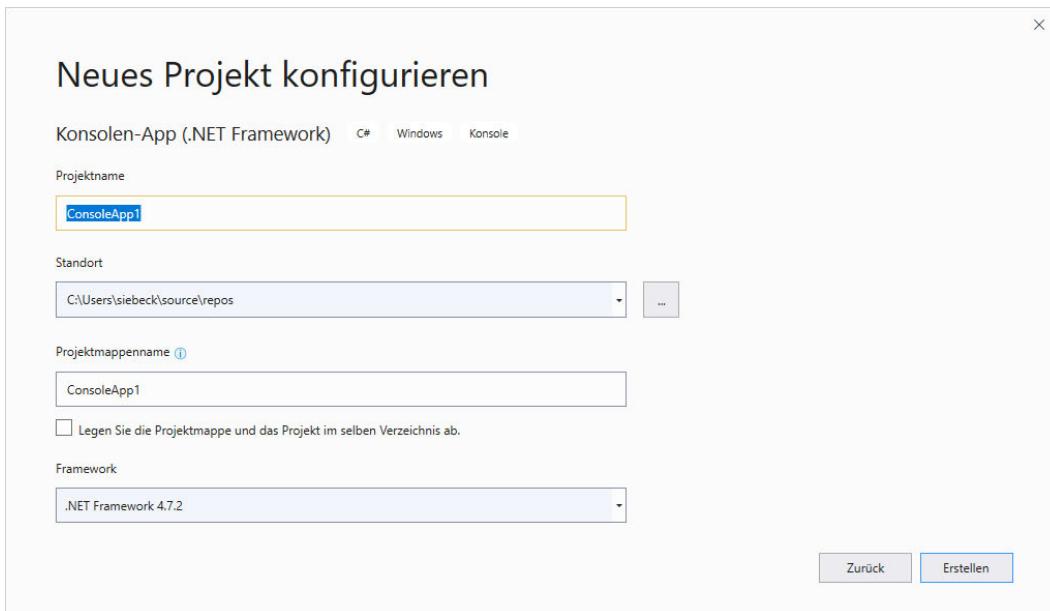


Abb. 3.3: Die Eingabe des Namens

Danach legen Sie den Namen für das Projekt fest. Unser Projekt soll den Namen **Hallo** erhalten. Tragen Sie diesen Namen bitte in das Feld **Projektname** ein. Das Fenster **Neues Projekt konfigurieren** sollte nun ungefähr so aussehen wie in der folgenden Abbildung.

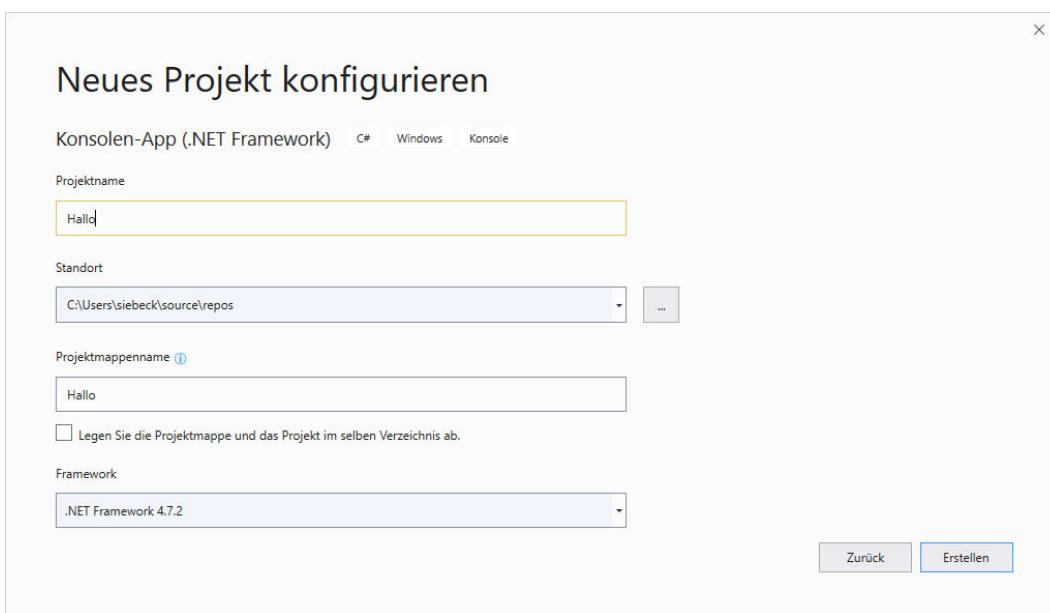


Abb. 3.4: Die Einstellungen für das erste Projekt

Unten im Fenster können Sie auch noch den Ordner, in dem das Projekt gespeichert wird, und den Namen der Projektmappe verändern. In der Regel können Sie die Vorschläge unverändert übernehmen.

Überprüfen Sie noch einmal die Einstellungen und klicken Sie dann auf die Schaltfläche **Erstellen**.

Visual Studio erstellt nun das neue Projekt und zeigt nach einiger Zeit das Quelltextgerüst in der Mitte des Fensters an. Die Datei für dieses Quelltextgerüst trägt den Namen **Program** und hat die Erweiterung **.cs**. Den Dateinamen finden Sie sowohl im Register oberhalb des Quelltextes als auch unten im Projektmappen-Explorer.

Hinweis:

Im Projektmappen-Explorer werden auch noch weitere Zweige mit Dateien angezeigt. Eine dieser Dateien enthält zum Beispiel Informationen zum **Assembly**. Dieses Assembly fasst alle wichtigen Informationen zu einem Projekt zusammen. Da diese weiteren Dateien für uns im Moment nicht wichtig sind, wollen wir uns an dieser Stelle nicht weiter mit ihnen beschäftigen.

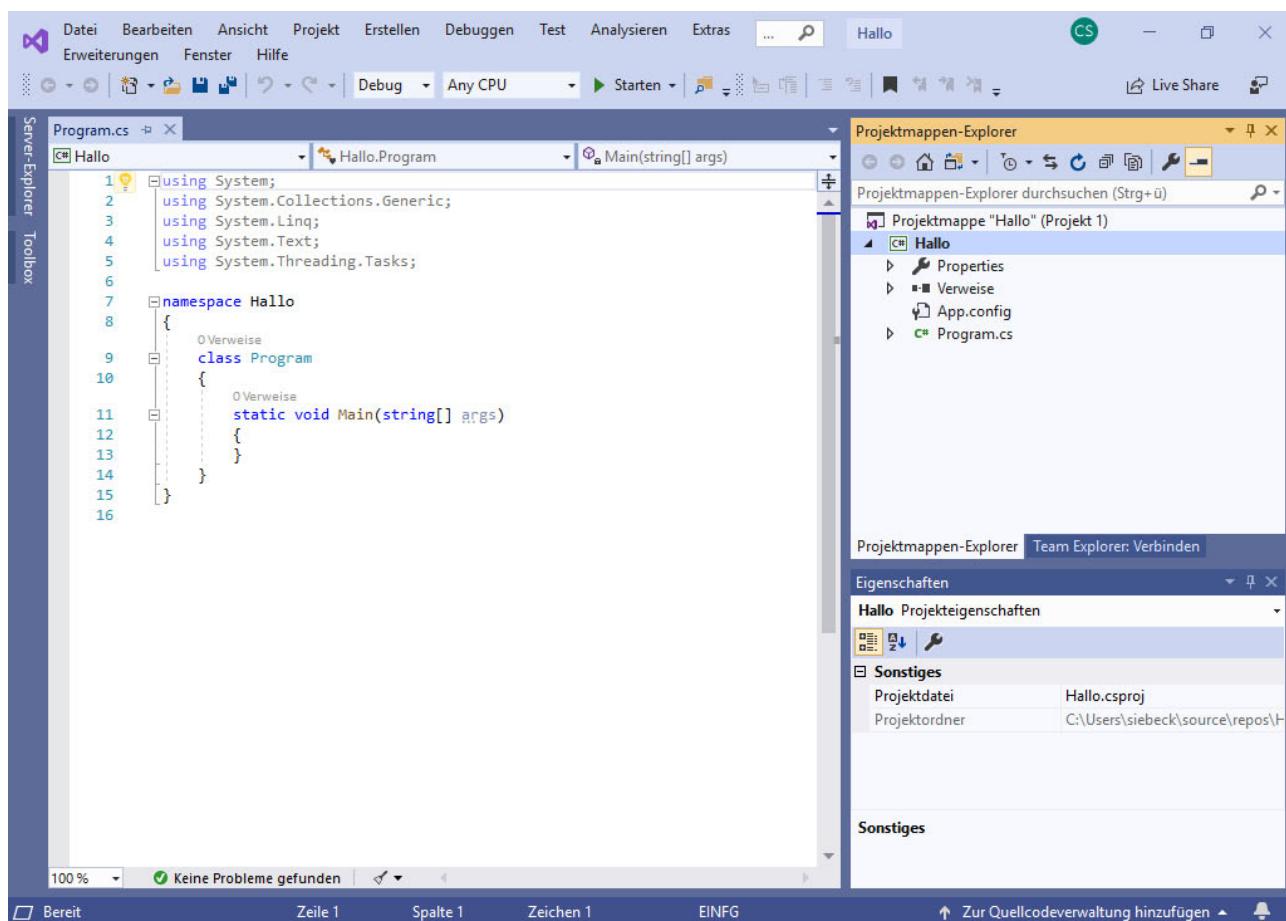
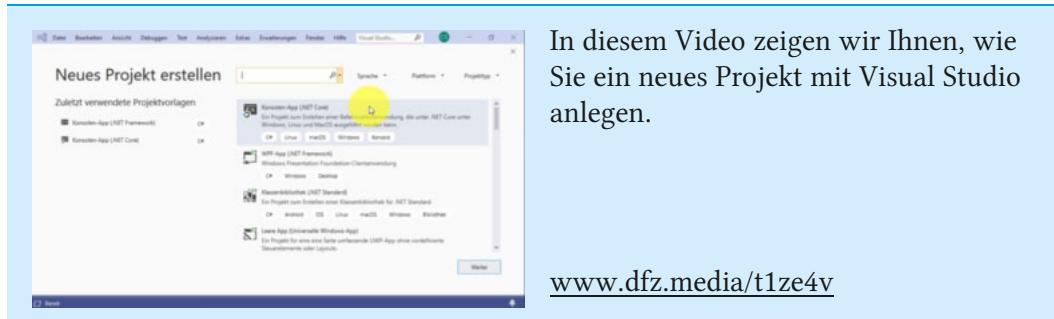


Abb. 3.5: Das neue Projekt



Video 3.1: Neues Projekt anlegen

3.2 Quelltext erfassen

Bei dem Quelltextgerüst, das Visual Studio für das neue Projekt angelegt hat, handelt es sich bereits um ein „fertiges“ Programm, das allerdings nichts weiter macht. Wir sorgen also erst einmal dafür, dass zwei Ausgaben auf dem Bildschirm erscheinen.

Stellen Sie die Einfügemarke hinter die Klammer { unterhalb der Zeile

```
static void Main(string[] args)
```

Drücken Sie dann die Eingabetaste. Der Editor fügt nun eine leere Zeile ein, in die Sie bitte die folgende Anweisung eingeben:

```
Console.WriteLine("Hallo Welt");
```

Der Text im Editor sollte nun so aussehen:

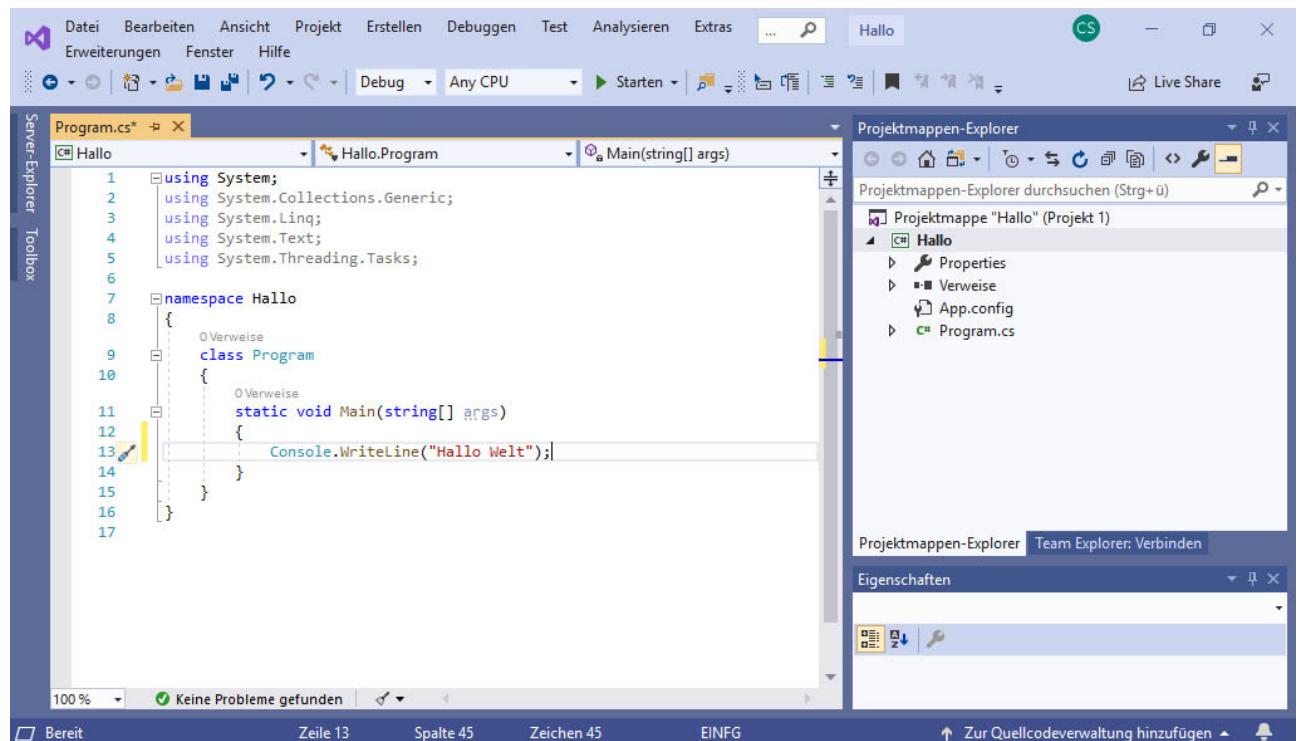


Abb. 3.6: Die erste eigene Anweisung im Editor

Fügen Sie dann hinter dieser Anweisung eine weitere Zeile mit der Eingabetaste ein und geben Sie dort die folgende Anweisung ein:

```
Console.WriteLine("Es grüßt Dich Max Meier");
```

Hinweis:

Den Text Max Meier in der letzten Zeile können Sie natürlich auch durch Ihren eigenen Namen ersetzen.

Der Text im Editor-Fenster sollte nun so aussehen wie im folgenden Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Hallo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hallo Welt");
            Console.WriteLine("Es grüßt Dich Max Meier");
        }
    }
}
```

Code 3.1: Das „Hallo Welt“-Programm

Überprüfen Sie bitte noch einmal, ob Ihr Code exakt so aussieht. Achten Sie dabei vor allem auf die Anführungszeichen, die Punkte, die Klammern und auch die Semikolons am Ende der neuen Zeilen.

Wie Sie sehen, werden Ihre Eingaben vom Editor in unterschiedlichen Farben dargestellt. Diese Farben kennzeichnen bestimmte Elemente des Quelltextes und helfen Ihnen so, den Quelltext leichter zu lesen. In der Standardeinstellung erscheinen zum Beispiel reservierte Wörter in blauer Schrift und Zeichenketten zwischen Anführungszeichen in dunkelroter Schrift.



Reservierte Wörter sind bestimmte Zeichenketten, die vom Compiler verwendet werden – zum Beispiel `int`. Bei `int` handelt es sich um eine Typangabe. Mehr zu den Typangaben erfahren Sie später.

3.3 Projekt speichern und öffnen

Damit Sie das Beispielprogramm nicht immer wieder neu abtippen müssen, sollten Sie es jetzt speichern. Dazu wählen Sie den Befehl **Alles speichern** im Menü **Datei** oder klicken auf das Symbol **Alles speichern**.

Ein wichtiger Hinweis:

Sie sollten sich bei umfangreicherer Quelltexten angewöhnen, die Projektdateien regelmäßig zu speichern. In jedem Fall sollten Sie **vor** dem ersten Übersetzen eine Sicherung durchführen. Andernfalls gehen Ihnen unter Umständen bei Fehlern im Programm sämtliche Änderungen seit dem letzten Speichern verloren. Gerade bei den ersten anspruchsvollerer Programmen müssen Sie damit rechnen, dass sich Fehler einschleichen, die unter Umständen zu einem kompletten Absturz des Rechners führen und einen Neustart erforderlich machen. Zwar speichert Visual Studio den Quelltext vor dem Übersetzen automatisch, trotzdem sollten Sie sich angewöhnen: **Erst speichern, dann ausführen.**

Speichern Sie jetzt bitte das Beispiel aus dem vorigen Code. Wir werden es im nächsten Kapitel ausführen.

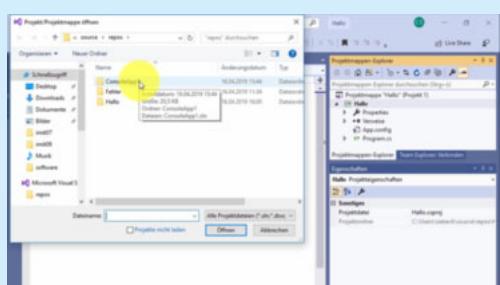
Um ein Projekt wieder zu öffnen, benutzen Sie am einfachsten die Funktion **Offnen/Projekt/Projektmappe...** im Menü **Datei**. Wechseln Sie dann im Dialog **Projekt/Projektmappe öffnen** in den Ordner, in dem Sie das Projekt erstellt haben. Markieren Sie anschließend die Projektappendatei und klicken Sie auf **Öffnen**. Sie erkennen die Projektappendatei an der Erweiterung **.sln** und dem Symbol .

Tipp:

Sie können Projekte auch direkt über die Startseite öffnen. Dazu klicken Sie entweder direkt auf den Namen des Projekts im Bereich **Zuletzt verwendete Elemente öffnen** oder auf den Eintrag **Projekt oder Projektmappe öffnen...** rechts auf der Seite.

Bitte beachten Sie:

Wenn Sie eine Datei direkt öffnen – zum Beispiel über die Funktion **Datei/Öffnen/Datei...** oder über das Symbol **Datei öffnen**  – müssen Sie selbst sorgfältig darauf achten, dass Sie die Projektappendatei mit der Erweiterung **.sln** laden. Wenn Sie nämlich einen Quelltext direkt öffnen, wird auch nur der Quelltext geladen und angezeigt. Übersetzt und ausgeführt werden aber nur die Dateien, die sich im Projekt befinden. Besonders am Anfang sollten Sie daher Ihre Projekte immer mit der Funktion **Datei/Öffnen/Projekt/Projektmappe...** laden oder die entsprechenden Funktionen in der Startseite benutzen.



In diesem Video zeigen wir Ihnen, wie Sie Projekte und Dateien speichern und öffnen.

www.dfz.media/w0z7hh



Video 3.2: Dateien speichern und öffnen

3.4 Kompilieren und Ausführen

Wie Sie bereits wissen, kann der Quelltext, den Sie gerade eingegeben und gespeichert haben, nicht direkt vom Computer ausgeführt werden. Sie müssen ihn erst durch den Compiler übersetzen lassen.

Der Compiler ist ein eigenes Programm, das separat mit verschiedenen Parametern gestartet werden muss. Über diese Parameter wird dem Compiler zum Beispiel mitgeteilt, aus welcher Datei der Quelltext stammt und wie das auszuführende Programm heißen soll.

Da es sich bei Visual Studio aber um eine integrierte Entwicklungsumgebung handelt, können Sie sich diesen Aufwand sparen. Sie müssen lediglich den Befehl **Starten ohne Debugging** wählen. Die gesamte restliche Arbeit nimmt Ihnen dann die IDE ab.

Den Befehl **Starten ohne Debugging** rufen Sie am einfachsten mit der Tastenkombination **Strg + F5** auf. Alternativ können Sie auch die Funktion **Starten ohne Debugging** im Menü **Debuggen** verwenden.



Bitte beachten Sie:

In einer Symbolleiste finden Sie auch ein Symbol **Starten** ►. Wenn Sie auf dieses Symbol klicken, wird das Programm im Debug-Modus ausgeführt. Und das führt zum Beispiel dazu, dass das Fenster der Eingabeaufforderung direkt nach der Ausführung wieder geschlossen wird.

Bei der Übersetzung überprüft der Compiler gleichzeitig auch, ob Ihr Programm syntaktisch korrekt ist. Das heißt, der Compiler kontrolliert, ob Ihre Eingaben den formalen Regeln der Programmiersprache C# entsprechen.

Exkurs: Syntax und Semantik eines Programms

Fehler in einem Programm können auf syntaktischer und auf semantischer Ebene auftreten. Die Syntax bezeichnet die Regeln, die durch die Programmiersprache vorgegeben werden. Die Semantik dagegen bezeichnet die Bedeutung eines Textes – bei einem Quelltext also das, was das Programm ausführen soll – die Funktionalität.

Ein syntaktisch korrekter Text muss nicht zwangsläufig auch semantisch korrekt sein. Ein Beispiel:

Das Auto zog: „Wir laufen keine überzähligen Steine.“

Dieser Satz ist syntaktisch korrekt, da er den formalen Regeln der deutschen Sprache entspricht. Allerdings ergibt der Satz keinen Sinn, da er zahlreiche semantische Fehler enthält.

Der Compiler überprüft vor allem die syntaktische Korrektheit des Programms. Geprüft wird also, ob die Regeln der Programmiersprache C# eingehalten werden. Die semantische Korrektheit – also den Sinn des Programms – kann der Compiler nicht überprüfen. Semantische Fehler werden daher bei der Übersetzung durch den Compiler fast nie gefunden.

Denken Sie daran: Wenn der Compiler Ihren Quelltext ohne Fehlermeldungen übersetzt, haben Sie die Regeln der Programmiersprache C# eingehalten. Eine fehlerfreie Übersetzung ist keine Garantie für ein korrekt funktionierendes Programm!

So viel zu Syntax und Semantik.

Starten Sie jetzt bitte die Übersetzung und das Ausführen des Programms. Rufen Sie dazu den Befehl **Starten ohne Debugging** auf.

Visual Studio beginnt dann mit dem Übersetzen. Wenn Ihnen beim Abtippen des Codes kein Fehler unterlaufen ist, wird nach kurzer Zeit die Eingabeaufforderung mit unserem Beispielprogramm geöffnet.

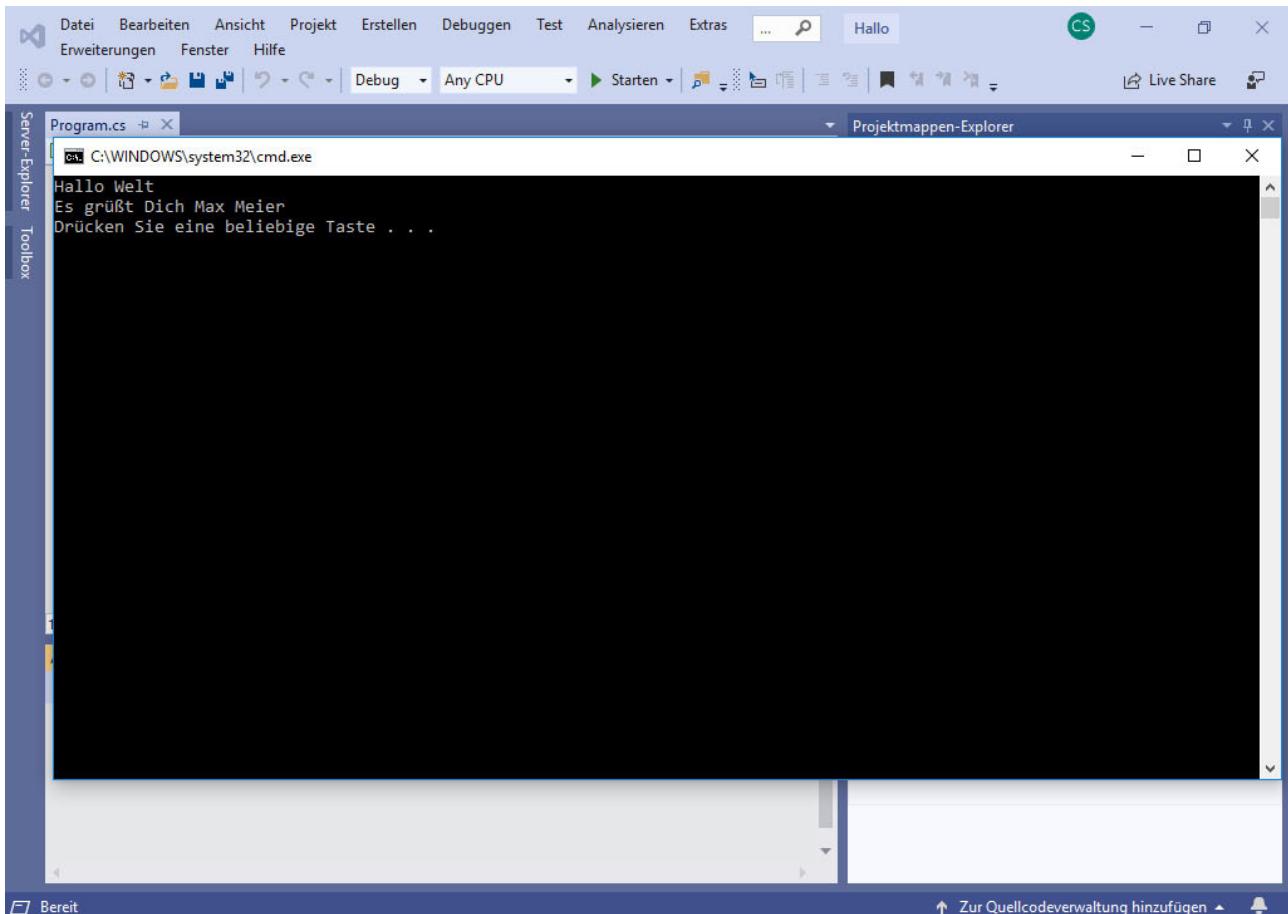


Abb. 3.7: Ausgabe von „Hallo Welt“

Um das Programm zu beenden, drücken Sie die Eingabetaste. Sie gelangen dann automatisch zurück zu Visual Studio.

Hinweis:

Die Meldung

Drücken Sie eine beliebige Taste...

wird beim Ausführen des Programms in der Entwicklungsumgebung automatisch erzeugt. Sie benötigen dafür keine eigene C#-Anweisung.

Die ausführbaren Dateien werden auch auf der Festplatte gespeichert, und zwar unter anderem im Ordner **\bin\Debug** Ihres Projektordners. Für unser Beispiel befindet sich die ausführbare Datei also im Unterordner **\source\repos\Hallo\Hallo\bin\Debug** in Ihrem Benutzerordner. Der Name der Datei entspricht dabei dem Projektnamen. Sie erkennen die ausführbare Datei in einem Explorerfenster an dem Symbol und der Dateierweiterung .exe.

Um eine ausführbare Datei direkt über die Eingabeaufforderung zu starten, öffnen Sie zunächst die Eingabeaufforderung und wechseln dann in den Unterordner im entsprechenden Projektordner. Rufen Sie anschließend die Datei mit dem Kommando *Programmname* auf.

Um unser Beispielprogramm zu starten, geben Sie in der Eingabeaufforderung also erst den Befehl

```
cd users\<benutzername>\source\repos\Hallo\Hallo\bin\Debug
```

für den Wechsel in den Ordner ein. Die Angabe *<benutzername>* ersetzen Sie dabei bitte durch Ihren Benutzernamen.

Anschließend starten Sie das Programm mit

```
Hallo
```



Bitte beachten Sie:

Sie können das Programm auch direkt aus einem Explorerfenster von Windows durch einen Doppelklick auf das Symbol starten. Dabei wird die Eingabeaufforderung aber nach der Ausführung des Programms sofort wieder geschlossen. Das heißt, das Fenster mit dem Programm ist nur ganz kurz auf dem Bildschirm zu sehen.

Später werden Sie erfahren, wie Sie das Fenster so lange geöffnet halten können, bis der Anwender eine Taste drückt.

3.5 Fehlersuche und -behebung

Visual Studio unterscheidet grundsätzlich drei Kategorien von Meldungen, die beim Eingeben eines Quelltextes beziehungsweise beim Übersetzen eines Programms angezeigt werden:

- Hinweise,
- Warnungen und
- Fehler.

Hinweise oder Mitteilungen sind nicht kritisch. Hier finden Sie vor allem zusätzliche Informationen. Das Programm wird trotz der Hinweismeldungen übersetzt und auch ausgeführt.

Warnungen weisen Sie auf Stellen hin, die möglicherweise zu Schwierigkeiten führen können. Eine Warnung kann zum Beispiel erscheinen, wenn Sie etwas im Quelltext ver einbaren, aber nie benutzen. Genau wie bei den Hinweisen führen auch Warnungen nicht zum Abbruch der Übersetzung.

Fehler dagegen verhindern eine Ausführung des Programms. Sie werden zum Beispiel dann angezeigt, wenn Sie beim Schreiben des Quelltextes ein Zeichen in einem Befehl vergessen oder eine Anweisung verwenden, die der Compiler nicht kennt.

Schauen wir uns jetzt genauer an, wie Visual Studio Fehler meldet und wie Sie diese Fehler beheben.

Erstellen Sie bitte ein neues Projekt für eine C#-Konsolenanwendung. Klicken Sie auf das Symbol **Neues Projekt** links in der Symbolleiste oder rufen Sie die Funktion **Neu/Projekt...** im Menü **Datei** auf. Wählen Sie dann im Fenster **Neues Projekt erstellen** in der Liste für die Projekte den Eintrag **Konsolen-App (.NET Framework)** für C# aus. Geben Sie anschließend einen beliebigen Namen für das Projekt ein und klicken Sie auf die Schaltfläche **Erstellen**.

Stellen Sie dann die Einfügemarkie im Quelltext hinter die Klammer { unterhalb der Zeile

```
static void Main(string[] args)
```

und geben Sie die Anweisung

```
Console.WriteLine("Hallo Welt")
```

ein. Lassen Sie dabei das Semikolon am Ende der Zeile ganz bewusst weg.

Visual Studio markiert nach kurzer Zeit das Ende der Zeile mit einer roten Wellenlinie.

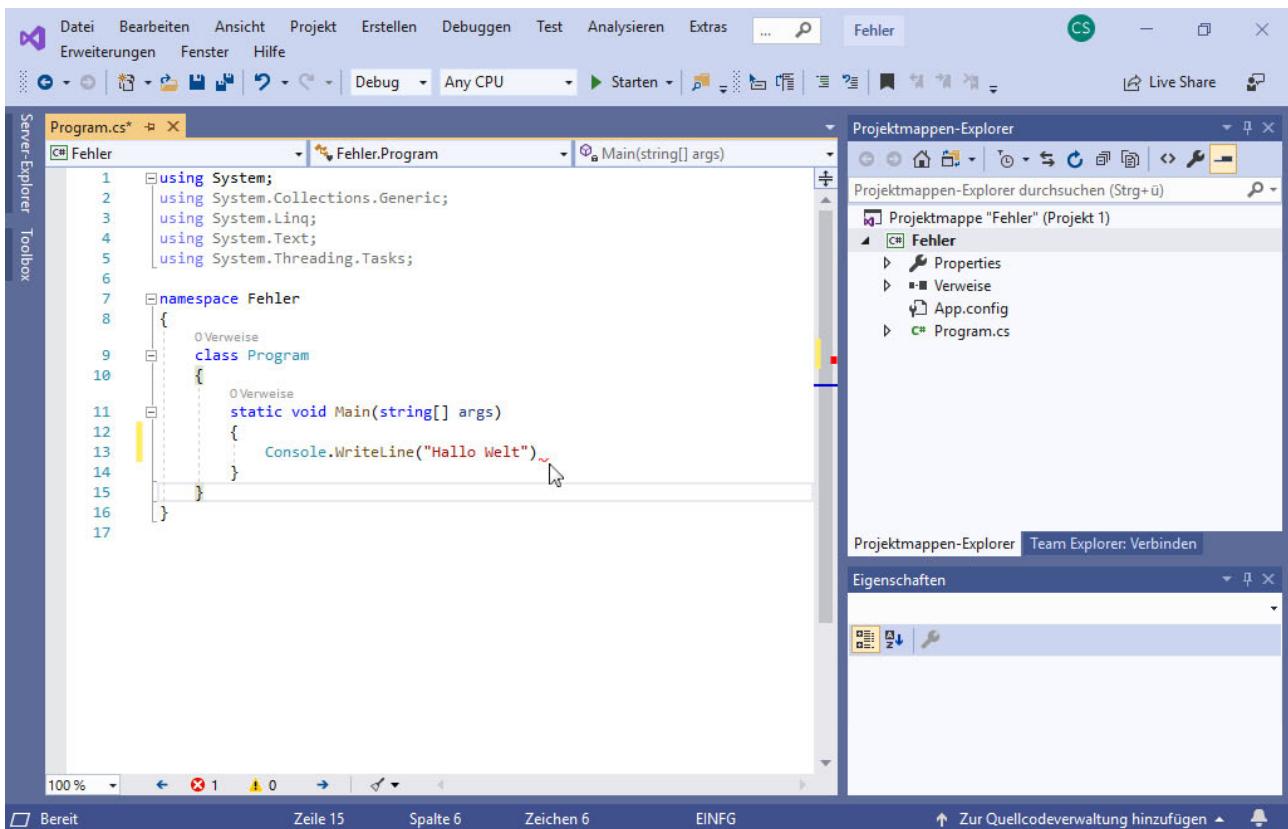


Abb. 3.8: Ein Fehler im Quelltext (der Mauszeiger steht unter der Markierung in der Zeile)

Daran erkennen Sie, dass Visual Studio einen Fehler gefunden hat.

Hinweis:

Die Anzahl der Fehler sehen Sie auch in der Leiste unterhalb des Codes. Fehler werden dabei durch den roten Kreis mit dem weißen Kreuz dargestellt.

Sie können nun den Fehler korrigieren, indem Sie die Zeile korrekt mit einem Semikolon abschließen. Danach sollte auch die Markierung verschwinden.

Sie können die Übersetzung auch starten, wenn Fehler vorhanden sind. Ändern Sie zum Beispiel einmal die Anweisung `WriteLine` in `writeline`.

```
Console.writeline("Hallo Welt");
```

Versuchen Sie dann, das Programm ausführen zu lassen. Rufen Sie dazu den Befehl **Starten ohne Debugging** mit der Tastenkombination **Strg + F5** oder über das Menü **Debuggen** auf. Visual Studio beginnt mit der Übersetzung und bricht nach kurzer Zeit mit einer Fehlermeldung ab.

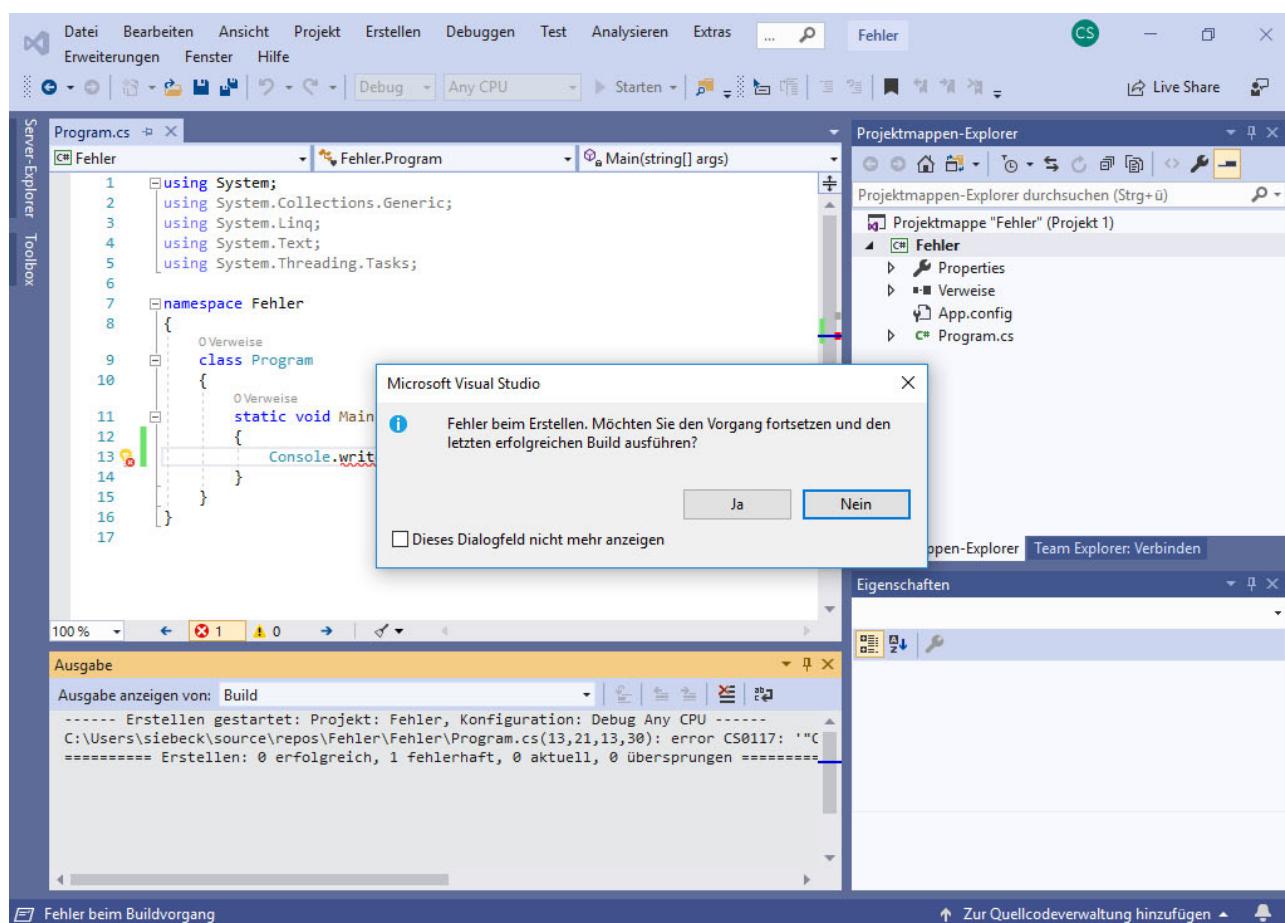


Abb. 3.9: Die fehlgeschlagene Übersetzung

In dem Fenster finden Sie zunächst einmal nur den Hinweis, dass beim Erstellen ein Fehler aufgetreten ist. Außerdem fragt Sie Visual Studio, ob Sie den Vorgang fortsetzen wollen und den letzten erfolgreichen Build¹⁰ ausführen wollen. Wenn Sie hier auf die

10. To build bedeutet wörtlich übersetzt so viel wie „bauen, erstellen“. Gemeint ist damit hier das ausführbare Programm, das aus dem Projekt erstellt wird.

Schaltfläche **Ja** klicken, wird die Version des Programms ausgeführt, die zuletzt erfolgreich übersetzt werden konnte. Das ergibt in den allermeisten Fällen natürlich keinen Sinn, da dann ja keine Änderungen berücksichtigt werden.

In unserem Beispiel ist außerdem auch noch keine lauffähige Version des Programms vorhanden, da die Übersetzung noch nie erfolgreich durchgeführt wurde. Klicken Sie deshalb in dem Dialog bitte auf **Nein**.

Tipp:

Das Fenster mit dem Hinweis auf den Fehler und die Rückfrage können Sie dauerhaft ausblenden lassen. Markieren Sie dazu die Option **Dieses Dialogfeld nicht mehr anzeigen**. Visual Studio bricht dann die Übersetzung bei Fehlern ohne jede Rückfrage ab.

Im unteren Bereich des Fensters wird jetzt die Fehlerliste mit einer Meldung angezeigt. Zusätzlich wird das Wort `writeline` im Quelltext mit einer roten Linie markiert.

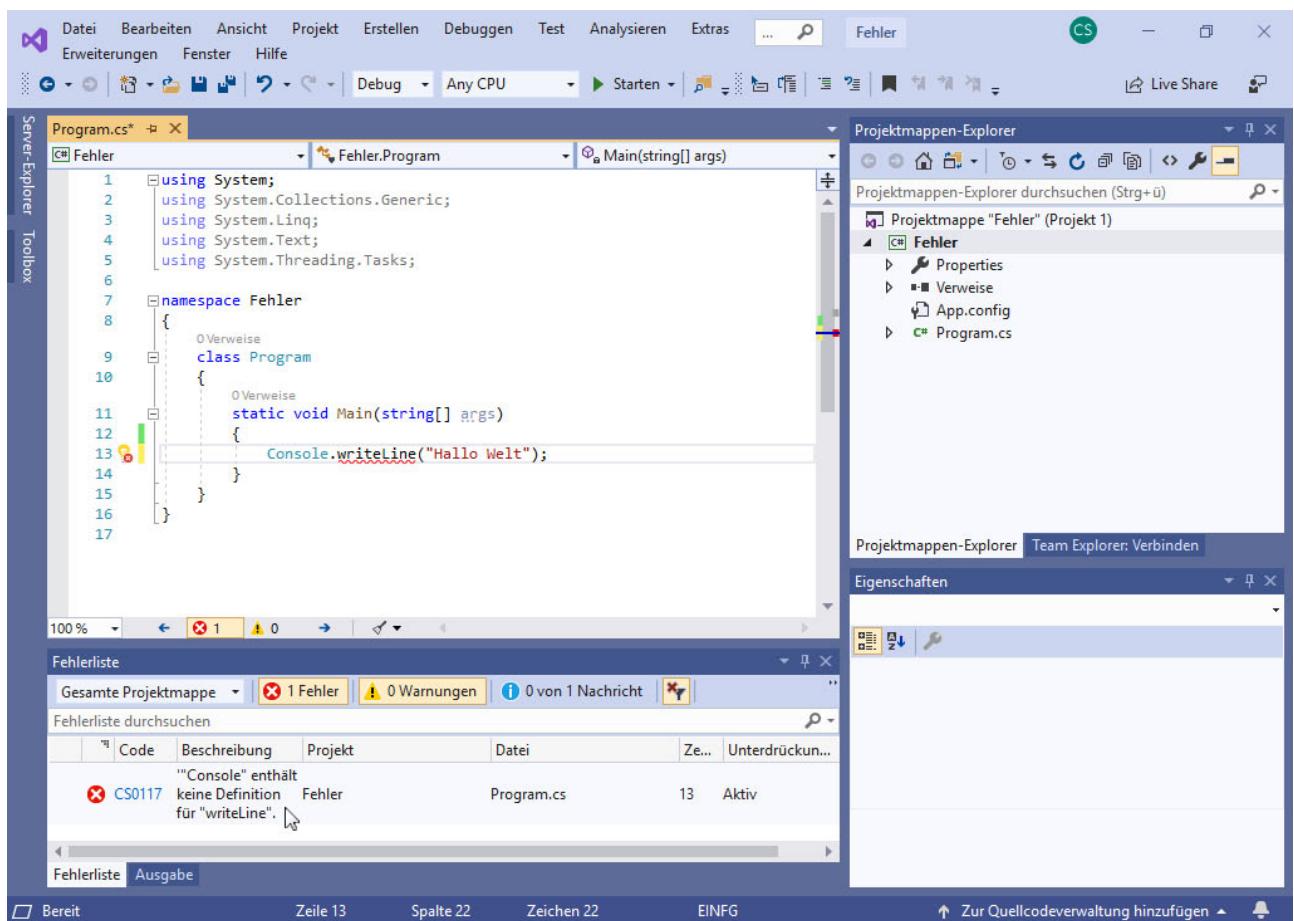


Abb. 3.10: Eine Fehlermeldung in der Fehlerliste (unten am Mauszeiger)

Schauen wir uns diese Meldung etwas genauer an.

Ganz am Anfang finden Sie das Symbol für einen Fehler und den Fehlercode. Danach folgt eine Beschreibung des Fehlers. In unserem Beispiel moniert Visual Studio, dass `Console` keine Definition von `writeline` enthält. Die Anweisung `writeline` ist dem Compiler in dieser Form also nicht bekannt.

Nach der Beschreibung folgen der Name des Projekts und der Datei, in der der Fehler aufgetreten ist. Dahinter steht die Nummer der Zeile, in der Visual Studio den Fehler vermutet. Abgeschlossen wird die Meldung mit Hinweisen zum Unterdrückungszustand. Damit wollen wir uns hier aber nicht weiter beschäftigen.

Um den Fehler zu korrigieren, können Sie durch einen Doppelklick auf die Meldung an die Stelle im Quelltext springen, an der Visual Studio den Fehler vermutet. Bitte beachten Sie aber, dass diese Stelle nicht immer auch tatsächlich den Fehler verursacht. Wenn Sie also beim Auftreten eines Fehlers in der angegebenen Zeile keine Probleme finden können, überprüfen Sie auch die Zeilen unmittelbar vor der Fehlermarkierung.

Tipp:

Die aktuelle Position der Einfügemarkie im Quelltext zeigt Ihnen Visual Studio immer in der Mitte in der Statusleiste an. Hier können Sie also sehr schnell überprüfen, in welcher Zeile und Spalte sich die Einfügemarkie gerade befindet.

Bei einigen Fehlern bietet Ihnen Visual Studio auch selbst Korrekturen an. Das erkennen Sie an der kleinen Glühbirne, die im Code erscheint. Nach dem Anklicken dieser Glühbirne wird der Korrekturvorschlag angezeigt.

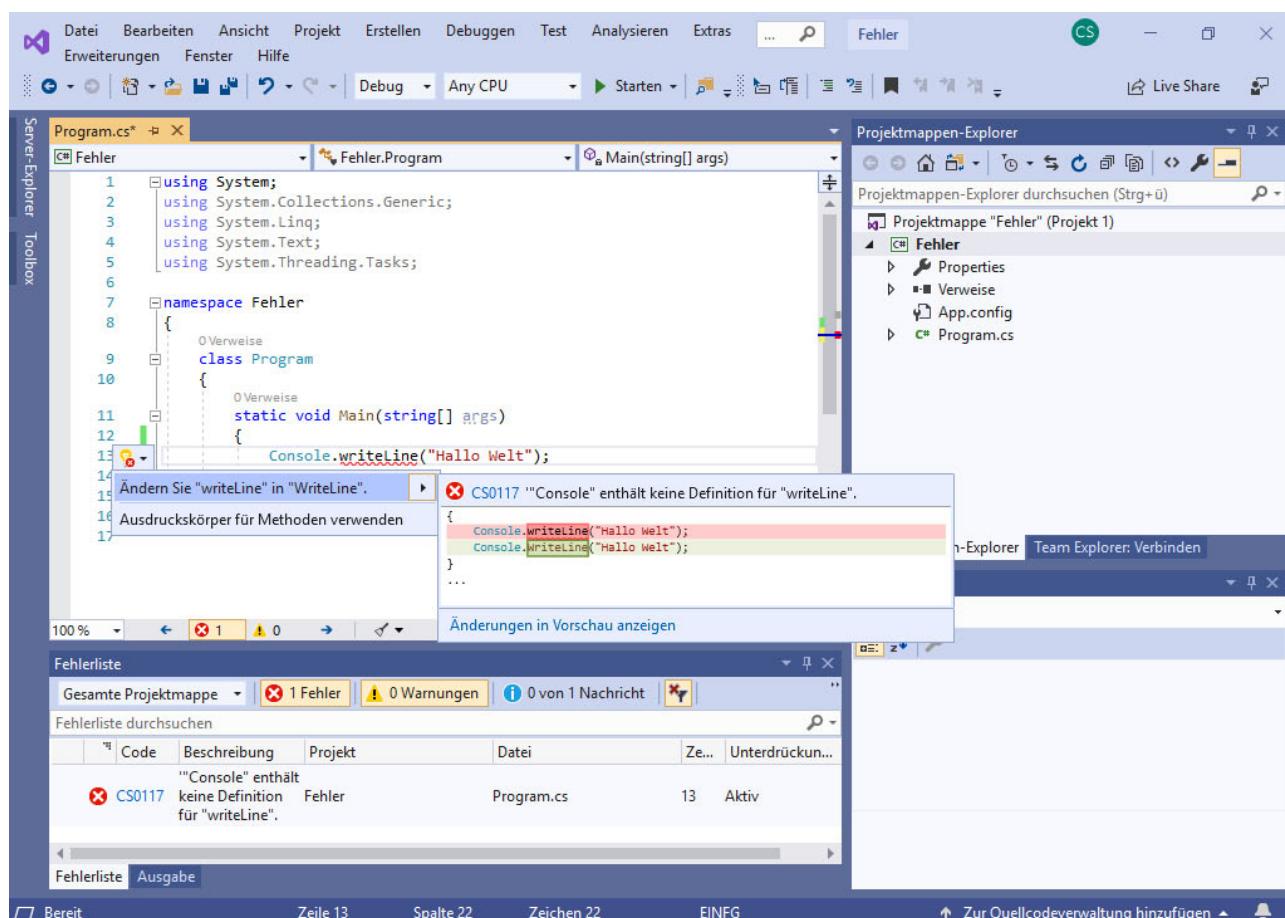


Abb. 3.11: Ein Korrekturvorschlag

Um ihn zu übernehmen, klicken Sie mit der Maus auf den entsprechenden Eintrag in der Liste. Bitte beachten Sie aber, dass die Korrekturvorschläge nicht immer das Problem auch wirklich beheben. Prüfen Sie sie daher gründlich, bevor Sie sie anwenden.

Die Korrekturvorschläge von Visual Studio sorgen nicht in jedem Fall dafür, dass ein Fehler verschwindet. Im Extremfall bauen Sie sich durch die Korrektur sogar neue Fehler ein.



Schauen wir uns noch einen anderen Fehler an. Korrigieren Sie bitte die Schreibweise von `writeline` wieder in `WriteLine`. Löschen Sie dann die öffnende geschweifte Klammer direkt oberhalb der Zeile mit der Anweisung

```
Console.WriteLine.
```

Jetzt werden in der Fehlerliste gleich 10 Fehler angezeigt. Sie beziehen sich allerdings nicht direkt auf den eigentlichen Fehler, sondern weisen nur auf die Auswirkungen der fehlenden Klammer hin.

Tipp:

Wenn Sie sehr viele Fehlermeldungen auf einmal erhalten, konzentrieren Sie sich zunächst auf den ersten angezeigten Fehler. Nach dem Beheben verschwinden dann häufig auch die anderen Meldungen.

Sehen wir uns nun an, was bei einer Warnmeldung geschieht. Fügen Sie bitte zunächst die öffnende geschweifte Klammer wieder oberhalb der Zeile mit der Anweisung `Console.WriteLine` ein. Geben Sie dann direkt hinter dieser Klammer die Zeile

```
int zahl;  
ein.
```

Der Quelltext sollte nun so aussehen:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
namespace Fehler  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int zahl;  
            Console.WriteLine("Hallo Welt");  
        }  
    }  
}
```

Code 3.2: Eine provozierte Warnung

Lassen Sie das Programm anschließend ausführen. Es wird ohne Schwierigkeiten übersetzt und auch in der Eingabeaufforderung angezeigt. Wenn Sie sich aber die Fehlerliste unten im Fenster von Visual Studio genau ansehen, sollte dort eine Warnung angezeigt werden.

Hinweis:

Wenn die Fehlerliste nicht automatisch angezeigt wird, klicken Sie einmal mit der Maus auf das Register **Fehlerliste** unten links im Fenster von Visual Studio.

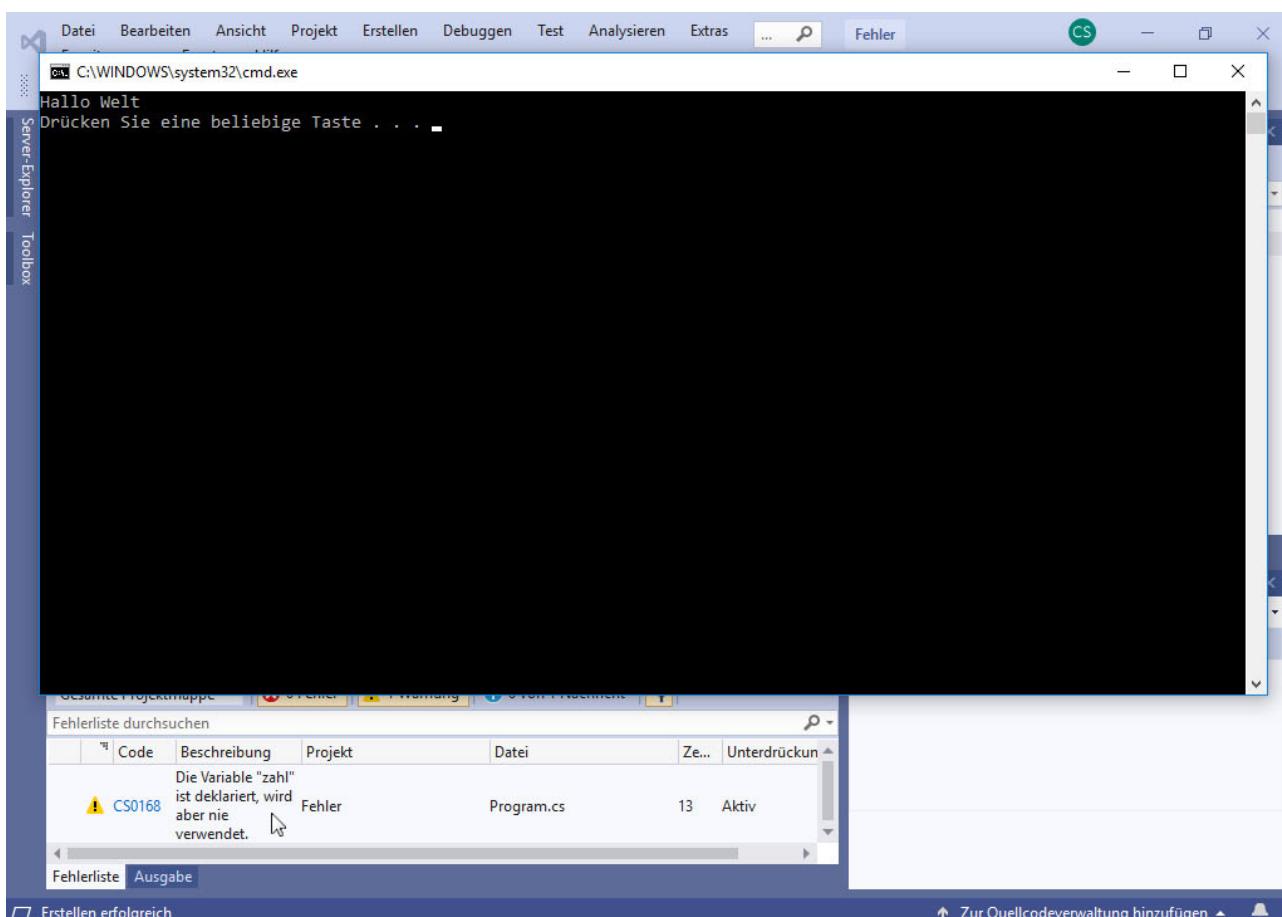
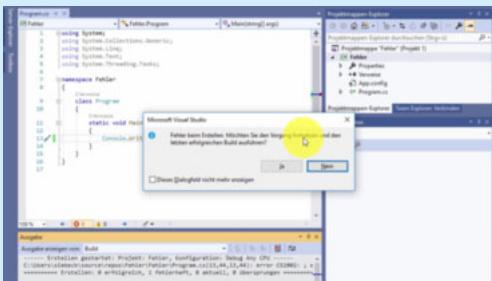


Abb. 3.12: Eine Warnmeldung (unten am Mauszeiger)

Im Quelltext selber wird die Warnung mit einer grünen Wellenlinie angezeigt.

Ein wichtiger Hinweis:

Auch wenn Warnungen scheinbar erst einmal keine Auswirkung haben, sollten Sie sie nicht einfach überlesen. Denn oft machen sich die Schwierigkeiten erst bei der Ausführung des Programms bemerkbar und führen dann zu massiven Problemen. Suchen Sie daher immer nach den Ursachen einer Warnung und beheben Sie sie.



In diesem Video stellen wir Ihnen die Fehler- und Warnmeldungen von Visual Studio vor.



www.dfz.media/buu8ts

Video 3.3: Fehler- und Warnmeldungen

So viel an dieser Stelle zur Anzeige von Fehlern und zur Fehlerbehebung. Experimentieren Sie jetzt ruhig noch weiter mit dem Quelltext. Provozieren Sie weitere Fehler und sehen Sie sich an, wie der Compiler reagiert und wie die Fehler gemeldet werden.

Zusammenfassung

Konsolenprogramme werden unter der Eingabeaufforderung von Windows ausgeführt.

Um ein neues Projekt für ein Konsolenprogramm anzulegen, starten Sie die Funktion **Datei/Neu/Projekt...** oder klicken Sie auf das Symbol **Neues Projekt** . Alternativ können Sie ein neues Projekt auch über die Startseite von Visual Studio anlegen.

Im Editor erfassen Sie den Quelltext.

Um die Dateien eines Projekts zu speichern, benutzen Sie den Befehl **Datei/Alles speichern** oder klicken Sie auf das Symbol **Alles speichern** .

Um ein Projekt wieder zu öffnen, benutzen Sie am einfachsten die Funktion **Öffnen/Projekt/Projektmappe...** im Menü **Datei**.

Um das Programm zu übersetzen und auszuführen, rufen Sie den Befehl **Starten ohne Debugging** mit der Tastenkombination **Strg + F5** oder über das Menü **Debuggen** auf.

Fehler und Warnungen werden in der Fehlerliste angezeigt.

Aufgaben zur Selbstüberprüfung

- ### 3.1 Was ist ein Konsolenprogramm?

- 3.2 Wie erstellen Sie ein neues Projekt für ein Konsolenprogramm? Nennen Sie die nötigen Befehle und Auswahlen.

- 3.3 Im Quelltext wird das Ende einer Zeile mit einer roten Wellenlinie markiert. Wofür steht diese Markierung?

- 3.4 Nennen Sie die drei Kategorien von Meldungen, die in der Fehlerliste von Visual Studio angezeigt werden.

4 Die Hilfefunktionen von Visual Studio Community 2019

Zum Abschluss dieses Studienhefts wollen wir uns die Hilfe von Visual Studio Community 2019 ansehen.

Die Hilfe starten Sie über die Funktion **Hilfe anzeigen** im Menü **Hilfe**.

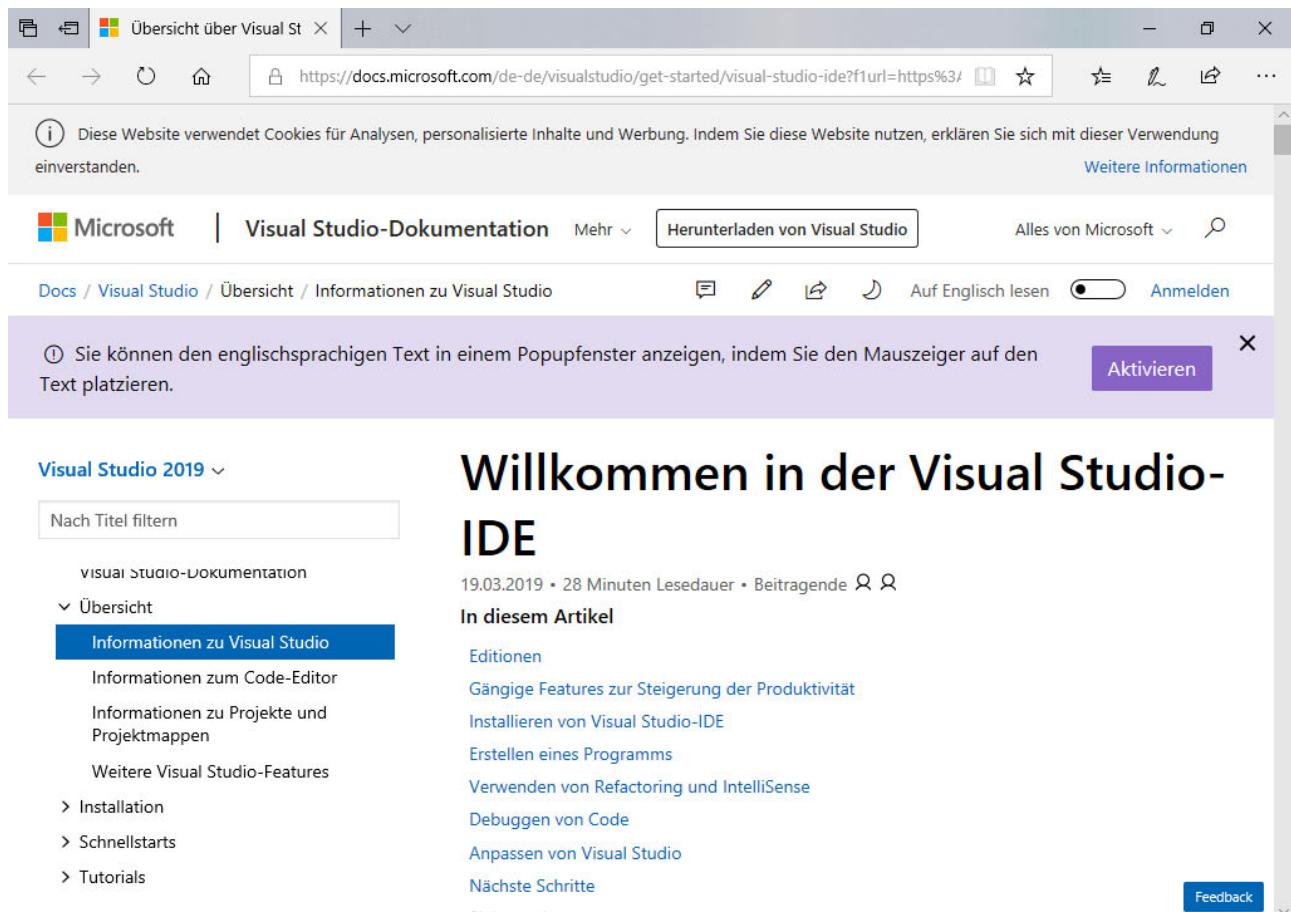


Abb. 4.1: Die Startseite der Hilfe

Die Startseite der Hilfe wird geladen und im Browser angezeigt. Über die Links auf der Seite können Sie nun weitere Hilfethemen aufrufen.

Rufen Sie einmal ein anderes Hilfethema auf. Klicken Sie auf den Eintrag **Informationen zum Code-Editor** im linken Bereich der Seite.

The screenshot shows a web browser window displaying the Microsoft Visual Studio Documentation. The URL is <https://docs.microsoft.com/de-de/visualstudio/get-started/tutorial-editor?view=vs-2019>. The page title is "Einführung in das Bearb." (Introduction to the Editor). A tooltip message in the top right corner says: "① Sie können den englischsprachigen Text in einem Popupfenster anzeigen, indem Sie den Mauszeiger auf den Text platzieren." (You can display the English text in a pop-up window by placing the cursor over the text). Below the tooltip, there is a purple button labeled "Aktivieren". The main content area has a large heading "Informationen zur Verwendung des Code-Editors". To the left, a sidebar menu for "Visual Studio 2019" is visible, with "Informationen zum Code-Editor" being the selected item.

Abb. 4.2: Die Hilfeseite zum Code-Editor

Auf dieser Seite finden Sie jetzt Informationen zum Umgang mit dem Editor. Sehen Sie sich diese Information ruhig einmal ein wenig an.

Interessant sind auch die Tutorials. Sie finden sie im linken Bereich der Seite. Rufen Sie einmal das Tutorial zu C# ab. Sie finden es im Zweig **Tutorials/C#**.

The screenshot shows a Microsoft Docs page titled "Visual Studio-Tutorials | C#". The URL is <https://docs.microsoft.com/de-de/visualstudio/get-started/csharp/?view=vs-2019>. The left sidebar has a "C#-Tutorials" section with links like "Übersicht", "Erstellen Sie eine App", and "Kennenlernen von Visual Studio". The main content area starts with a paragraph about tutorials for beginners, followed by a section titled "Erste Schritte" with two buttons: "Installieren von Visual Studio" and "Einführung beginnen".

Abb. 4.3: Die Tutorial zu C#

Auf dieser Seite können Sie jetzt umfangreiche Informationen zum Programmieren mit C# abrufen. Sehen Sie sich auch hier erst einmal ein wenig um – allein schon, um einen kleinen Vorgeschmack zu bekommen.

4.1 Suchen in der Hilfe

Um die Hilfe zu durchsuchen, klicken Sie auf das Symbol oben rechts in der Hilfeseite. Geben Sie dann einen Suchbegriff in das Suchfeld ein. Klicken Sie anschließend auf das Symbol hinten im Suchfeld oder drücken Sie die Eingabetaste .

Hinweis:

Unter Umständen wird ganz links im Suchfeld der Eintrag **VS IDE** angezeigt. Damit wird die Suche nur auf die Entwicklungsumgebung beschränkt. Für eine vollständige Suche sollten Sie die Einschränkung löschen. Klicken Sie dazu auf das Symbol hinter dem Eintrag.

Probieren Sie das einmal aus. Geben Sie **Console.WriteLine** als Suchbegriff ein und starten Sie die Suche. Denken Sie daran, vorher gegebenenfalls die Einschränkung der Suche zu löschen.

The screenshot shows a Microsoft Edge browser window with the following details:

- Title Bar:** "Suche | Microsoft Docs" is displayed in the title bar.
- Address Bar:** The URL is https://docs.microsoft.com/de-de/search/index?search=Console.WriteLine.
- Header:** The Microsoft logo, "Docs" (highlighted in blue), and other navigation links like Windows, Microsoft Azure, Visual Studio, Office, and Mehr are visible.
- Content Area:**
 - Section Header:** "27,807 Suchergebnisse für 'Console.WriteLine'"
 - First Result:**
 - Link:** [Console.WriteLine Method \(System\)](#)
 - Category:** System / Console
 - Description:** Schreibt die angegebenen Daten, gefolgt vom aktuellen Zeichen für den Zeilenabschluss, in den Standardausgabestream.Writes the specified data, followed by the current line terminator, to the standard output stream.
 - Update Info:** aktualisiert 12.04.2019
 - Second Result:**
 - Link:** [Workflow-Designer - WriteLine-Aktivitätsdesigner - Visual Studio](#)
 - Category:** Entwickeln von Apps mit dem Workflow-Designer / Aktivitätsdesigner / Primitive Aktivitätsdesigner
 - Description:** Die WriteLine Aktivitäts-Designer dient zum Erstellen und Konfigurieren einer WriteLine Aktivität.The WriteLine activity designer is used to create and configure a WriteLine activity.
 - Update Info:** aktualisiert 12.04.2019
 - Third Result:**
 - Link:** [StreamWriter.WriteLine Method \(System.IO\)](#)
 - Category:** System.IO / StreamWriter
 - Description:** Beschreibung nicht verfügbar
 - Update Info:** aktualisiert 12.04.2019
 - Fourth Result:**
 - Link:** [WebPageTraceListener.WriteLine Method \(System.Web\)](#)
 - Category:** System.Web / WebPageTraceListener
 - Description:** Schreibt eine Meldung in eine Webseite oder in den ASP.NET-Ablaufverfolgungs-Viewer.Writes a message to a Web page or to the

Abb. 4.4: Die Suchergebnisse

Nach kurzer Zeit werden die Suchergebnisse angezeigt. Durch einen Mausklick auf den Titel eines Dokuments können Sie dann das vollständige Dokument anzeigen lassen.

So viel zum Suchen in der Hilfe.

4.2 Weitere nützliche Hilfefunktionen

Sie können auch direkt aus dem Editor oder aus einem anderen Bereich von Visual Studio auf verschiedene Hilfefunktionen zugreifen.

Eine Variante ist dabei die **kontextsensitive Hilfe**. Sie liefert Ihnen Unterstützung zu dem Element, in dem sich die Einfügemarkierung gerade befindet. Das kann entweder ein Befehl sein, ein Bereich des Desktops oder auch eine Meldung. Gestartet wird die kontextsensitive Hilfe mit der Funktionstaste **F1**.

Probieren Sie das einmal aus. Laden Sie noch einmal das Projekt **Hallo**, das wir im letzten Kapitel angelegt haben. Klicken Sie dann mit der Maus auf die Anweisung `WriteLine` im Quelltext. Drücken Sie anschließend die Taste **F1**.

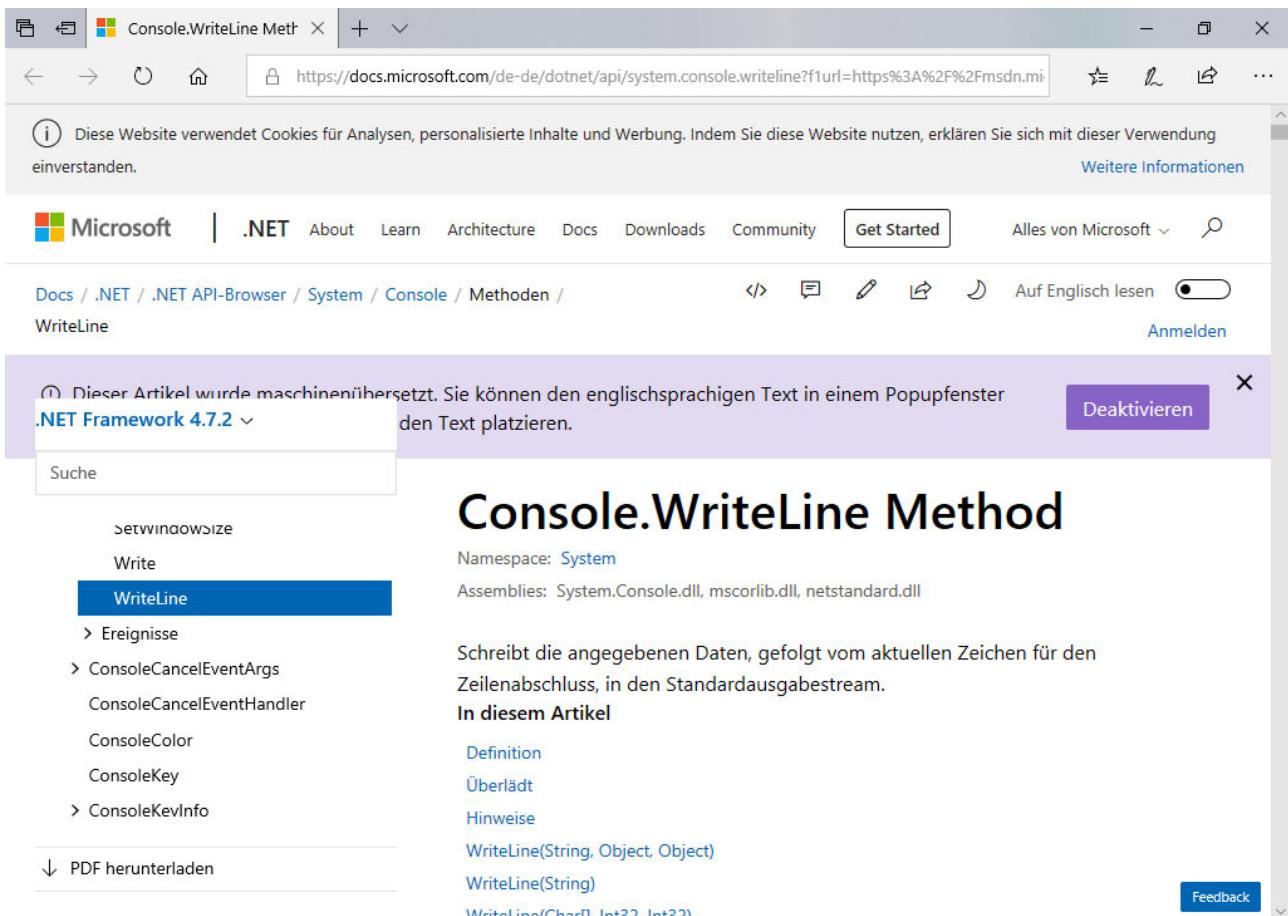


Abb. 4.5: Das Hilfedokument zur Methode `Console.WriteLine`

In diesem Hilfedokument können Sie jetzt Details zur Methode `Console.WriteLine` nachlesen.

Für viele Teile in einem Quelltext können Sie außerdem noch **Quickinfos** anzeigen lassen. Dazu stellen Sie den Mauszeiger auf das gewünschte Element und warten einen kurzen Moment ab. In der Quickinfo können Sie dann in sehr kompakter Form weitere Informationen nachlesen.

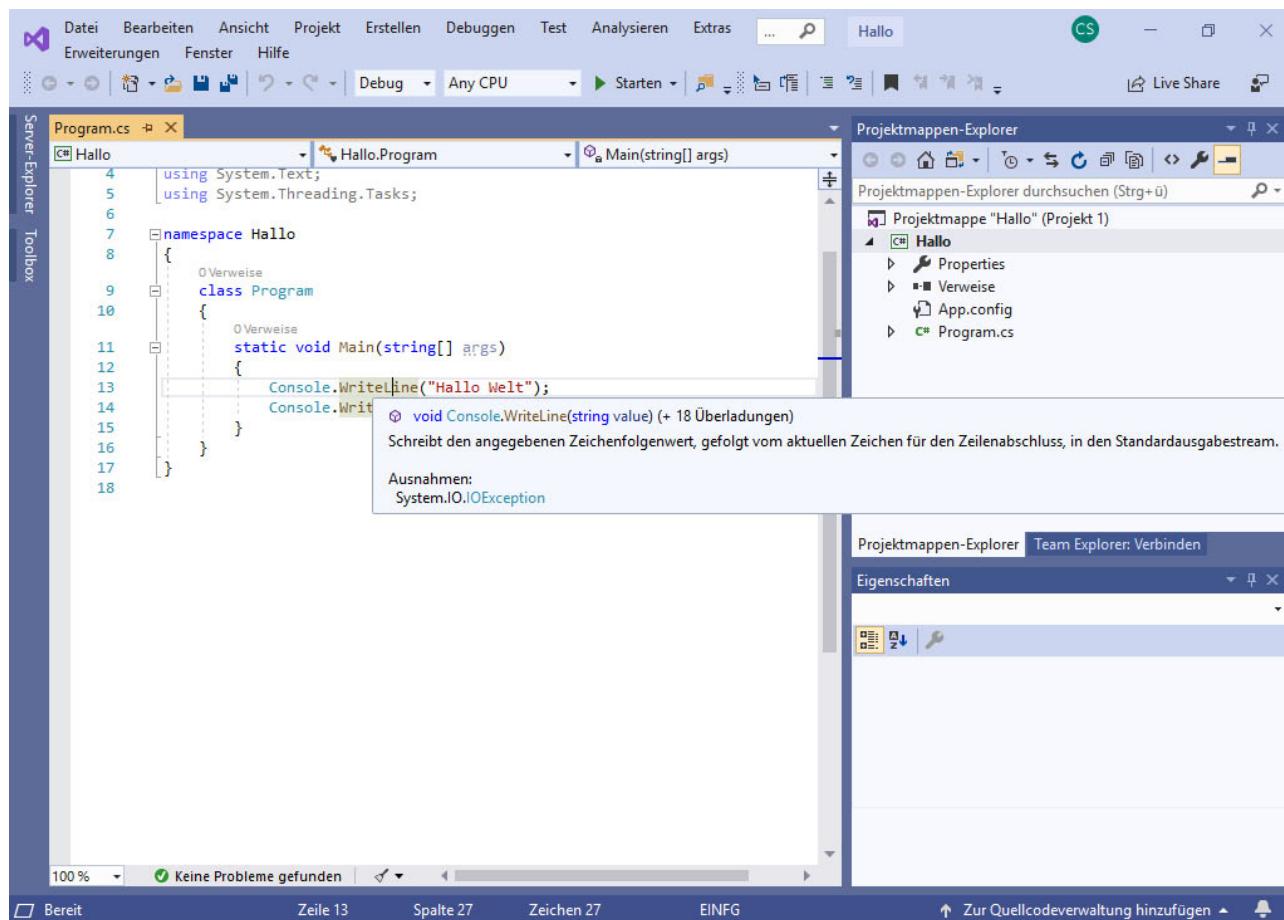


Abb. 4.6: Die Quickinfos zu WriteLine()

Für die ersten Schritte bei der Programmierung sind die Quickinfos häufig viel zu kompakt. Sie bieten aber später eine sehr interessante Möglichkeit, wenn Sie kurz nachsehen wollen, in welcher Form Sie zum Beispiel Daten an eine Methode übergeben müssen.

Eine weitere Hilfe ist die **Memberliste**¹¹ für ein Objekt. Diese Memberliste erscheint automatisch beim Eingeben von Quelltexten und schlägt Ihnen anhand Ihrer Eingaben mögliche Eigenschaften oder Methoden für ein Objekt vor.

Wahrscheinlich ist Ihnen die Memberliste schon beim Eingeben des ersten Programms aufgefallen. Probieren Sie sie aber noch einmal gezielt aus. Legen Sie im Quelltext eine neue Zeile an. Geben Sie anschließend den Text `Console.` ein. Visual Studio zeigt Ihnen dann die Memberliste an.

11. *Member* bedeutet übersetzt so viel wie „Mitglied“.

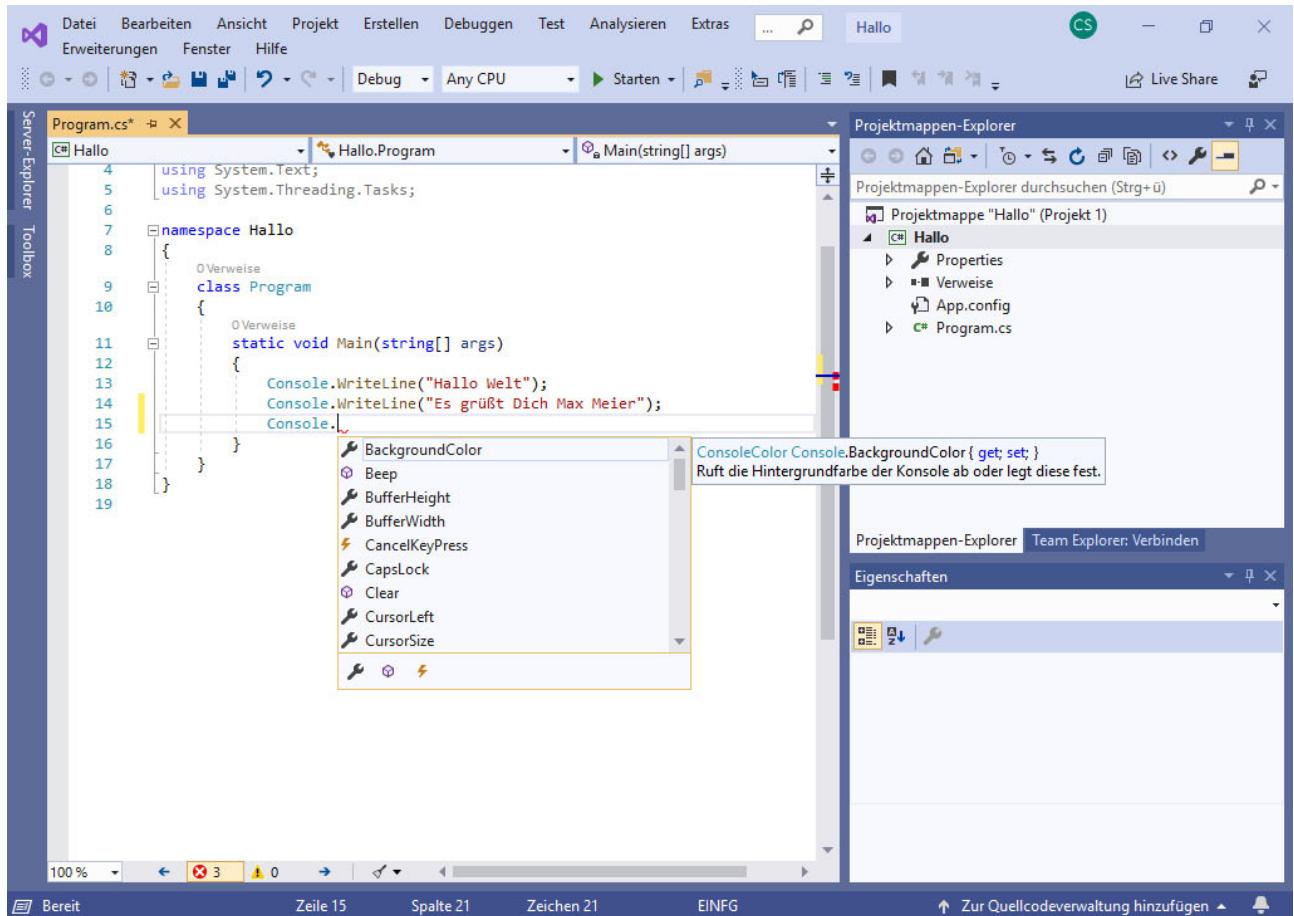


Abb. 4.7: Die Memberliste für Console

Aus dieser Liste können Sie jetzt mit einem Mausklick den gewünschten Eintrag in den Quelltext übernehmen. Wenn Sie weiterschreiben, werden die Einträge in der Liste übrigens automatisch aktualisiert.

Tipp:

An den Symbolen vor einem Eintrag in der Memberliste können Sie erkennen, ob es sich um eine Eigenschaft, eine Methode oder ein Ereignis handelt. Die Eigenschaft `CursorVisible` legt zum Beispiel fest, ob der Cursor angezeigt wird oder nicht. Eine Methode dagegen führt etwas aus. Mit der Methode `Clear()` löschen Sie zum Beispiel den Inhalt der Konsole.

Die Eigenschaften erkennen Sie am Symbol , die Methoden am Symbol und die Ereignisse am Symbol .

Mit Eigenschaften, Methoden und Ereignissen werden wir uns im weiteren Verlauf noch sehr intensiv auseinandersetzen.

Eine weitere Hilfefunktion ist die Livecodeanalyse. Hier zeigt Ihnen Visual Studio Hinweise an, wie Sie Ihren Code verbessern können. Diese Hinweise erkennen Sie an einer kleinen Glühbirne. In unserem Beispielprojekt sind zum Beispiel einige Anweisungen direkt zu Beginn überflüssig. Sie können sie automatisch löschen lassen.

Probieren Sie das einmal aus. Klicken Sie mit der Maus in eine der ausgegraut dargestellten Anweisungen oben im Code. Links vor der Zeile erscheint eine kleine Glühbirne.

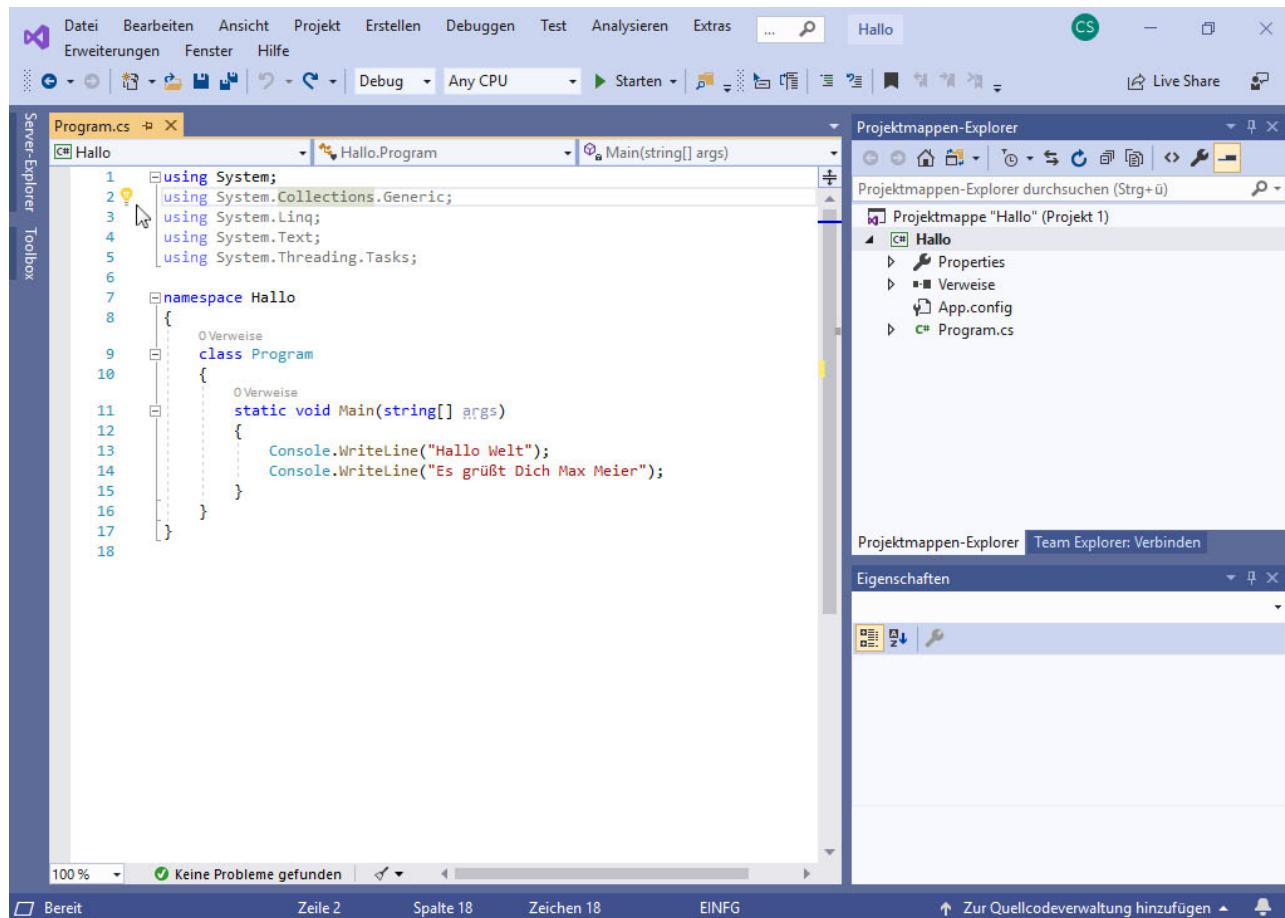


Abb. 4.8: Die Livecodeanalyse (links oben am Mauszeiger)

Durch einen Mausklick auf das Symbol können Sie sich jetzt Vorschläge anzeigen lassen.

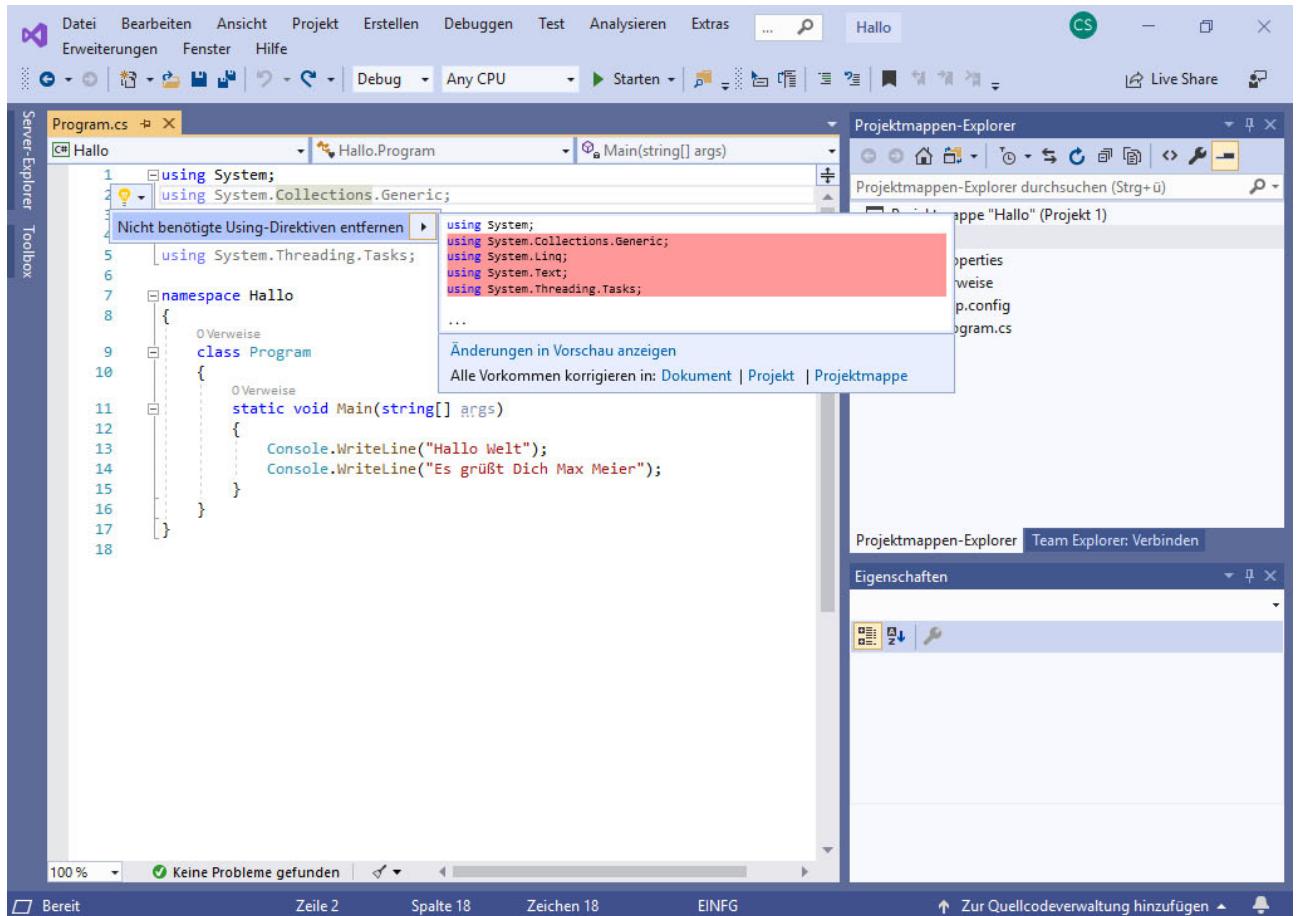


Abb. 4.9: Die Vorschläge der Livecodeanalyse

Durch einen weiteren Mausklick können Sie die Aktionen starten. Probieren Sie auch das aus. Lassen Sie die überflüssigen Anweisungen löschen.

So viel zur Hilfe.

The screenshot shows the Microsoft Visual Studio Documentation website with the following details:

- Header:** Microsoft | Visual Studio-Dokumentation | Herunterladen von Visual Studio | Alle von Microsoft | Dokumente | Visual Studio | Überblick | Informationen zum Code-Editor
- Left Sidebar:** Visual Studio 2019 – Nach Titel filtern, Kategorien: Informationen zu Visual Studio, Informationen zum Code-Editor (highlighted), Informationen zu Projekte und Projektmappen, Weitere Visual Studio-FEATURES.
- Main Content:** **Informationen zur Verwendung des Code-Editors** (30.11.2017 – 11 Minuten Lesedauer – Beitragsende 25). In diesem Artikel: Erstellen einer neuen Codebasis, Verwenden von Code-Completion, Automatisches Code, Reduzieren von Codefehlern, Anzeigen von Symbolinformationen, Verwenden von IntelliSense zum Vervollständigen von Wörtern, Refactoring eines Namens, Nächste Schritte.

In diesem Video zeigen wir Ihnen, wie Sie mit der Hilfe von Visual Studio arbeiten.

www.dfz.media/b60doo

dfz.media/b60doo

Video 4.1: Arbeiten mit der Hilfe

Zusammenfassung

Die Hilfe starten Sie über das Menü Hilfe.

Über die kontextsensitive Hilfe können Sie gezielt Hilfe zu einem Befehl, einer Meldung oder einem Bereich des Desktops anfordern.

Über die Quickinfos und die Memberliste können Sie direkt im Editor weitere Informationen anzeigen lassen.

Aufgaben zur Selbstüberprüfung

- 4.1 Woher stammen die Inhalte der Hilfe?

- 4.2 Wie können Sie in der Hilfe suchen?

- 4.3 Beim Übersetzen eines Quelltextes wird Ihnen eine Fehlermeldung angezeigt. Wie können Sie weitere Informationen zu der Meldung abrufen?

Schlussbetrachtung

Herzlichen Glückwunsch!

Sie haben erfolgreich die ersten Schritte beim Programmieren mit C# unternommen. Sie haben Visual Studio auf Ihrem Computer installiert und wissen, wie man ein neues Projekt anlegt und die Hilfe benutzt. Außerdem haben Sie Ihr erstes Programm erstellt, kompiliert und ausgeführt.

Üben Sie den Umgang mit Visual Studio noch ein wenig. Legen Sie neue Projekte an und testen Sie auch die verschiedenen Hilfefunktionen ausgiebig. Wenn Sie wollen, können Sie auch den Desktop von Visual Studio so anpassen, dass er Ihren Anforderungen optimal gerecht wird.

„Trauen“ Sie sich auch, absichtlich ein paar Fehler in ein Programm einzubauen, und sehen Sie sich die Reaktionen von Visual Studio an.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Programmiersprachen wie C, C# und Pascal gehören zu den höheren Programmiersprachen.
- 1.2 Zu den wichtigsten Vorläufern von C# gehören die Programmiersprachen C++ und Java.
- 1.3 Die beiden wesentlichen Vorteile des .NET Frameworks sind:
 - Programme können theoretisch mit einer beliebigen Programmiersprache auf einer beliebigen Plattform entwickelt werden.
 - Dem Programmierer stehen zahlreiche vorgefertigte Komponenten und Funktionen zur Verfügung.

Kapitel 2

- 2.1 Zu den wichtigsten Programmen der integrierten Entwicklungsumgebung gehören der Editor und der Compiler.
- 2.2 Um einen Bereich automatisch in den Hintergrund zu stellen, klicken Sie auf das Symbol **Automatisch im Hintergrund** .

Kapitel 3

- 3.1 Ein Konsolenprogramm ist ein Programm, das unter der Eingabeaufforderung von Windows ausgeführt wird.
- 3.2 Ein neues Projekt für ein Konsolenprogramm legen Sie über die Funktion **Datei/Neu/Projekt...** oder über das Symbol **Neues Projekt**  an. Sie können auch den Eintrag **Neues Projekt erstellen...** in der Startseite benutzen.
Im Fenster **Neues Projekt erstellen** wählen Sie in der Liste für die Projekte den Eintrag **Konsolen-App (.NET Framework)** für C# aus. Geben Sie anschließend einen beliebigen Namen für das Projekt ein und klicken Sie auf die Schaltfläche **Erstellen**.
- 3.3 Die rote Wellenlinie markiert einen Fehler.
- 3.4 In der Fehlerliste von Visual Studio werden Fehler, Warnungen und Meldungen angezeigt.

Kapitel 4

- 4.1 Die Inhalte der Hilfe werden aus dem Internet geladen.
- 4.2 Um die Hilfe zu durchsuchen, klicken Sie auf das Symbol  oben rechts in der Hilfeseite. Geben Sie dann einen Suchbegriff in das Suchfeld ein. Klicken Sie anschließend auf das Symbol  hinten im Suchfeld oder drücken Sie die Eingabetaste.
- 4.3 Sie stellen die Einfügemarkie in die Fehlermeldung und drücken die Taste **F1**.

B. Glossar

.NET Framework	<p>Das <i>.NET Framework</i> schafft Rahmenbedingungen für das plattformunabhängige Programmieren. Dazu werden die Programme zunächst in eine Zwischensprache (CIL) übersetzt. Diese Zwischensprache wird dann auf jedem einzelnen Computer durch einen <i>Just-in-Time-Compiler</i> (JIT) in ein ausführbares Programm umgewandelt. Der Just-in-Time-Compiler selbst ist Bestandteil einer Laufzeitumgebung – der <i>Common Language Runtime</i> (CLR).</p> <p>Das <i>.NET Framework</i> stellt außerdem zahlreiche vorgefertigte Funktionen und Steuerelemente für die Entwicklung zur Verfügung.</p>
Assembler	<p>Ein Assembler ist eine Programmiersprache, die auf einer sehr niedrigen Ebene arbeitet. Die Programmierung erfolgt über <i>Mnemonics</i>.</p> <p>Die Übersetzung in ein ausführbares Programm erfolgt ebenfalls durch den Assembler.</p>
CIL	Die CIL (<i>Common Intermediate Language</i>) ist eine Zwischensprache für die Ausführung von <i>.NET Framework</i> -Anwendungen. Aus der Zwischensprache werden durch den <i>Just-in-Time-Compiler</i> ausführbare Programme erzeugt.
CLI	CLI steht für <i>Common Language Infrastructure</i> . Es handelt sich um eine gemeinsame Basis für das Erstellen von Computerprogrammen – unabhängig von einer bestimmten Programmiersprache und auch von einer bestimmten Plattform.
CLR	<p>Die CLR (<i>Common Language Runtime</i>) ist die Laufzeitumgebung für <i>.NET Framework</i>-Anwendungen. Sie enthält unter anderem den JIT-Compiler.</p> <p>Die Laufzeitumgebung muss auf jedem Rechner vorhanden sein, der ein Programm, das für das <i>.NET Framework</i> erstellt wurde, ausführen soll.</p>
Common Intermediate Language	Siehe CIL.
Common Language Infrastructure	Siehe CLI.
Common Language Runtime	Siehe CLR.
Compiler	Ein Compiler setzt einen Quelltext in eine ausführbare Form um. Die gesamte Übersetzung erfolgt vor der Ausführung.

Editor	Ein Editor ist ein Programm zum Erfassen und Bearbeiten von Informationen.
Hochsprache	Eine Hochsprache ist eine Programmiersprache, die sich zum Teil eng an der menschlichen Sprache orientiert. Viele Hochsprachen unterstützen auch Konstruktionen wie Schleifen und die objektorientierte Programmierung.
IDE	Abkürzung für <i>Integrated Development Environment</i> . Siehe integrierte Entwicklungsumgebung.
Integrated Development Environment	<i>Integrated Development Environment</i> ist der englische Begriff für Integrierte Entwicklungsumgebung.
Integrierte Entwicklungsumgebung	Eine integrierte Entwicklungsumgebung fasst mehrere Werkzeuge zur Programmierung unter einer Oberfläche zusammen – zum Beispiel den Editor, den Compiler und den Linker.
Interpreter	Ein Interpreter setzt einen Quelltext oder Zwischencode in ein ausführbares Programm um. Die Umsetzung erfolgt zur Laufzeit des Programms.
JIT-Compiler	Ein JIT-Compiler (<i>Just in time</i>) erzeugt aus den Anweisungen einer Zwischensprache ausführbare Programme.
Just-in-time-Compiler	Siehe JIT-Compiler.
Kontextsensitive Hilfe	Die kontextsensitive Hilfe von Visual Studio liefert Ihnen Unterstützung zu dem Element, in dem sich die Einfügemarkie gerade befindet. Gestartet wird die kontextsensitive Hilfe mit der Funktionstaste F1 .
Linux	Linux ist ein Betriebssystem. Es ist, anders als die Windows-Betriebssysteme, ursprünglich nicht von einem Unternehmen entwickelt worden, sondern entstand durch die Zusammenarbeit vieler Programmierer. Bei Linux handelt es sich um ein Open-Source-Produkt. Das bedeutet, dass das Betriebssystem an sich kostenlos erhältlich ist und beliebig geändert werden darf. Linux ist in verschiedenen Distributionen erhältlich.
Main()	<code>Main()</code> ist die Hauptmethode eines C#-Konsolenprogramms.
Mnemonic	Ein <i>Mnemonic</i> ist ein Befehl eines Assemblers – zum Beispiel MOV.
Objektorientierte Programmierung	Bei der objektorientierten Programmierung werden Daten- und Verhaltensaspekte gemeinsam betrachtet. Das wesentliche Element der objektorientierten Programmierung ist das Objekt.

Projekt	Ein Projekt in Visual Studio fasst alle Elemente einer Anwendung zusammen. Es gibt verschiedene Arten von Projekten.
Projektmappendatei	Die Projektmappendatei fasst alle Dateien eines Projekts zusammen. Sie erkennen die Projektmappendatei an der Erweiterung .sln.
Quelltext	Quelltext sind die Anweisungen eines Programms, so wie sie im Editor eingegeben werden.
Quickinfos	Quickinfos sind kurze Hinwestexte. Sie werden normalerweise angezeigt, wenn der Mauszeiger einen kurzen Moment auf einem bestimmten Element stehen bleibt.
Semantik	Die Semantik beschreibt bei einem Quelltext die Funktionalität der Anwendung.
Software-Engineering	Der Begriff „Software-Engineering“ beschreibt allgemein die systematische und ingenieurmäßige Entwicklung von Software unter Einsatz von Prinzipien, Methoden und Werkzeugen. Das Software-Engineering entstand als Reaktion auf die Softwarekrise.
Softwarekrise	Der Begriff „Softwarekrise“ entstand Ende der 60er-Jahre. Er beschreibt die mangelhafte Qualität der Software und die Schwierigkeiten bei der Entwicklung.
Software-Technik	Software-Technik ist der deutsche Begriff für Software-Engineering.
Syntax	Die Syntax bezeichnet die Regeln, die durch die Programmiersprache vorgegeben werden.
Syntaxfehler	Syntaxfehler sind Verstöße gegen die Regeln der Programmiersprache – zum Beispiel ein fehlendes Semikolon oder eine falsche Klammer. Syntaxfehler werden vom Compiler bei der Übersetzung des Programms gemeldet.
Toolbox	Die Toolbox ist ein Teil der Oberfläche von Visual Studio. Sie ermöglicht das komfortable Einfügen von Steuerelementen.
Tooltipp	Siehe Quickinfos.
UML	UML steht als Abkürzung für <i>Unified Modeling Language</i> .
Unified Modeling Language (UML)	Die <i>Unified Modeling Language</i> ist eine objektorientierte Modellierungssprache für Softwaresysteme.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch.*

Spracheinführung, Objektorientierung, Programmiertechniken.
8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019.* Ideal für Programmieranfänger.
6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Stammbaum von C#	4
Abb. 2.1	Die Sicherheitswarnung beim Ausführen	8
Abb. 2.2	Der erste Schritt des Installers	8
Abb. 2.3	Installation der ersten Dateien	9
Abb. 2.4	Die Auswahl der Workloads	9
Abb. 2.5	Die Anmeldung	10
Abb. 2.6	Die Auswahl des Farbschemas	11
Abb. 2.7	Die Startseite	12
Abb. 2.8	Der Desktop von Visual Studio Community 2019	13
Abb. 2.9	Der Desktop ohne den Projektmappen-Explorer	14
Abb. 2.10	Der Projektmappen-Explorer im Hintergrund	15
Abb. 2.11	Das Positionieren eines Bereichs auf dem Desktop	16
Abb. 2.12	Das Fenster zur Anmeldung	17
Abb. 2.13	Die Eingabe für das Konto	18
Abb. 2.14	Die Anmeldung über das Microsoft-Konto	18
Abb. 2.15	Die aktualisierten Informationen	19
Abb. 3.1	Das Fenster Neues Projekt erstellen	22
Abb. 3.2	Die Auswahl des Projekttyps	22
Abb. 3.3	Die Eingabe des Namens	23
Abb. 3.4	Die Einstellungen für das erste Projekt	23
Abb. 3.5	Das neue Projekt	24
Abb. 3.6	Die erste eigene Anweisung im Editor	25
Abb. 3.7	Ausgabe von „Hallo Welt“	29
Abb. 3.8	Ein Fehler im Quelltext (der Mauszeiger steht unter der Markierung in der Zeile)	31
Abb. 3.9	Die fehlgeschlagene Übersetzung	32
Abb. 3.10	Eine Fehlermeldung in der Fehlerliste (unten am Mauszeiger)	33
Abb. 3.11	Ein Korrekturvorschlag	34
Abb. 3.12	Eine Warnmeldung (unten am Mauszeiger)	36
Abb. 4.1	Die Startseite der Hilfe	39
Abb. 4.2	Die Hilfeseite zum Code-Editor	40
Abb. 4.3	Die Tutorial zu C#	41
Abb. 4.4	Die Suchergebnisse	42
Abb. 4.5	Das Hilfedokument zur Methode Console.WriteLine	43

Abb. 4.6	Die Quickinfos zu WriteLine()	44
Abb. 4.7	Die Memberliste für Console.....	45
Abb. 4.8	Die Livecodeanalyse (links oben am Mauszeiger).....	46
Abb. 4.9	Die Vorschläge der Livecodeanalyse	47

E. Codeverzeichnis

Code 3.1	Das „Hallo Welt“-Programm	26
Code 3.2	Eine provozierte Warnung	35
Code H.1	Einsendeaufgabe 1.1	61

F. Medienverzeichnis

Video 2.1	Der Desktop von Visual Studio	17
Video 3.1	Neues Projekt anlegen	25
Video 3.2	Dateien speichern und öffnen	27
Video 3.3	Fehler- und Warnmeldungen	37
Video 4.1	Arbeiten mit der Hilfe	47

G. Sachwortverzeichnis

A	Mitteilung	30
Arbeitsbereich	13	
Assembler	3	
Assembly	24	
C	Mnemonic	3
C#	4	
Common Intermediate Language (CIL)	5	
Common Language Infrastructure (CLI)	5	
Common Language Runtime (CLR)	5	
Compiler	3	
D	Produkt	
Desktop	12	
anpassen	14	
F	registrieren	17
Fehler	31	
H	Projekt	21
Hilfe	konfigurieren	23
kontextsensitive	43	
Hilfefunktion	39	
Hinweis	30	
Hochsprache	3	
I	Neues erstellen	21
Installation	7	
Interpreter	4	
J	öffnen	27
Just-in-time-Compiler (JIT)	5	
K	speichern	26
Kompilieren und Ausführen	28	
L	Projektmappen-Explorer	13
Linker	3	
M	Q	
Memberliste	44	
R	Quelltext	3
Rapid Application Development (RAD)	5	
S	erfassen	25
Semantik	29	
Startseite	11	
Syntax	28	
T	W	
Tutorial	40	
Warnung	30	

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP01D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

- Korrigieren Sie die Fehler im folgenden Quelltext, sodass sich das Programm mit Visual Studio übersetzen und ausführen lässt. Führen Sie die Fehlerkorrektur mithilfe von Visual Studio aus. Benutzen Sie dazu den Compiler beziehungsweise den Editor und achten Sie auf die Fehlermeldungen.

Korrigieren Sie immer nur einen Fehler – und zwar in der Reihenfolge, in der Visual Studio die Fehler angibt.

Geben Sie zur Lösung dieser Aufgabe bitte auch den Fehler an und schreiben Sie die jeweils erste Fehlermeldung auf. Fügen Sie Ihrer Lösung bitte auch den vollständig korrigierten Quelltext bei.

Wenn Sie den Code nicht selbst abschreiben wollen, können Sie auch das Projekt **Cshp01dAufgabe** benutzen. Sie finden es im heftbezogenen Download-Bereich der Online-Lernplattform.

```
/* ##### Einstudeaufgabe 1.1 #####
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Cshp01cAufgabe
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Dieses Programm ist nun fehlerfrei");
        }
    }
}
```

Code H.1: Einstudeaufgabe 1.1

50 Pkt.

2. Erläutern Sie in einigen Sätzen den Unterschied zwischen Syntax und Semantik.
15 Pkt.
3. Welche Überprüfung führt der Compiler bei der Übersetzung eines Quelltextes durch: Prüfung der Syntax oder der Semantik?
10 Pkt.
4. Beim Eingeben beziehungsweise bei der Übersetzung eines Programms werden folgende Markierungen im Quelltext angezeigt:
 - a) rote Wellenlinie
 - b) grüne WellenlinieWofür stehen diese Markierungen? Welche Auswirkungen haben sie auf die Übersetzung? Begründen Sie Ihre Antworten.
15 Pkt.

5. C# kennt eine Anweisung, um die Einfügemarken in der Konsole zu positionieren. Finden Sie diese Anweisung mit der Memberliste für `Console` heraus.

Ein kleiner Tipp: Einfügemarken heißt übersetzt *cursor*.

10 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Programmaufbau, Variablen, Konstanten,
Datentypen und Operatoren, Ein- und Ausgabe

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0919N01

CSHP02D

Objektorientierte Software-Entwicklung mit C#

Programmaufbau, Variablen, Konstanten, Datentypen und Operatoren, Ein- und Ausgabe

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Programmaufbau, Variablen, Konstanten, Datentypen und Operatoren, Ein- und Ausgabe

Inhaltsverzeichnis

Einleitung	1
1 Programmaufbau und Kommentare	3
1.1 Grundsätzlicher Aufbau eines C#-Programms	3
1.2 Kommentare	5
Zusammenfassung	7
2 Bildschirmausgabe	9
2.1 Bestandteile des „Hallo Welt“-Programms	9
2.2 Textausgabe ohne und mit Zeilenumbruch	11
2.3 Escape-Sequenzen	12
Zusammenfassung	15
3 Operatoren, Variablen und symbolische Konstanten	17
3.1 Arithmetische Operatoren	17
3.2 Variablen	22
3.3 Namenskonventionen für Bezeichner	25
3.4 Ein komplexeres Beispiel	27
3.5 Weitere Operatoren	29
3.6 Symbolische Konstanten	33
Zusammenfassung	35
4 Datentypen	37
4.1 Darstellung von Werten im Computer	37
4.2 Die Datentypen von C#	41
4.2.1 Datentypen für ganze Zahlen	41
4.2.2 Datentypen für Gleitkommazahlen	44
4.2.3 Datentypen für Zeichen und Zeichenketten	49
4.2.4 Der Datentyp für logische Werte	51
4.3 Die Datentypen des .NET Frameworks	52
Zusammenfassung	53

5 Arbeiten mit Variablen unterschiedlicher Datentypen	55
5.1 Die Zuweisung von float-Werten	55
5.2 Zuweisungen zwischen Variablen unterschiedlichen Typs	55
5.3 Typecasting	56
Zusammenfassung.....	62
6 Eingabe	63
Zusammenfassung.....	68
7 Die Gestaltung von Konsolenanwendungen	69
Schlussbetrachtung	71
Anhang	
A. Lösungen der Aufgaben zur Selbstüberprüfung	72
B. Glossar	76
C. Literaturverzeichnis	80
D. Abbildungsverzeichnis	81
E. Tabellenverzeichnis	82
F. Codeverzeichnis	83
G. Sachwortverzeichnis.....	84
H. Einsendeaufgabe	87

Einleitung

In diesem Studienheft steigen wir „richtig“ in die Programmierung ein.

Sie lernen:

- wie ein C#-Programm grundsätzlich aufgebaut ist,
- aus welchen Elementen ein C#-Quelltext besteht,
- wie Sie Kommentare in Ihren Quelltext einfügen,
- wie Sie Textausgaben auf dem Bildschirm durchführen,
- wie Sie arithmetische Operationen durchführen,
- wie Sie Daten in einem Programm in Variablen zwischenspeichern,
- wie Sie mit symbolischen Konstanten arbeiten,
- welche Datentypen C# und das .NET Framework unterstützen,
- wie Sie mit Variablen unterschiedlicher Datentypen arbeiten,
- wie Sie die Typumwandlung von Variablen erzwingen,
- wie Sie Daten über die Tastatur einlesen und
- wie Sie die Ausgabe in der Eingabeaufforderung gestalten.

Christoph Siebeck

1 Programmaufbau und Kommentare

In diesem Kapitel lernen Sie, wie ein C#-Programm grundsätzlich aufgebaut ist. Außerdem erfahren Sie, wie Sie Kommentare in Ihren Quelltext einfügen.

1.1 Grundsätzlicher Aufbau eines C#-Programms

Genau wie andere Sprachen folgen auch Programmiersprachen bestimmten formalen Regeln. Während in anderen Sprachen – wie zum Beispiel Deutsch – eine gewisse Freiheit in der Beachtung dieser Regeln besteht, müssen Sie sich in einer Programmiersprache immer exakt an die Vorgaben halten. Diese exakten Regeln sind erforderlich, damit der Compiler bei der Übersetzung des Quelltextes weiß, was er machen soll. Schon kleinste Abweichungen von den Regeln führen dazu, dass Visual Studio die Übersetzung mit einer Fehlermeldung abbricht.

Die formalen Regeln beziehen sich ausschließlich auf die **syntaktische Korrektheit** eines Quelltextes. Ein syntaktisch korrekter Quelltext kann von Visual Studio in ein ausführbares Programm übersetzt werden. Das ausführbare Programm kann aber trotzdem reichlich logische Fehler enthalten.



Merken Sie sich daher:

Wenn Visual Studio Ihr Programm übersetzt, können Sie lediglich sicher sein, dass Ihnen keine syntaktischen Fehler unterlaufen sind. Die Funktionalität des Programms müssen Sie selbst testen.

Sehen wir uns jetzt den formalen Aufbau eines C#-Quelltextes am kleinsten möglichen C#-Programm unter Visual Studio an:

```
class Program
{
    static void Main(string[] args)
    {
    }
}
```

Code 1.1: Das kleinste mögliche C#-Programm unter Visual Studio

Das Programm besteht aus insgesamt vier Elementen:

- der Klassenvereinbarung `class Program`,
- den beiden geschweiften Klammerpaaren `{` und `}` sowie
- der Zeile `static void Main(string[] args)`

Schauen wir uns diese Elemente der Reihe nach an:

Die Anweisung `class Program` vereinbart eine Klasse mit dem Namen `Program`. Was sich genau hinter diesen Klassen verbirgt und welche Bedeutung die einzelnen Angaben haben, erklären wir Ihnen noch ausführlich im weiteren Verlauf des Lehrgangs.



Hier sollten Sie sich zunächst einmal merken, dass jedes C#-Programm aus mindestens einer Klasse besteht.

Mit der Zeile

```
static void Main(string[] args)
```

wird eine **Methode** `Main()` vereinbart. Diese Methoden enthalten die Anweisungen, die ein C#-Programm ausführen soll. Eine sehr wichtige Methode ist dabei `Main()`. Sie wird bei der Ausführung zuerst abgearbeitet und bildet damit quasi den Startpunkt für ein Programm.

Die Anweisungen, die von der Methode ausgeführt werden, stehen im **Anweisungsteil** zwischen den beiden geschweiften Klammern `{` und `}`. In unserem Beispiel ist der Anweisungsteil für die Methode leer. Daher wird das Programm bei der Ausführung gestartet und ohne jede Aktion direkt wieder beendet. Es erscheint lediglich der Text

```
Drücken Sie eine beliebige Taste ...
```

auf dem Bildschirm, wenn Sie das Programm in der Entwicklungsumgebung ausführen lassen.

Neben den beiden Klammern `{` und `}` für den Anweisungsteil der Methode `Main()` finden Sie im vorigen Code aber noch ein weiteres Klammerpaar `{ } – nämlich für die Klasse Program. Dieses Klammerpaar fasst die Anweisungen der Klasse zusammen. Das bedeutet für unser Beispiel also: Die Methode Main() selbst gehört zur Klasse Program.`



Bitte beachten Sie:

Für jede öffnende Klammer `{` muss es auch eine schließende Klammer `}` geben. Eine schließende Klammer `}` bezieht sich dabei immer auf die öffnende Klammer `{`, die sich im Quelltext vor der schließenden Klammer befindet. Daher müssen auch unsere ersten Anwendungen immer mit mindestens zwei schließenden Klammern enden – die erste für die Methode `Main()` und die zweite für die Klasse. Mit den Regeln für die Klammerung werden wir uns ebenfalls noch ausführlicher beschäftigen.

Auch wenn das Programm aus dem vorigen Code eigentlich unsinnig ist, sollten Sie es einmal als Konsolenprogramm eingeben, kompilieren und ausführen lassen.

Sie werden sehen: Der Compiler akzeptiert das Programm, da es formal völlig korrekt ist. Dass das Programm sinnlos ist, wird vom Compiler nicht erkannt. Er überprüft den Quelltext also nur auf formale Fehler und nicht auf logische Fehler oder gar den Sinn.

Kommen wir wieder zurück zum Anweisungsteil für die Methode `Main()`. Normalerweise enthält er mehrere Anweisungen, die Sie durch ein Semikolon abschließen müssen. Zur besseren Übersicht sollten Sie außerdem für jede Anweisung eine neue Zeile beginnen.

Die grundsätzliche Struktur eines C#-Programms mit mehreren Anweisungen könnte dann so aussehen:

```
class Program
{
    static void Main(string[] args)
    {
        Anweisung1;
        Anweisung2;
    }
}
```

Code 1.2: Grundsätzliche Struktur eines C#-Programms mit mehreren Anweisungen

Zwischen den geschweiften Klammern für die Methode `Main()` finden Sie die Platzhalter für die Anweisungen – nämlich `Anweisung1` und `Anweisung2`. Die beiden Anweisungen werden jeweils durch ein Semikolon abgeschlossen. Wie Sie sehen, ist die grundsätzliche Struktur eines C#-Programms sehr einfach.

Bitte beachten Sie:

Die Zeilen `Anweisung1;` und `Anweisung2;` dienen im Code 1.2 nur als Platzhalter. Wenn Sie versuchen, dieses Programm übersetzen zu lassen, erhalten Sie Fehlermeldungen, da der Compiler mit dem Text `Anweisung1` und `Anweisung2` nichts anfangen kann. In den nächsten Kapiteln werden wir die Platzhalter durch „echte“ ausführbare Anweisungen ersetzen.



1.2 Kommentare

Auch wenn Ihnen der Aufbau eines C#-Programms jetzt vielleicht noch sehr einfach vorkommt, werden größere Programme recht schnell unübersichtlich, oder es kann sein, dass Sie im Laufe der Zeit einfach vergessen, was Sie sich bei Ihrem Quelltext gedacht haben und wie Sie die Methoden programmiert haben.

Daher sollten Sie Ihre Quelltexte immer angemessen kommentieren. Solche Kommentare dienen als Gedächtnisstütze und helfen erheblich beim Verständnis der Funktionsweise eines Programms.

Besonders, wenn Sie Ihre Programme an andere Programmierer weitergeben, sind Kommentare unverzichtbar. Andernfalls muten Sie den anderen Programmierern sehr viel zusätzliche und vor allem überflüssige Arbeit beim Hineindenken in Ihre Quelltexte zu.

Gewöhnen Sie sich daher an, Ihre Quelltexte **immer** zu kommentieren. Achten Sie dabei aber darauf, dass der Quelltext übersichtlich und lesbar bleibt. Einige Hinweise zur richtigen Kommentierung finden Sie am Ende dieses Kapitels in einem Exkurs.

Damit der Compiler einen Kommentar erkennt und beim Übersetzen ignoriert, muss er besonders gekennzeichnet werden. Dazu gibt es in C# zwei Möglichkeiten:

1. Das Kommentarzeichen `//`

Alles, was in der Zeile rechts von dem Kommentarzeichen steht, wird vom Compiler als Kommentar betrachtet und ignoriert. Das Kommentarzeichen `//` wird häufig eingesetzt, um einzelne Anweisungen zu kommentieren oder um einzelne Anweisungen zur Fehlersuche in einem Programm abzuschalten. Man spricht in diesem Fall auch von „auskommentieren“.

2. Die Kommentarzeichen `/*` und `*/`

Mit den Kommentarzeichen `/*` und `*/` können Sie ganze Blöcke im Quelltext als Kommentar markieren. Die Zeichen `/*` markieren dabei den Beginn eines Kommentars und die Zeichen `*/` das Ende des Kommentars. Alles, was zwischen diesen Zeichen steht, wird vom Compiler als Kommentar betrachtet. Diese Methode wird benutzt, um größere Kommentare zu erstellen oder bei der Fehlersuche in Programmen große Blöcke abzuschalten.

Nach diesen Regeln könnte also ein formal korrektes Programm mit Kommentaren so aussehen:

```
/* ##### Beschreibung des Programms #####
class Program
{
    static void Main(string[] args)
    {
    }
}

// #####
```

Code 1.3: C#-Programm mit Kommentaren

Wie Sie sehen, sind die ersten drei Zeilen des Programms durch die Zeichen `/*` und `*/` als Kommentar markiert. Danach folgen die Klasse und die Methode `Main()` mit einem leeren Anweisungsteil. Die letzte Zeile ist wieder als Kommentar markiert – diesmal allerdings mit dem Zeichen `//`.

Auch dieses Programm hat keine Wirkung, da der Anweisungsteil der Methode `Main()` leer ist. Sie können es aber trotzdem übersetzen und ausführen lassen.

Tipp:

Sie können Zeilen in einem Quelltext auch über das Symbol **Kommentiert die ausgewählten Textzeilen aus**  auskommentieren. Visual Studio setzt dann vor die aktuelle Zeile die Kommentarzeichen `//`. Mit dem Symbol **Hebt die Auskommierung der ausgewählten Textzeilen auf**  können Sie die Kommentarzeichen wieder entfernen.

Das funktioniert nicht nur für einzelne Zeilen, sondern auch für mehrere markierte Zeilen. Ziehen Sie dazu einfach die Maus mit gedrückter linker Taste über die gewünschten Zeilen und klicken Sie dann auf das entsprechende Symbol.

Exkurs: Regeln für die Kommentierung

Die Dokumentation des Quelltextes sollte den logischen Programmablauf darstellen, nicht jede Programmzeile einzeln erläutern.

Kommentierungen wie

```
//hier beginnt die Methode Main()  
static void Main(string[] args)
```

sind unsinnig, da sich die Anweisung selbst erklärt. Einem Laien wird auch die Kommentierung jeder einzelnen Zeile nicht helfen, das Programm zu verstehen. Für jemanden, der die Programmiersprache kennt, sind solche Kommentare nur überflüssiger Ballast.

Die Kommentierung einzelner Zeilen sollte sich daher auf ungewöhnliche oder sehr komplexe Befehle beschränken.

Eine zu ausführliche Kommentierung von Quelltext kann sogar zu Schwierigkeiten führen. Wird nämlich der Quelltext geändert, müssen natürlich auch die Kommentare angepasst werden. Je mehr Kommentare vorhanden sind, desto größer ist die Gefahr, dass ein Kommentar bei der Änderung vergessen wird. Und ein falscher Kommentar kann sehr viel schlimmere Auswirkungen haben als ein fehlender, da vorhandene Kommentare meistens als richtig angesehen werden.

Wichtiger ist die Kommentierung von größeren Einheiten in ihrer Wirkung – zum Beispiel

```
//Einlesen des nächsten Datensatzes und Umformen für Ausgabe
```

Denken Sie bei der Kommentierung auch an die Zeilenbreite. Nach Möglichkeit sollte ein Kommentar nicht länger als 65 Zeichen sein. Dann passt er auch beim Ausdruck auf eine DIN-A4-Seite noch ohne Umbruch in eine Zeile.

Zusammenfassung

Jedes C#-Programm besteht aus mindestens einer Klasse.

Methoden enthalten die Anweisungen, die ein C#-Programm ausführt. Die wichtigste Methode trägt den Namen `Main()`. Diese Methode bildet den Startpunkt für die Ausführung des Programms.

Zu jeder Methode gehört ein Anweisungsteil.

Sie können entweder einzelne Zeilen als Kommentar markieren oder ganze Blöcke.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Welche Methode muss in jedem C#-Programm enthalten sein? Warum?

- 1.2 Mit welchen Zeichen wird der Anweisungsteil einer Methode eingeschlossen? Mit welchem Zeichen werden einzelne Anweisungen im Anweisungsteil von einander getrennt?

- 1.3 Wie können Sie Kommentare in ein C#-Programm einfügen? Nennen Sie bitte beide Möglichkeiten.

2 Bildschirmausgabe

Nachdem wir uns im letzten Kapitel den grundsätzlichen Aufbau von C#-Programmen angesehen haben, wollen wir uns in diesem Kapitel mit der Bildschirmausgabe beschäftigen.

2.1 Bestandteile des „Hallo Welt“-Programms

Sie haben ja bereits einen ersten Kontakt mit einem Programm gehabt, das eine Ausgabe von Text auf dem Bildschirm durchführt. Dieses „Hallo Welt“-Programm sah so aus:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Hallo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hallo Welt");
            Console.WriteLine("Es grüßt Dich Max Meier");
        }
    }
}
```

Code 2.1: Das „Hallo Welt“-Programm

Schauen wir uns die einzelnen Anweisungen des Programms einmal genauer an:

Die Zeilen

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

machen **Namensräume** bekannt – in unserem Fall die Namensräume `System`, `System.Collections.Generic`, `System.Linq`, `System.Text` und `System.Threading.Tasks`. Diese Anweisungen sind unter anderem erforderlich, damit der Compiler die Anweisung `Console.WriteLine()` kennt. Denn eigentlich lautet die Anweisung vollständig `System.Console.WriteLine()`.

Hinweise:

Sie können die Bekanntgabe der Namensräume auch weglassen. Dann müssen Sie allerdings die Angaben direkt bei den Anweisungen machen. Wir benutzen die Anweisungen für die Namensräume daher in allen weiteren Beispielen, um Schreibarbeit zu sparen.

In der Vorlage für ein Konsolenprogramm werden immer zahlreiche Namensräume eingefügt, die nicht zwingend erforderlich sind. Diese überflüssigen Bekanntgaben können Sie durch die Livecodeanalyse sehr einfach entfernen lassen.

Wir drucken in den folgenden Codes immer nur die Namensräume ab, die auch wirklich benötigt werden.

Mit der Zeile

```
namespace Hallo
```

wird ein eigener Namensraum `Hallo` vereinbart. Die Anweisungen für den Namensraum werden dabei wie die Klasse und die Methode `Main()` durch die Klammern `{ }` umfasst. Daher enden die automatisch erstellten Quelltexte für Konsolenprogramme immer mit drei schließenden Klammern `}`. Die erste Klammer markiert das Ende des Anweisungsblocks der Methode `Main()`, die zweite das Ende der Klasse des Programms und die dritte schließlich das Ende des Namensraums.

Hinweise:

Der Namensraum hat in der Standardeinstellung immer denselben Namen wie das Projekt.

Sie können die Vereinbarung des Namensraums ebenfalls weglassen. Denken Sie dann aber bitte unbedingt daran, auch die dazugehörige öffnende und schließende Klammer zu löschen. Um Probleme zu vermeiden, sollten Sie aber gerade bei den ersten Projekten die Vereinbarung unverändert stehen lassen.

Die Zeile

```
class Program
```

vereinbart die Klasse `Program`. Danach folgt die öffnende Klammer `{` für den Anweisungsblock dieser Klasse.

Anschließend kommen der Kopf der Methode `Main()` und die öffnende Klammer `{` für den Anweisungsblock der Methode. Die Angaben in den Klammern bei der Methode `Main()` ermöglichen die Auswertung von Argumenten, die beim Aufruf des Programms übergeben werden können. Da wir davon in diesem Lehrgang aber keinen Gebrauch machen, wollen wir uns damit nicht weiter beschäftigen.

Danach folgen zwei Ausgabeanweisungen:

```
Console.WriteLine("Hallo Welt");
Console.WriteLine("Es grüßt Dich Max Meier");
```

Eingeleitet werden die Anweisungen durch die Angabe `Console`. Dadurch weiß der Compiler, dass Sie die Methode `WriteLine()` aus der Klasse `Console` benutzen möchten. Wenn Sie diese Angabe weglassen, erhalten Sie eine Fehlermeldung, dass `WriteLine()` nicht bekannt ist.

Bitte beachten Sie:

Eigentlich heißt die Klasse vollständig `System.Console`. Da Sie aber mit der Anweisung `using System;` den Namensraum `System` für das gesamte Programm nutzen können, reicht es aus, wenn Sie nur `Console.` angeben.

Was sich genau hinter Klassen, Methoden und Eigenschaften verbirgt, erfahren Sie detailliert im weiteren Verlauf.



Hinter der eigentlichen Anweisung folgt dann in Klammern die Ausgabe. In unserem Beispiel sollen die **Zeichenketten** „Hallo Welt“ und „Es grüßt Dich Max Meier“ auf dem Bildschirm ausgegeben werden. Damit der Compiler weiß, dass es sich um Zeichenketten handelt, müssen Sie den Text mit einem Anführungszeichen beginnen und auch abschließen. Wenn Sie eins dieser Anführungszeichen vergessen, erhalten Sie eine Fehlermeldung.

Zeichenketten werden im Fachjargon auch **Strings** genannt. Wörtlich übersetzt bedeutet String so viel wie „Schnur“ oder „Band“. Gemeint ist damit, dass die Zeichen hintereinanderstehen.



Die letzten drei Zeilen in unserem „Hallo Welt“-Programm schließlich sorgen dafür, dass die Anweisungsblöcke für die Methode `Main()`, die Klasse und den Namensraum korrekt geschlossen werden.

2.2 Textausgabe ohne und mit Zeilenumbruch

Neben der Anweisung `WriteLine()` können Sie einen Text auch mit der Anweisung `Write()` ausgeben. Dann wird allerdings nach der Ausgabe kein automatischer Zeilenumbruch durchgeführt – auch dann nicht, wenn Sie die Ausgabe auf mehrere Zeilen verteilen.

Schauen wir uns den Unterschied der beiden Anweisungen an einem Beispiel an:

```
/* ##### Textausgabe mit und ohne Zeilenumbruch #####
using System;
namespace Cshp02d_02_02
{
    class Program
    {
```

```

static void Main(string[] args)
{
    //bitte in einer Zeile eingeben
    Console.Write("Textausgabe in C#, ohne einen
    Zeilenumbruch");
    Console.Write("Textausgabe in C#,");
    Console.Write("ohne einen Zeilenumbruch");
    Console.WriteLine("Textausgabe in C#,");
    Console.WriteLine("mit Zeilenumbruch");
}
}

```

Code 2.2: Textausgabe mit und ohne Zeilenumbruch

Legen Sie bitte ein neues Projekt für eine Konsolenanwendung an und geben Sie den Quelltext aus dem vorigen Code ein. Speichern Sie alle Änderungen und lassen Sie das Programm dann ausführen. Die Ausgabe in der Eingabeaufforderung sollte ungefähr so aussehen:

```

Textausgabe in C#, ohne einen ZeilenumbruchTextausgabe in C#, ohne
einen ZeilenumbruchTextausgabe in C#,
mit Zeilenumbruch

Drücken Sie eine beliebige Taste . . .

```

Wie Sie sehen, werden die Ausgaben bei der Anweisung `Write()` direkt aneinandergehängt. Wo der Zeilenumbruch erfolgt, hängt von der Breite des Konsolenfensters ab. Erst die beiden letzten Ausgaben über `WriteLine()` werden wieder durch einen Zeilenumbruch getrennt. Dieser Umbruch erfolgt allerdings erst nach der Ausgabe. Daher steht die dritte Ausgabe von „Textausgabe in C#“ noch direkt hinter dem Text „Zeilenumbruch“.

2.3 Escape-Sequenzen

Sie können einen Zeilenumbruch auch mitten in der Ausgabe setzen. Dazu verwenden Sie die Escape¹-Sequenz `\n`.



Escape-Sequenzen sind spezielle Steuerzeichen. Sie werden durch den Backslash \ eingeleitet^{a)}.

a) Backslash bedeutet frei übersetzt so viel wie „Rückstrich“.

Im praktischen Einsatz finden Sie die Escape-Sequenz `\n` im folgenden Code:

1. Wörtlich übersetzt bedeutet *escape* so viel wie „Flucht“.

```
/* ##### Textausgabe mit Zeilenumbruch über \n #####
Textausgabe mit Zeilenumbruch über \n
##### */  
using System;  
  
namespace Cshp02d_02_03  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.Write("Textausgabe in C#, \nmit Zeilenumbruch\n");  
            Console.Write("Textausgabe\nin\nC#, \n");  
            Console.Write("mit\nZeilenumbruch\n\n");  
            Console.WriteLine("Textausgabe\nin C#, \n");  
            Console.WriteLine("\nmit Zeilenumbruch");  
        }  
    }  
}
```

Code 2.3: Textausgabe mit Zeilenumbrüchen über die Escape-Sequenz \n

Das Programm im vorigen Code entspricht weitgehend dem Code 2.2. Neu hinzugekommen ist lediglich die Escape-Sequenz \n in den Zeichenketten.

Die Ausgabe in der Eingabeaufforderung sieht dann so aus:

```
Textausgabe in C#,  
mit Zeilenumbruch  
Textausgabe  
in  
C#,  
mit  
Zeilenumbruch
```

```
Textausgabe  
in C#,
```

```
mit Zeilenumbruch
```

Hinweis:

Die automatisch erzeugte Meldung Drücken Sie eine beliebige Taste ... drucken wir bei allen weiteren Ausgaben aus Platzgründen nicht immer wieder ab.

Wenn Sie einmal die Ausgabe der einzelnen Zeichenketten vergleichen, werden Sie feststellen, dass überall dort, wo die Escape-Sequenz \n steht, ein Zeilenumbruch durchgeführt wurde. Dabei spielt es keine Rolle, ob sich die Escape-Sequenz zu Beginn der

Zeichenkette, mitten in der Zeichenkette oder am Ende der Zeichenkette befindet. Sie können auch mehrere Zeilenumbrüche direkt hintereinander ausgeben lassen. Dann erscheinen in der Eingabeaufforderung entsprechend leere Zeilen.

Neben der Escape-Sequenz `\n` für einen Zeilenumbruch gibt es auch noch eine Reihe weiterer Escape-Sequenzen, mit denen Sie die Ausgabe beeinflussen können. Eine Übersicht finden Sie in der folgenden Tabelle:

Tab. 2.1: Steuerzeichen mit Escape-Sequenzen

Escape-Sequenz	Wirkung
<code>\n</code>	Es wird ein Zeilenumbruch eingefügt.
<code>\t</code>	Die Einfügemarke wird an den nächsten horizontalen Tabulator gesetzt. Sie bewegt sich damit nach rechts.
<code>\b</code>	Die Einfügemarke wird ein Zeichen nach links bewegt.
<code>\r</code>	Die Einfügemarke wird an den Beginn der aktuellen Zeile gesetzt. Es erfolgt kein Zeilenumbruch.
<code>\a</code>	Es wird ein Signalton ausgegeben.

Probieren Sie die Wirkung der verschiedenen Escape-Sequenzen am besten direkt in einem Programm aus. Sie können zum Beispiel den vorigen Code ein wenig überarbeiten. Ersetzen Sie die Escape-Sequenz `\n` durch andere Escape-Sequenzen aus der Tabelle und fügen Sie weitere Escape-Sequenzen mitten in die Texte ein.

Escape-Sequenzen werden auch verwendet, um bestimmte Zeichen, die für den Compiler eine besondere Bedeutung haben, als „normale“ Zeichen auszugeben.

Schauen wir uns das an einem Beispiel an. Wie Sie ja bereits wissen, werden die Anführungszeichen in C# verwendet, um eine Zeichenkette einzuleiten und zu beenden. Wenn Sie nun aber selbst ein Anführungszeichen auf dem Bildschirm ausgeben wollen, müssen Sie dem Compiler mitteilen, dass das Anführungszeichen nicht als Anfangs- beziehungsweise Endemarkierung für eine Zeichenkette steht, sondern als „normales“ Zeichen behandelt werden soll.

Nehmen wir einmal an, Sie wollen die Zeichenkette Anfang "Mitte" Ende auf dem Bildschirm ausgeben. Das Wort „Mitte“ soll dabei durch zwei Anführungszeichen eingeschlossen sein. Die folgende Ausgabeanweisung sieht auf den ersten Blick ganz viel versprechend aus:

```
Console.WriteLine("Anfang \"Mitte\" Ende");
```

Wenn Sie sie aber so eingeben, werden Sie schnell feststellen, dass der Compiler damit nichts anfangen kann. Denn das Wort „Mitte“ wird in Visual Studio nicht dunkelrot dargestellt, sondern ganz normal und auch mit einer roten Wellenlinie markiert. Das heißt, der Editor erkennt gar nicht, dass „Mitte“ zu der Zeichenkette gehört. Und prompt gibt es beim Übersetzen auch mehrere Fehlermeldungen.

Diese Meldungen erscheinen, weil die Zeichenkette zwar korrekt vor `Anfang` eingeleitet wird, aber durch die Anführungszeichen vor dem Wort `Mitte` für den Compiler wieder zu Ende ist. Das Wort `Ende` wird dann als zweite eigene Zeichenkette interpretiert.

Damit auch die mittleren beiden Anführungszeichen wie gewünscht als Zeichen ausgegeben werden, müssen Sie sie als Escape-Sequenz schreiben. Die korrekte Schreibweise für unser Beispiel sieht dann so aus:

```
Console.WriteLine("Anfang \"Mitte\" Ende");
```

Durch den einleitenden Backslash `\` erkennt der Compiler jetzt, dass die beiden Anführungszeichen als Zeichen ausgegeben werden sollen. Die Zeichenkette ist damit erst beim letzten Anführungszeichen hinter dem Wort `Ende` zu Ende.

Weitere Escape-Sequenzen für die Ausgabe von besonderen Zeichen finden Sie in der folgenden Tabelle:

Tab. 2.2: Escape-Sequenzen für besondere Zeichen

auszugebendes Zeichen	Escape-Sequenz
<code>"</code>	<code>\ "</code>
<code>'</code>	<code>\ '</code>
<code>\</code>	<code>\ \</code>

So viel an dieser Stelle zur Ausgabe von Texten.

Zusammenfassung

Die Ausgabe von Text in der Eingabeaufforderung erfolgt über die Anweisungen `Console.WriteLine()` und `Console.Write()`. Die Anweisung `Console.WriteLine()` führt nach der Ausgabe automatisch einen Zeilenumbruch durch.

Zeichenketten müssen mit Anführungszeichen eingeschlossen werden.

Über Escape-Sequenzen können Sie die Ausgabe auf dem Bildschirm steuern und bestimmte Zeichen, die in C# eine besondere Bedeutung haben, ausgeben. Zu diesen Zeichen gehört zum Beispiel das Anführungszeichen.

Aufgaben zur Selbstüberprüfung

- 2.1 Wie lauten die Escape-Sequenzen für
a) einen Zeilenumbruch?

Digitized by srujanika@gmail.com

- b) einen horizontalen Tabulator?

Digitized by srujanika@gmail.com

- c) einen Backslash (\)?

reiben Sie ein C#-Programm, das mithilfe von Escape-Sequenzen folgende Ausgabe erzeugt:

```
Dies ist ein C#-Programm,  
das mithilfe von Escape-Sequenzen sogar Zeichen  
wie " " ", " \ " und " ' ",  
die in C# eine besondere Bedeutung haben, ausgibt.
```

Verwenden Sie für die Ausgabe bitte nur die Anweisung `Console.WriteLine()`. Achten Sie besonders auf die Anführungszeichen und die Zeilenumbrüche.

3 Operatoren, Variablen und symbolische Konstanten

Sie wissen jetzt, wie Sie Texte auf dem Bildschirm ausgeben lassen können. In diesem Kapitel beschäftigen wir uns mit der Verarbeitung von Zahlen. Sie lernen, welche arithmetischen Operatoren C# unterstützt und wie Sie Werte in Variablen speichern können. Außerdem erfahren Sie etwas zu symbolischen Konstanten.

Beginnen wir mit den arithmetischen Operatoren.

3.1 Arithmetische Operatoren

Die **Arithmetik** ist ein Teilgebiet der Mathematik, das sich mit Zahlen und den für sie geltenden Rechenregeln befasst.



Arithmetische Operationen wie die Addition und die Subtraktion werden durch arithmetische Operatoren gekennzeichnet. Diese Operatoren sind Zeichen, die eine bestimmte Operation auslösen – zum Beispiel eben eine Rechenoperation. Zu den arithmetischen Operatoren gehören unter anderem das Zeichen + für Addition und das Zeichen – für Subtraktion.

C# unterstützt fünf wichtige arithmetische Operatoren. Sie finden diese Operatoren und ihre Bedeutung in der folgenden Tabelle:

Tab. 3.1: Arithmetische Operatoren in C#

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
% ^{a)}	Modulo

a) Bei dem Zeichen % handelt es sich um das Prozentzeichen.

Die Operatoren für die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division sollten Sie noch aus Ihrer Schulzeit kennen. Neu ist daher wahrscheinlich nur der Operator %. Er liefert den Rest einer Division.

Beispiel 3.1:



Der Ausdruck 9 % 6 liefert das Ergebnis 3 – den Rest der Division 9/6 (9/6 = 1 Rest 3).

Bei der Verarbeitung der Operatoren gibt es eine feste Rangfolge – die **Priorität**. Sie bestimmt, in welcher Reihenfolge die Operatoren verarbeitet werden, wenn sie gemeinsam in einem Ausdruck verwendet werden. Eine Rangfolge mit einigen wichtigen Operatoren finden Sie in der folgenden Tabelle. Die oberen Operatoren werden dabei zuerst ausgewertet – sie **binden** stärker an ihren Operanden.

Tab. 3.2: Rangfolge der C#-Operatoren

Operatoren	Auswertungsreihenfolge
. () []	von links nach rechts
! ~ + - ++ --	von rechts nach links
* / %	von links nach rechts
+ -	von links nach rechts
< <= > >=	von links nach rechts
== !=	von links nach rechts
&&	von links nach rechts
	von links nach rechts
= *= /= %= += -=	von rechts nach links

Hinweise:

Die Tabelle ist nicht vollständig.

Die meisten Operatoren der Tabelle behandeln wir im weiteren Verlauf. Einen vollständigen Überblick finden Sie in der Hilfe im Dokument **C#-Operatoren**.

+ und – in der zweiten Zeile meinen die Vorzeichen. Die arithmetischen Operatoren für Addition und Subtraktion finden Sie in der vierten Zeile der Tabelle.

Sie müssen diese Tabelle nun keineswegs auswendig lernen. Sie sollten vor allem folgende Grundregeln berücksichtigen:

1. Punktrechnung geht vor Strichrechnung

Die Modulo-Operation zählt dabei wie die Multiplikation und die Division zu den Punktrechnungen. Addition und Subtraktion sind Strichrechnungen.

2. Bei gleicher Priorität werden die Operatoren in der Regel von links nach rechts abgearbeitet.

Wenn also Addition und Subtraktion (zwei Strichrechnungen) direkt aufeinanderfolgen, wird zuerst addiert und dann subtrahiert. Ausnahmen von dieser Regel sind in der vorigen Tabelle aufgeführt.

3. Durch Klammern können Sie die Reihenfolge der Auswertung verändern.

Dazu sehen wir uns gleich ein Beispiel an.

Die Priorität der Operatoren können Sie auch grob an der Anzahl der Operanden festmachen, die ein Operator benötigt. Die höchste Priorität haben in der Regel die **unären Operatoren** – die Operatoren mit einem Operanden. Dann folgen die **binären Operatoren**, die mit zwei Operanden arbeiten, und anschließend der **ternäre Operator** mit drei Operanden.

Bitte beachten Sie aber, dass diese Zuordnung nur erste Hinweise gibt. Im Zweifelsfall sollten Sie daher immer in der vorigen Tabelle nachsehen.

Schauen wir uns die arithmetischen Operationen nun an einem praktischen Beispiel an. Der folgende Code führt einige Rechenoperationen aus und gibt das Ergebnis auf dem Bildschirm aus.

```
/* ##### Arithmetische Operationen ##### */
using System;
namespace Cshp02d_03_01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("5 + 9\t\t= ");
            Console.WriteLine(5 + 9);

            Console.Write("5 - 9\t\t= ");
            Console.WriteLine(5 - 9);

            Console.Write("5 * 9\t\t= ");
            Console.WriteLine(5 * 9);

            Console.Write("9 / 5\t\t= ");
            Console.WriteLine(9 / 5);

            Console.Write(" Rest ");
            Console.WriteLine(9 % 5);

            Console.Write("10 + 8 / 2\t= ");
            Console.WriteLine(10 + 8 / 2);

            //Geänderte Reihenfolge durch Klammern
            Console.Write("(10 + 8) / 2\t= ");
            Console.WriteLine((10 + 8) / 2);
        }
    }
}
```

Code 3.1: Arithmetische Operationen in C#

Die Ausgabe auf dem Bildschirm sieht so aus:

5 + 9	= 14
5 - 9	= -4
5 * 9	= 45
9 / 5	= 1 Rest 4
10 + 8 / 2	= 14
(10 + 8) / 2	= 9

Wenn Sie sich die Ausgaben für die Division genau ansehen, werden Sie feststellen, dass das Ergebnis in der Anweisung

```
Console.WriteLine(9 / 5);
```

nicht ganz richtig ist. Statt 1,8 wird hier nur 1 ausgegeben und direkt dahinter das Ergebnis der Modulo-Operation. Dieser „Fehler“ liegt daran, dass C# intern ohne weitere Angaben immer mit ganzen Zahlen arbeitet und eventuelle Nachkommastellen schlicht und einfach unterschlägt. Wie Sie dieses Problem beheben können, erfahren Sie im weiteren Verlauf dieses Studienheftes.

An den letzten beiden Rechnungen und den dazugehörigen Ausgaben

```
Console.WriteLine(10 + 8 / 2); 10 + 8 / 2 = 14
Console.WriteLine((10 + 8) / 2); (10 + 8) / 2 = 9
```

sehen Sie, wie Sie die Reihenfolge bei der Auswertung der Operatoren durch Klammern beeinflussen können.

In der ersten Zeile wird zunächst 8 durch 2 dividiert und dann das Ergebnis mit 10 addiert. Hier gilt die Regel „Punktrechnung vor Strichrechnung“.

In der zweiten Zeile dagegen wird der Ausdruck 10 + 8 geklammert. Er hat damit höhere Priorität als die Division. Hier wird also erst addiert und anschließend dividiert.

Der vorige Code mit den Beispielen der arithmetischen Operatoren ist bereits etwas unübersichtlich, da wir die Ausgaben jeweils über eigene Anweisungen ausführen lassen. Sie können die Ausgabe der Texte und der Ergebnisse aber auch direkt hintereinander stellen. Dazu hängen Sie die Ausgabe der Ergebnisse durch ein Komma getrennt an die Ausgabe der Texte an. Damit der Compiler weiß, wo er die Ergebnisse in die Zeichenkette schreiben soll, geben Sie außerdem noch einen Platzhalter an. Dieser Platzhalter besteht aus einer laufenden Nummer, die durch geschweifte Klammern umfasst wird.

Die erste Ausgabe sieht dann so aus:

```
Console.WriteLine("5 + 9\t\t= {0}", 5 + 9);
```

Der Platzhalter {0} hinter dem Gleichheitszeichen = zeigt dem Compiler, dass er an dieser Stelle den Wert des Ausdrucks 5 + 9, der hinter dem Komma steht, ausgeben soll.

Bitte beachten Sie unbedingt:

Die Nummerierung in den Platzhaltern beginnt bei 0 und nicht bei 1. Die Anweisung

```
Console.WriteLine("5 + 9\t\t= {1}", 5 + 9);
```

führt zu Problemen bei der Ausführung des Programms. Hier sprechen Sie nämlich mit dem Platzhalter `{1}` einen **zweiten** Ausdruck an, den es aber gar nicht gibt.



Der vollständige Code mit den aneinandergehängten Ausgaben sieht so aus:

```
/* ##### Arithmetische Operationen mit verkürztem Quelltext #####
using System;
namespace Cshp02d_03_02
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("5 + 9\t\t= {0}", 5 + 9);
            Console.WriteLine("5 - 9\t\t= {0}", 5 - 9);
            Console.WriteLine("5 * 9\t\t= {0}", 5 * 9);
            //Ausgabe mit zwei Platzhaltern
            Console.WriteLine("9 / 5\t\t= {0} Rest {1}", 9 / 5, 9 % 5);
            Console.WriteLine("10 + 8 / 2\t\t= {0}", 10 + 8 / 2);
            //Geänderte Reihenfolge durch Klammern
            Console.WriteLine("(10 + 8) / 2\t\t= {0}", (10 + 8) / 2);
        }
    }
}
```

Code 3.2: Verkürzter Quelltext für Code 3.1

In der Anweisung

```
Console.WriteLine("9 / 5\t\t= {0} Rest {1}", 9 / 5,
9 % 5);
```

sehen Sie auch zwei Platzhalter im Einsatz. Über den Platzhalter `{0}` wird das Ergebnis des Ausdrucks $9 / 5$ ausgegeben und über den Platzhalter `{1}` das Ergebnis des Ausdrucks $9 \% 5$. Die Reihenfolge der Platzhalter muss dabei auch nicht unbedingt mit der Reihenfolge der Ausdrücke übereinstimmen. Möglich wäre zum Beispiel auch die folgende Anweisung

```
Console.WriteLine("9 / 5 ergibt als Rest {1} und als
Ergebnis {0}", 9 / 5, 9 % 5);
```

Hier wird zuerst das Ergebnis des zweiten Ausdrucks ausgegeben und dann das Ergebnis des ersten Ausdrucks. Allerdings sind solche Konstruktionen vor allem beim flüchtigen Hinsehen recht verwirrend.

Sie können einen Platzhalter auch mehrfach verwenden. Dann wird jedes Mal der Wert des dazugehörigen Ausdrucks ausgegeben – so wie in der folgenden Anweisung:

```
Console.WriteLine("5 + 4 ist {0} und 4 + 5 ist {0}", 5 + 4);
```

Sie können die Berechnungen auch direkt in den Klammern angeben – also auf die Platzhalter komplett verzichten. Dann müssen Sie allerdings vor der gesamten Zeichenkette das Zeichen \$ angeben. Die Anweisung

```
Console.WriteLine("$5 + 9\t\t= {0}", 5 + 9);
```

könnte also auch so aussehen:

```
Console.WriteLine($"5 + 9\t\t= {5 + 9});
```

Welche Form Sie benutzen, ist im Wesentlichen eine Frage des persönlichen Geschmacks. Die Ergebnisse sind identisch.



Zeichenketten, die durch das Zeichen \$ gekennzeichnet sind, werden auch interpolierte Zeichenketten genannt.

So viel an dieser Stelle zu den Operatoren und den Platzhaltern. Einige weitere Operatoren und auch zusätzliche Möglichkeiten zur Formatierung der Ausgabe werden Sie im weiteren Verlauf dieses Studienheftes kennenlernen. Jetzt beschäftigen wir uns aber zunächst einmal mit den Variablen.

3.2 Variablen

Bei den Programmen aus den beiden vorigen Codes werden die Ergebnisse der Operationen immer direkt ausgegeben und können nicht zwischengespeichert werden. Sehr viel interessanter als solche Programme, die nur feste Werte verwenden, sind natürlich Programme, die mit beliebigen Werten arbeiten können.

Dazu benötigen wir zunächst einmal überhaupt eine Möglichkeit, Werte in einem Programm zwischenspeichern zu können. Diese Aufgabe übernehmen **Variablen**.



Der Inhalt einer Variablen kann – wie der Name schon sagt – nahezu beliebig verändert werden.

Damit Sie leichter nachvollziehen können, was Variablen sind, sehen wir uns ein Beispiel aus der Welt außerhalb des Computers an.

Nehmen wir einmal an, Sie haben einen Schrank mit Schubladen verschiedener Größen und einige Kleidungsstücke unterschiedlicher Art. In jede Schublade passt genau ein Kleidungsstück.

Der Schrank steht dabei für den **Arbeitsspeicher** eines Computers und die Kleidungsstücke für die **Daten**, die Sie im Arbeitsspeicher ablegen möchten. Die Schubladen entsprechen dann unseren **Variablen** – also einem **Speicherplatz**, in dem Sie etwas ablegen können. So wie die Schubladen unterschiedliche Größen haben können, haben auch Variablen unterschiedliche Größen beziehungsweise unterschiedliche Typen. Es gibt zum Beispiel ganze Zahlen und Zahlen mit Kommastellen, die – wie Sie später sehen werden – unterschiedlich viel Speicherplatz benötigen.

Wenn Sie einer Variablen einen Wert **zuweisen**, so bedeutet das in unserem Beispiel, dass Sie ein Kleidungsstück in eine Schublade legen. Das Kleidungsstück in einer Schublade – also den Inhalt – nennt man den **Wert** einer Variablen.

Da die Kleidungsstücke unterschiedliche Größen haben, passen sie nur in bestimmte Schubladen. Auf Variablen übertragen heißt das: Sie können einer Variablen nur solche Werte zuweisen, die auch zum Typ der Variablen passen – dem **Datentyp**. Eine Zahl mit Nachkommastellen passt zum Beispiel nicht in eine Variable, die für ganze Zahlen erstellt wurde.

Damit Sie wissen, was in den einzelnen Schubladen liegt, kleben Sie ein Schildchen mit einem Namen auf die Schubladen. Bei Variablen nennt man diesen Namen den **Bezeichner**.

Der Datentyp einer Variablen legt fest, welche Daten Sie in der Variablen ablegen können. Der Bezeichner einer Variablen dient zur eindeutigen Identifikation.



Schauen wir uns jetzt einmal den Einsatz von Variablen in einem C#-Programm an. Wir zeigen Ihnen dabei zunächst einmal den vollständigen Quelltext und sehen uns dann die einzelnen Zeilen der Reihe nach an.

```
/* ##### Variablen Einführung ##### */
using System;
namespace Cshp02d_03_03
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung einer Variable zahl für ganze Zahlen
            int zahl;
            //1. Zuweisung
            zahl = 10;
            //1. Ausgabe
            Console.WriteLine("zahl hat den Wert {0}", zahl);
            //2. Ausgabe
            Console.WriteLine("zahl hat den Wert {0}", zahl);
```

```
//2. Zuweisung
zahl = 22;
Console.WriteLine("zahl hat jetzt den Wert {0}", zahl);
}
```

Code 3.3: Beispiel für die Verwendung von Variablen

Wenn Sie das Programm ausführen lassen, sollten folgende Ausgaben auf dem Bildschirm erscheinen:

```
zahl hat den Wert 10
zahl hat den Wert 10
zahl hat jetzt den Wert 22
```

Schauen wir uns an, wie diese Ausgaben zustande kommen:

In der Zeile

```
int zahl;
```

wird die Variable dem Compiler bekannt gemacht – sie wird **vereinbart**.



Die Vereinbarung einer Variablen umfasst den **Datentyp** und den **Bezeichner** der Variablen. Abgeschlossen wird die Vereinbarung durch ein Semikolon.

Als **Datentyp** verwenden wir in unserem Beispiel `int`. Diese Angabe steht als Abkürzung für *Integer*² und bedeutet, dass die Variable eine positive oder negative ganze Zahl in einem bestimmten Wertebereich aufnehmen kann. Mit den Datentypen beschäftigen wir uns im nächsten Kapitel noch ausführlich.

Als **Bezeichner** verwenden wir `zahl`. Über diesen Namen greifen wir dann in den Anweisungen auf den Wert der Variablen zu.

Hinweise:

Die Vereinbarung einer Variablen kann an nahezu beliebiger Stelle im Quelltext erfolgen. Sie müssen lediglich darauf achten, dass Sie die Variable vor dem ersten eigentlichen Gebrauch vereinbaren. Es hat sich aber in C# eingebürgert, dass Variablen wie im Beispiel zu Beginn einer Methode vereinbart werden. Dann ist auf einen Blick klar, welche verschiedenen Variablen in der Methode genutzt werden.

Die Bezeichner von lokalen Variablen beginnen nach den C#-Konventionen mit einem Kleinbuchstaben. Was genau lokale Variablen sind, erfahren Sie später.

Schauen wir uns nun die erste Anweisung des Programms an:

```
zahl = 10;
```

Hier wird der Wert 10 in die Variable `zahl` geschrieben. Dazu wird der **Zuweisungsoperator** `=` benutzt. Er weist der links stehenden Variablen das Ergebnis des rechts stehenden Ausdrucks zu. Nach der Zuweisung hat die Variable `zahl` also den Wert 10.

2. *Integer* bedeutet übersetzt „ganze Zahl“.

Danach folgen zwei Ausgaben.

```
//1. Ausgabe  
Console.WriteLine("zahl hat den Wert {0}", zahl);  
//2. Ausgabe  
Console.WriteLine("zahl hat den Wert {0}", zahl);
```

Dabei wird der aktuelle Wert der Variablen `zahl` ausgelesen und auf dem Bildschirm ausgegeben. Bitte beachten Sie, dass sich durch das Auslesen der Wert der Variablen nicht verändert. Daher ist die zweite Ausgabe identisch mit der ersten.

Statt des Platzhalters können Sie auch bei der Ausgabe einer Variablen mit einer interpolierten Zeichenkette arbeiten. Die Anweisung sieht dann so aus:

```
Console.WriteLine($"zahl hat den Wert {zahl}");
```

Hinweis:

Bei der Ausgabe über `Console.WriteLine()` und `Console.Write()` ist der Datentyp der Variablen im Allgemeinen ohne Bedeutung. Es wird automatisch das richtige Format für die Ausgabe gewählt.

Nach den beiden Ausgaben weisen wir dann der Variablen `zahl` den Wert 22 zu. Das übernimmt die folgende Anweisung:

```
zahl = 22;
```

Nach der Zuweisung hat die Variable jetzt nicht mehr den Wert 10, sondern den Wert 22. Das heißt also, ein bereits vorhandener Wert wird bei einer neuen Zuweisung überschrieben und geht damit verloren. Das erkennen Sie auch an der letzten Ausgabe.

```
Console.WriteLine("zahl hat jetzt den Wert {0}", zahl);
```

Diese Ausgabe liefert jetzt nämlich den neuen Wert von `zahl` – also 22.

3.3 Namenskonventionen für Bezeichner

Die Wahl von Bezeichnern für Variablen – also die Wahl von Namen – unterliegt in C# einigen Einschränkungen. Diese Einschränkungen gelten auch für die meisten anderen Bezeichner – zum Beispiel für symbolische Konstanten.

1. Schlüsselwörter dürfen nicht einzeln als Bezeichner verwendet werden.

Bei den Schlüsselwörtern handelt es sich um reservierte Wörter, die vom Compiler selbst verwendet werden – zum Beispiel `using` oder `namespace`. Eine Variable können Sie also nicht `using` nennen.

Sie können allerdings Schlüsselwörter als Teil eines Bezeichners verwenden. Der Bezeichner `usingVar` wäre dann zum Beispiel erlaubt.

2. Das erste Zeichen eines Bezeichners **muss** entweder ein Unterstrich `_` oder ein Buchstabe sein.

Alle anderen Zeichen und auch Ziffern sind als erstes Zeichen nicht erlaubt und führen zu einer Fehlermeldung des Compilers.

3. Die weiteren Zeichen des Namens **müssen** Buchstaben, Ziffern oder Unterstriche sein.

Zeichen wie . oder / dürfen Sie in einem Bezeichner nicht verwenden. Auch Leerzeichen sind nicht möglich.

4. C# unterscheidet bei den Bezeichnern zwischen **Groß- und Kleinschreibung**.

Die Bezeichner Zahl und zahl stehen also für zwei unterschiedliche Variablen.

Neben diesen Einschränkungen, die vom Compiler vorgegeben werden, sollten Sie bei der Bezeichnung von Variablen noch einige Stilrichtlinien beachten. Diese Richtlinien stellen sicher, dass Ihr Quelltext gut lesbar wird und damit auch für Dritte nachvollziehbar ist.

1. Verwenden Sie „sprechende“ Bezeichner, die bereits erste Hinweise auf den Zweck der Variablen geben.

So sind Namen wie a, b oder c nicht sonderlich geeignet, weil sie keinen Hinweis auf den Zweck der Variablen geben. Besser wären zum Beispiel summe, rest oder ergebnis.

2. Wählen Sie kurze prägnante Namen.

Zwar ist die Länge der Bezeichner in C# nicht beschränkt, aber lange komplizierte Namen führen schnell zu Fehlern und machen den Quelltext schwer lesbar. Denken Sie daran: Sie müssen immer den vollständigen Namen der Variablen eingeben, wenn Sie mit der Variablen arbeiten wollen.

Zu kurz sollten Sie den Namen aber auch nicht wählen, da er andernfalls nicht mehr verständlich ist.

3. Wenn Sie zusammengesetzte Namen verwenden, beginnen Sie jeden Bestandteil des Namens mit einem Großbuchstaben.

So lässt sich ergebnisAddition sehr viel leichter lesen als ergebnisaddition oder auch ergebnis_addition.

In der folgenden Tabelle finden Sie die Regeln an einigen Beispielen noch einmal zusammengefasst:

Tab. 3.3: Beispiele für Bezeichner

Bezeichner	Hinweis
using	Der Bezeichner ist nicht gültig, da es sich um ein Schlüsselwort handelt.
usingVar	Der Bezeichner ist gültig, da das Schlüsselwort in Kombination verwendet wird.
1Zahl	Der Bezeichner ist ungültig, da das erste Zeichen eine Zahl ist.
1 . Zahl	Der Bezeichner ist ungültig. Der Punkt ist nicht zulässig. Außerdem beginnt der Bezeichner mit einer Zahl.

Bezeichner	Hinweis
Dies_ist_die_erste_Zahl	Der Bezeichner ist grundsätzlich gültig, aber unnötig lang. Außerdem ist er durch die Unterstriche recht schwer zu lesen.
diesIstDieErsteZahl	Der Bezeichner ist grundsätzlich gültig, aber auch ebenfalls unnötig lang.
z	Der Bezeichner ist gültig, aber nicht sprechend.
Zahl, zahl	Beide Bezeichner sind gültig. Es handelt sich aber um zwei unterschiedliche Variablen.

3.4 Ein komplexeres Beispiel

Schauen wir uns jetzt ein etwas komplexeres Beispiel für den Einsatz von Variablen an. Wir verwenden in einem Programm vier Variablen vom Datentyp `int` und weisen ihnen unterschiedliche Werte zu. Dabei benutzen wir zum einen eine Zahl, zum anderen aber auch arithmetische Operationen. In einer der arithmetischen Operationen beziehen wir uns auf den Wert einer der Variablen.

Abschließend sollen die Werte von zwei Variablen getauscht werden, ohne dass die ursprünglichen Werte dabei verloren gehen. Dieses Verfahren nennt man **Ringtausch**.

Sehen wir uns zuerst wieder den vollständigen Quelltext an:

```
/* ##### Ein komplexeres Beispiel für Variablen #####
using System;
namespace Cshp02d_03_04
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            int zahl1 = 1;
            int zahl2, zahl3, zahl4;
            //Zuweisungen
            //zahl2 erhält das Ergebnis des Ausdrucks 1+1, also 2
            zahl2 = 1 + 1;
            //zahl3 erhält den aktuellen Wert von Zahl2 * 5, also
            //10
            zahl3 = zahl2 * 5;

            //Ausgaben
            Console.WriteLine("zahl1 hat den Wert {0}", zahl1);
            Console.WriteLine("zahl2 hat den Wert {0}", zahl2);
            Console.WriteLine("zahl3 hat den Wert {0}", zahl3);
```

```
//Hier beginnt der Ringtausch
//Die Werte von zahl1 und zahl2 werden getauscht
//Zuerst wird der Wert von zahl1 in zahl4 gesichert
zahl4 = zahl1;
//dann erhält zahl1 den Wert von zahl2
zahl1 = zahl2;
//und zum Schluss wird zahl2 der gesicherte
//Wert von zahl1 aus zahl4 zugewiesen
zahl2 = zahl4;

//Ausgaben nach dem Tausch
Console.WriteLine("zahl1 hat jetzt den Wert {0}", zahl1);
Console.WriteLine("zahl2 hat jetzt den Wert {0}", zahl2);
}
}
```

Code 3.4: Komplexeres Beispiel für die Arbeit mit Variablen

Nehmen wir wieder die einzelnen Zeilen unter die Lupe:

Bei der Vereinbarung

```
int zahl1 = 1;
```

erhält die Variable `zahl1` direkt einen Wert zugewiesen. Damit sparen wir uns die separate Zuweisung in einer eigenen Anweisung.

Anschließend erfolgt die Vereinbarung von drei Variablen gleichzeitig:

```
int zahl2, zahl3, zahl4;
```

Hier werden mehrere Bezeichner – getrennt durch Kommas – hinter dem Datentyp angegeben. Das ist eine einfache Methode, um mehrere Variablen desselben Typs in einem Rutsch zu vereinbaren. Wir erzeugen hier also drei Variablen vom Typ `int`.

In den folgenden zwei Anweisungen setzen wir den Zuweisungsoperator `=` ein.

```
zahl12 = 1 + 1;  
zahl13 = zahl12 * 5;
```

In der ersten Zuweisung wird der Variablen `zahl12` das Ergebnis des Ausdrucks `1 + 1` zugewiesen. Das heißt, wenn Sie hinter dem Zuweisungsoperator einen Ausdruck angeben, wird erst der Ausdruck ausgewertet und dann das Ergebnis zugewiesen. In unserem Beispiel erhält die Variable `zahl12` also den Wert `2`.

In der nächsten Anweisung benutzen wir den aktuellen Wert einer Variablen in einem Ausdruck. Dazu müssen Sie lediglich den Bezeichner der Variablen angeben, deren Wert Sie verwenden möchten. Beim Ausführen des Programms ersetzt der Compiler dann automatisch den Bezeichner durch den aktuellen Wert der Variablen. In unserem Beispiel erhält `zahl3` den Wert 10 – also das Ergebnis der Rechnung `2 * 5` (dem aktuellen Wert von `zahl2`).

In den folgenden Anweisungen werden dann die Werte der Variablen auf dem Bildschirm ausgegeben. Dabei gibt es keine Besonderheiten.

Kommen wir nun zum Ringtausch

Beim Vertauschen der Werte von zwei Variablen ergibt sich folgendes Problem: Wenn Sie den Wert einer Variablen direkt mit dem Wert einer anderen Variablen überschreiben, ist der Wert der ersten Variablen verloren. Sie müssen daher den Wert der ersten Variablen zunächst in einer weiteren Variablen zwischenspeichern und nach der Zuweisung dann wieder auslesen. Diese Zuweisungen und das Zwischenspeichern erledigen die Anweisungen:

```
zahl4 = zahl1;  
zahl1 = zahl2;  
zahl2 = zahl4;
```

Zuerst wird der Wert von `zahl1` in der Variablen `zahl4` zwischengespeichert. Dazu geben Sie einfach den Bezeichner der Variablen hinter dem Zuweisungsoperator an.

Danach können Sie den Wert der Variablen `zahl2` in der Variablen `zahl1` ablegen und anschließend den vorher gesicherten Wert von `zahl4` in die Variable `zahl2` schreiben. Nach dem Ausführen dieser drei Anweisungen haben Sie die Werte der Variablen `zahl1` und `zahl2` getauscht.

Die Ausgabeanweisungen, die auf den Ringtausch folgen, geben zur Kontrolle das Ergebnis der gesamten Operation aus.

Experimentieren Sie ruhig auch mit dem vorigen Code ein wenig. Sie können ja einmal versuchen, allen vier Variablen feste Werte zuzuweisen und dann die Werte der Variablen `zahl3` und `zahl4` zu vertauschen. Denken Sie aber vor Ihren Experimenten daran, den Quelltext zu sichern.

Bitte beachten Sie:

Alle Variablen, mit denen wir bisher gearbeitet haben, sind **lokale Variablen**. Sie erhalten allein durch die Vereinbarung keinen Standardwert. Das folgende Fragment führt daher zu einer Fehlermeldung des Compilers, da `zahl` keinen definierten Wert hat.

```
int zahl;  
Console.WriteLine("zahl hat den Wert {0}", zahl);
```

Achten Sie daher darauf, dass eine Variable in jedem Fall einen eindeutig festgelegten Wert hat – zum Beispiel durch eine Zuweisung direkt bei der Vereinbarung oder auch durch eine Zuweisung beim ersten „richtigen“ Zugriff. Diese erste Zuweisung eines Wertes nennt man auch **Initialisierung**.

Wenn Sie die Initialisierung einer lokalen Variablen vergessen, können Sie das Programm nicht ausführen lassen.



3.5 Weitere Operatoren

Zu Beginn dieses Kapitels haben Sie bereits die arithmetischen Operatoren kennengelernt. Im Folgenden werden wir Ihnen noch einige weitere Operatoren vorstellen, die bei der Arbeit mit Variablen häufig verwendet werden.

Beginnen wir mit dem **Inkrement-** und dem **Dekrement-Operator**. Mit diesen Operatoren können Sie den Wert einer ganzzahligen Variablen um 1 erhöhen (Inkrement) beziehungsweise um 1 verringern (Dekrement). Der Inkrement-Operator besteht aus zwei Pluszeichen `++`, der Dekrement-Operator aus zwei Minuszeichen `--`.

Anders als bei den bisher vorgestellten Operatoren haben sowohl der Inkrement- als auch der Dekrement-Operator nur **einen** Operanden. Schauen wir uns einige Beispiele an:

```
++zahl;
ausgabe = ++zahl;
ausgabe = --zahl;
```

Im ersten Beispiel wird der Wert von `zahl` um 1 erhöht und dann der Variablen `zahl` direkt wieder zugewiesen. Die Wirkung von `++zahl` entspricht exakt dem Ausdruck `zahl = zahl + 1`. Sie sparen also vor allem Tipparbeit.

Im zweiten Beispiel wird der Wert von `zahl` durch den Inkrement-Operator um 1 erhöht und dann der Variablen `ausgabe` zugewiesen. Diese Konstruktion entspricht damit den Anweisungen

```
zahl = zahl + 1;
ausgabe = zahl;
```

Im dritten Beispiel schließlich wird der Wert von `zahl` durch den Dekrement-Operator um 1 verringert und dann der Variablen `ausgabe` zugewiesen. Das entspricht den Anweisungen

```
zahl = zahl - 1;
ausgabe = zahl;
```

Der Inkrement- und der Dekrement-Operator können auch **hinter** dem Operanden angegeben werden.

```
ausgabe = zahl++;
ausgabe = zahl--;
```

In diesem Fall wird allerdings die Zuweisung vor der Veränderung von `zahl` durchgeführt. Das heißt, der Ausdruck

```
ausgabe = zahl++;
```

entspricht den Anweisungen

```
ausgabe = zahl;
zahl = zahl + 1;
```

Hinweise:

Bei Ausdrücken wie `zahl++` oder `++zahl` ohne Zuweisung des Ergebnisses an eine andere Variable spielt die Position des Operators eigentlich keine Rolle. In beiden Fällen erhöht sich der Wert von `zahl` um 1.

Im praktischen Einsatz finden Sie die Inkrement- und Dekrement-Operatoren im Code weiter unten.

Kommen wir jetzt noch zu den **vereinfachenden Operatoren**.

Mit den vereinfachenden Operatoren können Sie vor allem Tipparbeit sparen. Sie erinnern sich sicher noch an die Ausdrücke, in denen links und rechts vom Zuweisungsoperator dieselbe Variable und sonst nur konstante Zahlen vorkamen – zum Beispiel:

```
zahl = zahl + 5;
zahl = zahl / 7;
zahl = zahl * 9;
```

C# erlaubt für solche Ausdrücke eine verkürzte Schreibweise in der folgenden Form:

```
zahl += 5; //entspricht zahl = zahl + 5;
zahl /= 7; //entspricht zahl = zahl / 7;
zahl *= 9; //entspricht zahl = zahl * 9;
```

Dabei sind vor dem Zuweisungsoperator alle arithmetischen Operatoren erlaubt.

Schauen wir uns jetzt die Inkrement- und Dekrementoperatoren sowie die vereinfachenden Operatoren in einem Code an:

```
/* #####
Beispiele für weitere Operatoren
#####
using System;

namespace Cshp02d_03_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl = 5;
            int ausgabe;

            // einfaches Inkrement
            Console.WriteLine("zahl hat den Wert {0}", zahl);
            ++zahl;
            // bitte in einer Zeile eingeben
            Console.WriteLine("zahl hat nach ++zahl den
Wert {0}\n", zahl);
            // Inkrement nach der Zuweisung
            ausgabe = zahl++;
            // bitte jeweils in einer Zeile eingeben
            Console.WriteLine("ausgabe hat den Wert {0}", ausgabe);
            Console.WriteLine("zahl hat jetzt den Wert {0}\n", zahl);

            // Inkrement vor der Zuweisung
            ausgabe = ++zahl;
            // bitte jeweils in einer Zeile eingeben
            Console.WriteLine("ausgabe hat den Wert
{0}", ausgabe);
            Console.WriteLine("zahl hat jetzt den Wert
{0}\n", zahl);
        }
    }
}
```

```
//entspricht zahl = zahl / 7
zahl /= 7;
//bitte in einer Zeile eingeben
Console.WriteLine("zahl /= 7 liefert den Wert
{0}\n", zahl);

//entspricht zahl = zahl + 10
zahl += 10;
Console.WriteLine("zahl +=10 liefert den Wert {0}", zahl);
}
}
```

Code 3.5: Beispiel für weitere Operatoren

Das Programm erzeugt folgende Ausgabe:

```
zahl hat den Wert 5  
zahl hat nach ++zahl den Wert 6
```

```
ausgabe hat den Wert 6  
zahl hat jetzt den Wert 7
```

```
ausgabe hat den Wert 8  
zahl hat jetzt den Wert 8
```

zahl /= 7 liefert den Wert 1

zahl +=10 liefert den Wert 11

Beachten Sie vor allem die unterschiedliche Wirkung bei der Position des Inkrement-Operators in den ersten Anweisungen. Steht der Inkrement-Operator hinter dem Operanden, erfolgt die Veränderung erst nach der Zuweisung.

Hinweis:

Ob Sie die Inkrement- und Dekrement-Operatoren sowie die vereinfachenden Operatoren einsetzen, ist auch eine Frage des persönlichen Stils und Geschmacks. Sie ersparen sich damit ein wenig Schreibarbeit, aber vor allem die vereinfachenden Operatoren machen einen Quelltext auch recht schwer zu lesen.

So viel zu den weiteren Operatoren. Kommen wir nun zu den symbolischen Konstanten.

3.6 Symbolische Konstanten

Neben den Variablen können Sie in einem C#-Programm auch **symbolische Konstanten** verwenden. Diese symbolischen Konstanten erhalten einmal einen festen Wert, der danach nicht mehr verändert werden kann.

Hinweis:

Symbolische Konstanten werden auch einfach nur Konstante genannt. Für konstante Werte, die einem Datenobjekt zugewiesen werden, wird dagegen der Begriff Literal benutzt. Beispiele für Literale sind Zahlen wie 1, 2 oder 100 oder Zeichenketten wie „Wert 1“.

Symbolische Konstanten sollten Sie immer dann verwenden, wenn Sie in Ihrem Programm mehrfach einen unveränderlichen Wert verwenden – zum Beispiel einen Schlüssel für die Berechnung der Mehrwertsteuer oder mathematische Werte wie die Zahl π (Pi). Da der Wert der symbolischen Konstanten einmal festgelegt wird, können Sie – wenn erforderlich – später genau an dieser einen Stelle den Wert verändern – zum Beispiel bei Erhöhungen der Mehrwertsteuer.

Wenn Sie den Wert dagegen direkt in den Quelltext schreiben, müssten Sie jedes Vorkommen des Wertes einzeln ändern. Übersehen Sie dabei auch nur eine Stelle, treten Fehler in den Berechnungen auf.

Die Vereinbarung einer symbolischen Konstanten sieht so aus:

```
const Datentyp Bezeichner = Wert;
```

Eingeleitet wird die Vereinbarung durch das Schlüsselwort `const`. Dann folgen der Datentyp der Konstanten und der Bezeichner. Für den Bezeichner gelten dieselben Regeln wie für Variablenbezeichner. Abgeschlossen wird die Vereinbarung durch den Zuweisungsoperator `=`, den Wert der Konstanten und ein Semikolon.

Hinweis:

Mit den Datentypen beschäftigen wir uns ausführlich im nächsten Kapitel. Hier gehen wir davon aus, dass die symbolische Konstante einen ganzzahligen Wert erhalten soll.

Die Anweisung

```
const int konstante = 10;
```

vereinbart eine symbolische Konstante mit dem Namen `konstante` und dem Wert 10. Der Datentyp ist `int` – also eine ganze Zahl.

Symbolische Konstanten können Sie als Teil eines Ausdrucks verwenden – zum Beispiel in einer Anweisung wie

```
zahl = 50 * konstante;
```

Sie können aber auf keinen Fall den Wert einer symbolischen Konstanten durch das Programm verändern lassen. Jeder Versuch wird vom Compiler mit einer Fehlermeldung quittiert – zum Beispiel im folgenden Code:

```
/* #####Symbolische Konstanten#####
(das Programm lässt sich nicht übersetzen)
##### */

using System;

namespace Cshp02d_03_06
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Vereinbarung der symbolischen Konstanten
            const int konstante = 10;
            // Vereinbarung der Variablen
            int zahl2 = 15;
            int ausgabe;

            //Ausgabe der Konstante
            //bitte in einer Zeile eingeben
            Console.WriteLine("Konstante hat den Wert
{0}", konstante);
            //Rechnung mit der Konstanten
            ausgabe = zahl2 + konstante;
            //Ausgabe des Ergebnisses
            Console.WriteLine("ausgabe hat den Wert {0}", ausgabe);
            //DAS GEHT NICHT!
            konstante = 5;
        }
    }
}
```

Code 3.6: Beispiel für die Verwendung von symbolischen Konstanten

Wenn Sie diesen Code eingeben, meldet der Compiler, dass die linke Seite einer Zuweisung zum Beispiel eine Variable oder eine Eigenschaft sein muss. Das heißt also, wenn Sie eine symbolische Konstante auf der linken Seite des Zuweisungsoperators sehen, können Sie praktisch immer davon ausgehen, dass es sich um einen Fehler handelt. Die einzige Ausnahme bildet die Vereinbarung der symbolischen Konstanten. Auf der rechten Seite des Zuweisungsoperators dagegen können symbolische Konstanten beliebig erscheinen.

Setzen Sie jetzt einmal an den Anfang der Zeile

```
konstante = 5;
```

die Kommentarzeichen //. Sie werden sehen, das Programm funktioniert danach. Verändern Sie dann auch einmal zum Test den Wert der symbolischen Konstanten bei der Vereinbarung.

Zusammenfassung

C# kennt fünf wichtige arithmetische Operatoren. Neben den vier Grundrechenarten wird noch die Modulo-Operation unterstützt.

Für die Auswertung der Operatoren gelten feste Regeln. So geht zum Beispiel Punktrechnung vor Strichrechnung.

Variablen werden für die Speicherung von Werten beim Ablauf eines Programms verwendet.

Vor dem ersten Einsatz müssen Sie eine Variable vereinbaren. Dazu geben Sie den Datentyp und den Bezeichner der Variablen an.

Mit den Inkrement- und Dekrement-Operatoren und den vereinfachenden Operatoren können Sie Tipparbeit sparen.

Symbolische Konstanten erhalten einmal einen Wert, der während des Programmablaufs nicht verändert werden kann.

Aufgaben zur Selbstüberprüfung

- 3.1 Schreiben Sie ein Programm, das folgende Rechnung durchführt und das Ergebnis auf dem Bildschirm ausgibt. Wie lautet das Ergebnis der Rechnung?

$$(((20 * 4) \% 3 + 8) * 10 + 8) / 2$$

- 3.2 Geben Sie bitte bei den folgenden Namen an, ob es sich um gültige Variablenbezeichner handelt. Wenn nicht, begründen Sie, warum der Bezeichner nicht verwendet werden kann.

- a) 1Aenderung _____
- b) aenderung _____
- c) _zahl _____
- d) variable_1 _____

e) using _____

f) nummer*3 _____

g) info 1 _____

- 3.3 Wie wird der Operator genannt, mit dem Sie einer Variablen einen Wert zuweisen? Wie wird dieser Operator geschrieben?

Digitized by srujanika@gmail.com

- 3.4 Wie lautet das Schlüsselwort zur Vereinbarung einer symbolischen Konstanten?

In welcher Reihenfolge werden die Operatoren in dem folgenden Ausdruck ausgewertet? Notieren Sie die einzelnen Zwischenschritte.

$$e = \pm a * -b / c + d$$

4 Datentypen

In den vorigen Kapiteln war bereits mehrfach vom Datentyp die Rede. In diesem Kapitel werden wir uns genauer ansehen, was sich dahinter verbirgt.

Wie Sie ja bereits wissen, können Sie in Variablen unterschiedliche Arten von Daten speichern – zum Beispiel ganze Zahlen oder auch Zahlen mit Nachkommastellen. Damit der Compiler weiß, welche Daten Sie in einer Variablen ablegen möchten, müssen Sie bei der Vereinbarung einer Variablen den **Datentyp** angeben. Damit haben wir uns ja schon kurz im letzten Kapitel beschäftigt.

Der Datentyp übernimmt zwei wichtige Funktionen:

1. Der Compiler reserviert entsprechend Platz im Speicher des Computers. Damit wird automatisch auch der mögliche Wertebereich der Variablen festgelegt.
2. Der Compiler interpretiert die gespeicherten Informationen anhand des Typs. So kann er zum Beispiel unterscheiden, ob es sich bei den Daten um eine ganze Zahl oder eine Zahl mit Nachkommastellen handelt.

Damit Sie leichter nachvollziehen können, warum der Datentyp so wichtig ist, wollen wir uns jetzt aber erst einmal ansehen, wie Werte überhaupt im Computer gespeichert werden.

Hinweis:

Wenn Sie mit der Darstellung von Werten im Dualsystem vertraut sind, können Sie den folgenden Abschnitt auch überspringen.

4.1 Darstellung von Werten im Computer

Wie Sie wissen, besteht ein Computer aus elektronischen Bauelementen. Für die Speicherung von Werten werden die Zustände Strom an („1“) und Strom aus („0“) verwendet. Daher müssen alle Werte, die verarbeitet werden sollen, als Nullen und Einsen dargestellt werden.

Diese Darstellung mit zwei Werten nennt man binäre Darstellung.

Das dazugehörige Zahlensystem heißt **Dualsystem**. Es arbeitet mit den beiden Ziffern 0 und 1 und dem Stellenwert 2. Eine Zahl wird durch Potenzen des Stellenwerts 2 dargestellt.



Die Zahl 101 aus dem Dualsystem steht zum Beispiel für die Zahl 5 im Dezimalsystem.

$$\begin{aligned}
 & 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\
 = & 1 * 4 + 0 * 2 + 1 * 1 \\
 = & 4 + 0 + 1 \\
 = & 5
 \end{aligned}$$

Die Zahl 1 0110 1101 aus dem Dualsystem steht dann für die Zahl 365 im Dezimalsystem.

$$\begin{aligned}
 & 1 * 2^8 + 0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\
 = & 1 * 256 + 0 * 128 + 1 * 64 + 1 * 32 + 0 * 16 + 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 \\
 = & 365
 \end{aligned}$$



Zur Auffrischung: Das Dezimalsystem

Das gewohnte Dezimalsystem arbeitet mit ähnlichen Verfahren wie das Dualsystem. Lediglich die verwendeten Ziffern und der Stellenwert unterscheiden sich. Im Dezimalsystem werden die Ziffern 0 bis 9 verwendet. Der Stellenwert ist 10.

Die Zahl 365 wird im Dezimalsystem so dargestellt:

$$\begin{aligned}
 & 3 * 10^2 + 6 * 10^1 + 5 * 10^0 \\
 = & 3 * 100 + 6 * 10 + 5 * 1
 \end{aligned}$$

Eine einzelne Ziffer des Dualsystems – also 0 oder 1 – ist ein **Bit**. Mit diesem Bit können die beiden Zustände „Strom an“ beziehungsweise „Strom aus“ dargestellt werden.

Der Speicherplatz, den ein Wert beansprucht, lässt sich recht einfach durch das Abzählen der Bits ermitteln. Der duale Wert 1 0110 1101 besteht zum Beispiel aus neun Ziffern. Damit benötigt der Compiler also auch mindestens neun Bits für die Speicherung.

Um nun nicht für jeden Wert einzelne Bits im Speicher reservieren zu müssen, werden in der Regel größere Einheiten gebildet – zum Beispiel auf der Basis eines **Bytes**. Es umfasst genau acht Bits.

Die Anzahl der Bits beziehungsweise Bytes, die für einen Datentyp im Speicher reserviert werden, entscheidet dann über den Wertebereich, der mit einer entsprechenden Variablen dargestellt werden kann. Schauen wir uns das einmal am Beispiel des Datentyps `byte` an.

Er reserviert – wie der Name schon vermuten lässt – ein Byte beziehungsweise acht Bits Speicherplatz. Damit lassen sich genau 2^8 (= 256) unterschiedliche Ziffernkombinationen abbilden. Der kleinste darstellbare Wert wäre dann 0 und der größte darstellbare Wert 255. Der kleinste mögliche Wert ergibt sich, wenn alle Bits den Wert 0 haben und der größte mögliche Wert entsprechend, wenn alle Bits den Wert 1 haben.

Für den Datentyp `int` dagegen werden 32 Bits Speicherplatz reserviert. Er kann damit bereits 2^{32} (= 4 294 967 296) unterschiedliche Ziffernkombinationen darstellen. Der kleinste darstellbare Wert wäre wieder die 0 und der größte Wert 4 294 967 295 – zumindest theoretisch. Denn der Datentyp `int` kann sowohl positive als auch negative Zahlen darstellen. Und für die Darstellung des Vorzeichens wird eins von den 32 Bits verwendet. Damit bleiben für die eigentlichen Zahlen nur noch 31 Bits übrig. Der darstellbare Zahlenraum reicht damit von -2 147 483 648 bis +2 147 483 647.

Neben ganzen Zahlen lassen sich mit Bitfolgen auch nahezu beliebige andere Informationen darstellen – zum Beispiel Zahlen mit Nachkommastellen. Hier ist das Verfahren allerdings ein wenig komplizierter als bei „einfachen“ Zahlen.

Denn die Darstellung erfolgt als **Gleitkommawert mit Mantisse und Exponent**. Schauen wir uns das einmal an einigen Beispielen an:

Die Zahl 6 324,098 lässt sich durch den Ausdruck $63,24098 \cdot 10^2$ darstellen. 63,24098 wäre dabei die Mantisse und 2 im Ausdruck 10² der Exponent.

Der Ausdruck 10² steht für $10 \cdot 10$ – also für 100. Der Wert 10 wird verwendet, da er die Basiszahl des Dezimalsystems ist.



Die Zahl 34 897,12 könnte auch als $348,9712 \cdot 10^2$ abgebildet werden. Hier wäre dann 348,9712 die Mantisse und 2 im Ausdruck 10² wieder der Exponent.

Durch einen anderen Exponenten können Sie gezielt festlegen, wie viele Zahlen vor dem Komma stehen. Die Zahl 6 324,098 lässt sich zum Beispiel auch als $6,324098 \cdot 10^3$ darstellen. Die Zahl 34 897,12 könnte auch durch $3,489712 \cdot 10^4$ abgebildet werden.

Der Exponent legt fest, um wie viele Stellen das Komma in einer Zahl nach links verschoben werden soll. Der Ausdruck 10³ verschiebt das Komma zum Beispiel im Ergebnis um drei Stellen nach links und der Ausdruck 10⁴ verschiebt es um vier Stellen nach links.



Dieser Effekt wird für die **Normierung** von Gleitkommazahlen verwendet. Durch die Wahl eines entsprechenden Exponenten wird dafür gesorgt, dass eine Zahl zwischen 1 und 9,999... als Mantisse verwendet wird. Vor dem Komma steht dabei immer genau eine Ziffer. Damit ist klar, dass das Komma immer an der zweiten Position der Zahl steht. Wie viele Stellen hinter dem Komma folgen, hängt von der gewünschten Genauigkeit ab.

Auch Zahlen zwischen 0 und 1 lassen sich mit der normierten Darstellung abbilden. Dazu müssen lediglich negative Exponenten verwendet werden, damit vor dem Komma keine Null mehr steht. So liefert der Ausdruck 10⁻¹ zum Beispiel den Wert 0,1 oder der Ausdruck 10⁻² den Wert 0,01.

Zur Auffrischung:

Durch einen negativen Exponenten wird eine positive Zahl kleiner.



Die normierte Darstellung lässt sich auch für Zahlen in anderen Zahlensystemen verwenden – zum Beispiel eben für das Dualsystem. Hier hat dann die Mantisse einen dualen Wert zwischen 1 und 1,111 ... Der Exponent stellt entsprechend Potenzen der Basiszahl 2 dar.

Um solche Zahlen nun als Bitfolge speichern zu können, wird ein wenig „in die Trickkiste gegriffen“: Die Zahl vor dem Komma in der Mantisse kann ja immer nur die Ziffer 1 sein. Daher müssen nur der Nachkommateil der Mantisse, der Exponent und das Vorzeichen der gesamten Zahl als einzelne Bits abgelegt werden. Für Zahlen mit einfacher Genauigkeit können diese Daten zum Beispiel so auf die einzelnen Bits verteilt werden:



Abb. 4.1: Bitweise Darstellung einer Zahl mit einfacher Genauigkeit

Das Bit ganz links wird für die Speicherung des Vorzeichens benutzt. Dann folgen 8 Bits für die Speicherung des Exponenten und anschließend 23 Bits für die Speicherung der Nachkommastellen der Mantisse.

Damit könnten dann Zahlen in den Bereichen $-3,40 \cdot 10^{38}$ bis $-1,18 \cdot 10^{-38}$ und $1,18 \cdot 10^{-38}$ bis $3,40 \cdot 10^{38}$ bei acht Stellen Genauigkeit gespeichert werden.



Die Darstellung von Gleitkommazahlen wird durch Normen festgelegt – zum Beispiel durch die Norm IEEE 754r.

Die Speicherung als Gleitkommazahl ist zwar sehr kompakt, hat aber auch eine ganz besondere Tücke: Da die Nachkommastellen durch fortlaufende Additionen von negativen Potenzen der Zahl 2 gebildet werden, können Ungenauigkeiten auftreten. So lässt sich zum Beispiel die Zahl 0,33 nicht exakt darstellen, sondern nur annähernd exakt als 0,329999... Und diese Ungenauigkeiten können in der Summe zu Rundungsfehlern führen.

Die Darstellung von Zeichen über Bitfolgen dagegen ist wieder relativ einfach. Hier wird jedem Zeichen ein nummerischer Code zugeordnet, der dann über eine Kombination von 0 und 1 abgebildet werden kann. Weitverbreitete Codes für solche Zuordnungen sind zum Beispiel ASCII oder Unicode.



ASCII steht für *American Standard Code for Information Interchange* – übersetzt etwa „Amerikanischer Standard-Code zum Informationsaustausch“.

Nachdem Sie jetzt wissen, wie unterschiedliche Daten intern abgebildet werden können und warum der Datentyp bei Zahlen immer auch den darstellbaren Zahlenraum festlegt, bleibt nur noch eine Frage: Woher weiß der Computer, ob sich hinter einem Bitmuster eine ganze Zahl, ein Buchstabe oder eine Zahl mit Nachkommastellen verbirgt? Denn an dem Bitmuster selbst ist das ja nicht zu erkennen. Es besteht immer nur aus einer mehr oder weniger langen Folge von Nullen und Einsen.

Und hier kommt dann wieder der Datentyp ins Spiel. Der Computer „merkt“ sich – etwas vereinfacht dargestellt –, wie er die Werte an einer Speicherstelle interpretieren soll. Wenn Sie also eine Variable vom Typ `int` vereinbaren, reserviert der Computer auf der einen Seite den erforderlichen Speicher, weiß auf der anderen Seite aber auch, dass er die Daten an dieser Speicherstelle als ganze Zahl interpretieren muss.

Wenn Ihnen jetzt der Kopf vor lauter Theorie brummt – ein Trost: Sie müssen die Zahlen und Zeichen, die Sie in einem Programm verwenden wollen, natürlich nicht selbst in das Dualsystem transportieren. Diese Arbeit nimmt Ihnen der Compiler ab. Sie müssen ihm lediglich mitteilen, welchen Datentyp Sie verwenden wollen. Alles andere geschieht dann automatisch. Trotzdem hat sich dieser kleine Ausflug in die Welt des Dualsystems gelohnt. Denn Sie werden gleich einige „Problemchen“ einfacher nachvollziehen können.

4.2 Die Datentypen von C#

Schauen wir uns jetzt die verschiedenen Datentypen von Visual C# im Detail an. Grundsätzlich werden folgende Datentypen unterschieden:

- ganze Zahlen,
- Gleitkommazahlen,
- einzelne Zeichen und Zeichenketten sowie
- logische Werte wie wahr und falsch beziehungsweise `true` und `false`.

4.2.1 Datentypen für ganze Zahlen

Für ganze Zahlen kennt C# insgesamt acht verschiedene Typen, die sich im Wertebereich und in der Darstellung des Vorzeichens unterscheiden. Details zu diesen Datentypen finden Sie in der folgenden Tabelle:

Tab. 4.1: Die Datentypen von C# für ganze Zahlen

Name	Speicherbedarf	Vorzeichen	Wertebereich
byte	8 Bits	nein	0 bis 255
sbyte	8 Bits	ja	-128 bis 127
short	16 Bits	ja	-32 768 bis 32 767
ushort	16 Bits	nein	0 bis 65 535
int	32 Bits	ja	-2 147 483 648 bis 2 147 483 647
uint	32 Bits	nein	0 bis 4 294 967 295
long	64 Bits	ja	-9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807
ulong	64 Bits	nein	0 bis 18 446 744 073 709 551 615

Hinweis:

Zur leichteren Lesbarkeit stellen wir große Zahlen immer mit einer Leerstelle als Tausendertrennzeichen dar. Intern werden die Zahlen aber ohne diese Leerstelle abgebildet. In einem Quelltext müssen Sie die Werte daher immer ohne das Trennzeichen eingeben – also 32767 und nicht 32 767. Sie können aber bei den meisten numerischen Typen den Unterstrich `_` als Trennzeichen verwenden. Der Wert 32767 lässt sich in einem Code also auch als `32_767` darstellen.

Tipp:

Ob ein Datentyp ein Vorzeichen abbildet oder nicht, können Sie sich bei den meisten Typen anhand des ersten Zeichens merken. Das u steht nämlich für unsigned (engl.: „vorzeichenlos“). Lediglich beim Datentyp sbyte funktioniert diese „Eselsbrücke“ nicht. Hier erfolgt die Markierung genau in die andere Richtung. Das s steht für signed (engl.: „mit Vorzeichen“).

Die Liste aus der vorigen Tabelle können Sie auch sehr einfach durch ein Programm erstellen lassen. Der entsprechende Code sieht so aus:

```
/* ##### Eine Liste der ganzzahligen Typen #####
using System;
namespace Cshp02d_04_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            byte byteVariable = 1;
            sbyte sByteVariable = 1;
            short int16Variable = 1;
            ushort UInt16Variable = 1;
            int int32Variable = 1;
            uint UInt32Variable = 1;
            long int64Variable = 1;
            ulong UInt64Variable = 1;
            //Ausgabe der Tabelle
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("{0}\tvon {1} bis {2}",
                byteVariable.GetType(), byte.MinValue, byte.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                sByteVariable.GetType(), sbyte.MinValue, sbyte.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                int16Variable.GetType(), short.MinValue, short.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                UInt16Variable.GetType(), ushort.MinValue,
                ushort.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                int32Variable.GetType(), int.MinValue, int.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                UInt32Variable.GetType(), uint.MinValue, uint.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                int64Variable.GetType(), long.MinValue, long.MaxValue);
            Console.WriteLine("{0}\tvon {1} bis {2}",
                UInt64Variable.GetType(), ulong.MinValue, ulong.MaxValue);
        }
    }
}
```

Code 4.1: Eine Liste der ganzzahligen Datentypen

Zuerst werden Variablen für die unterschiedlichen Datentypen vereinbart. In den folgenden Anweisungen geben wir dann für jede Variable den Datentyp sowie den minimalen und den maximalen Wert aus. Den Datentyp erhalten Sie dabei durch den Ausdruck `<bezeichner>.GetType()`³ und die beiden Grenzwerte durch die Ausdrücke `<Datentyp>.MinValue` und `<Datentyp>..MaxValue`⁴.

Probieren Sie den Code einfach einmal aus. Sie werden sehen: Vorn in der Liste finden Sie den Namen des Typs, dahinter wird dann der Wertebereich dargestellt.

Hinweise:

Beim Namen wird der Name des Datentyps aus dem .NET Framework angezeigt.
Mit diesem Typ beschäftigen wir uns gleich.

Sie müssen den Variablen im vorigen Code Werte zuweisen – auch wenn wir diese Werte gar nicht weiter benutzen. Denn ohne die Zuweisung weigert sich der Compiler, das Programm auszuführen.

Welche der verschiedenen Typen Sie in Ihrem Programm verwenden, hängt von der größten Zahl ab, die Sie wahrscheinlich verarbeiten werden. Wenn Sie nämlich einen Datentyp mit einem zu kleinen Wertebereich benutzen, kann ein **Überlauf** eintreten – nämlich dann, wenn Sie den Wertebereich überschreiten.

Schauen wir uns diesen Überlauf an einem Beispiel an:

Beispiel 4.1:

Sie arbeiten mit einer Variablen vom Typ `byte` und weisen dieser Variablen den maximalen Wert 255 zu. Wenn Sie nun den Wert 1 addieren, ist das Ergebnis nicht wie erwartet 256, sondern 0. Dieser vermeintliche Fehler entsteht dadurch, dass der Wertebereich verlassen wird und C# beim nächsten folgenden Wert im Wertebereich weiterarbeitet. Und dieser nächste folgende Wert ist beim Datentyp `byte` der Wert 0.

Diesen Effekt können Sie mit dem folgenden Code selbst ausprobieren.

```
/* ##### Ein provozierter Überlauf ##### */
using System;
namespace Cshp02d_04_02
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            byte byteVariable = 255;
```

3. *Get type* bedeutet übersetzt so viel wie „beschaffe Typ“.

4. `MinValue` steht für *minimum value* und `.MaxValue` für *maximum value*. Übersetzt bedeuten die beiden Ausdrücke so viel wie „minimaler Wert“ und „maximaler Wert“.

```
//bitte in einer Zeile eingeben
Console.WriteLine("Die Variable hat den Wert:
{0}",byteVariable);
//jetzt erhöhen wir den Wert um 1 über den
//Inkrement-Operator
byteVariable++;
//bitte in einer Zeile eingeben
Console.WriteLine("255 + 1 ist gleich
{0}????",byteVariable);
}
}
```

Code 4.2: Ein provoziert Überlauf

Die erste Ausgabe liefert noch ganz korrekt 255. Bei der zweiten Ausgabe dagegen erscheint der Wert 0 – die Variable `byteVariable` ist „übergelaufen“. Auf diesen Überlauf weist Sie der Compiler in der Standardeinstellung noch nicht einmal hin. Nur bei der direkten Zuweisung eines zu großen oder zu kleinen Wertes erscheint eine Fehlermeldung, dass der Wert nicht passt.



Überläufe können zu sehr unangenehmen Fehlern führen, die schwer zu finden sind. Benutzen Sie daher im Zweifelsfall lieber zu große Datentypen als zu kleine. Damit verschwenden Sie zwar möglicherweise ein wenig Speicher, aber das ist immer noch das „geringere Übel“.

Das heißt allerdings nicht, dass Sie kurzerhand immer mit dem größten möglichen Datentyp arbeiten sollten, um Probleme mit Überläufen zu vermeiden. Überlegen Sie sich vorher, welcher Datentyp der Aufgabe gerecht wird, und denken Sie dabei auch daran, welche Dimensionen der Wert möglicherweise annehmen könnte.

Tipp:

Sie können das Standardverhalten bei Überläufen über die Operatoren `checked` und `unchecked` gezielt verändern. Details zu diesen beiden Operatoren schlagen Sie bei Interesse bitte in der Hilfe nach.

So viel zu den Datentypen für ganze Zahlen.

4.2.2 Datentypen für Gleitkommazahlen

Die Datentypen für Gleitkommazahlen sind – anders als die Typen für ganze Zahlen – recht übersichtlich. Unterschieden werden nämlich im Wesentlichen lediglich zwei Typen. Der Datentyp `float` beansprucht 32 Bits Speicherplatz und stellt Zahlen im Wertebereich von $+1.5 \cdot 10^{-45}$ bis $+3.40 \cdot 10^{38}$ mit einer Genauigkeit von sieben Stellen dar.

Der Datentyp `double` dagegen beansprucht 64 Bits und stellt Zahlen im Wertebereich von $+5 \cdot 10^{-324}$ bis $+1.7 \cdot 10^{308}$ mit einer Genauigkeit von mindestens 15 Stellen dar.

Bitte beachten Sie:

Sowohl der Typ `float` als auch der Typ `double` können keine extrem kleinen Zahlen darstellen. Für den Typ `float` ist der kleinste mögliche positive Wert zum Beispiel $1,5 \cdot 10^{-45}$. Hinter dem Komma stehen bei diesem Wert allerdings immer noch 44 Nullen.



Die Genauigkeit der beiden Datentypen bezieht sich auf sämtliche Stellen der Zahl – also sowohl auf die Stellen vor dem Komma als auch auf die Stellen nach dem Komma. Da Gleitkommazahlen intern immer in der normierten Darstellung abgebildet werden, garantiert Ihnen der Datentyp `float`, dass sieben Nachkommastellen korrekt abgebildet werden. Bei der Ausgabe dagegen hängt die Anzahl der korrekt dargestellten Nachkommastellen auch von den Stellen vor dem Komma ab.

Schauen wir uns dazu einige Beispiele an:

Beispiel 4.2:

Die Zahl 1,22129219 hat insgesamt neun Stellen – eine Stelle vor dem Komma und acht Stellen hinter dem Komma. Wenn Sie diese Zahl mit einer Genauigkeit von acht Stellen ausgeben, geht die letzte Stelle – also die 9 – verloren, da die maximale Genauigkeit überschritten wird.

Die Zahl 122 129,212156 hat insgesamt 12 Stellen – sechs Stellen vor dem Komma und sechs Stellen hinter dem Komma. Das heißt, bei einer Genauigkeit von acht Stellen gehen bei der Ausgabe die letzten vier Nachkommastellen verloren beziehungsweise werden falsch dargestellt.

Bevor Sie weiterlesen ...

Erstellen Sie einmal selbst ein Programm, das Ihnen für die Datentypen `float` und `double` den Namen sowie den Wertebereich ausgibt. Orientieren Sie sich dabei am Code 4.1.

Schauen wir uns jetzt den Umgang mit Gleitkommazahlen an einem konkreten Beispiel an.

Wir lassen in einem Programm einige Gleitkommazahlen zunächst mit den Standardeinstellungen ausgeben und setzen dann die Genauigkeit auf 18 Stellen. Danach wiederholen wir dann die Ausgaben. Erläuterungen zu den einzelnen Anweisungen finden Sie wie gewohnt hinter dem vollständigen Code.

```
/* ##### Die Gleitkommatypen von C# #####
using System;
namespace Cshp02d_04_03
{
    class Program
    {
```

Code 4.3: Datentypen für Gleitkommazahlen im praktischen Einsatz

Das Programm sollte folgende Ausgaben erzeugen:

Ausgabe von Gleitkommazahlen:

Standardeinstellungen:

Große Zahlen (10000000000): 1E+10

Kommazahlen (1.333333333333333333)

Als float: 1,333333

Als double: 1,333333333333333

18 SCOTT

Große Zahlen (10000000000): 10000000000

Kommazahlen (1.333333333333333333)

Als float: 1,3333337

Als double: 1,333333333333333

Als float: 1001,33331

Sehen wir uns an, wie diese Ausgaben zustande kommen. Zunächst werden in den Zeilen

```
floatVariable = 10000000000;
Console.WriteLine("Große Zahlen (10000000000): {0}",
floatVariable);
Console.WriteLine("Kommazahlen (1.333333333333333333)");
floatVariable = 1.333333333333333F;
Console.WriteLine("Als float: {0}", floatVariable);
doubleVariable = 1.333333333333333;
Console.WriteLine("Als double: {0}", doubleVariable);
```

einige Werte zugewiesen und in den Standardeinstellungen ausgegeben. Die Zahl 10 000 000 000 erscheint dabei in der Exponentialschreibweise als 1E+10. Die Zahl vor dem E steht für die Mantisse und die Zahl hinter dem E gibt den Exponenten an. 1E+10 müssen Sie also als $1 \cdot 10^{10}$ lesen.

Bei der Nachkommazahl erscheinen für den Datentyp `float` insgesamt sieben Stellen – obwohl wir eine Zahl mit 20 Nachkommastellen zugewiesen haben. Beim Datentyp `double` dagegen werden 15 Stellen ausgegeben, 14 davon nach dem Komma. Die restlichen Stellen werden bei der Ausgabe in den Standardeinstellungen schlicht und einfach abgeschnitten.

Bitte beachten Sie:

Bei der Zuweisung auf einen Gleitkommatyp müssen Sie den Punkt als Dezimaltrennzeichen verwenden. Wenn Sie das Komma verwenden, erscheint eine Fehlermeldung. Der Einsatz des Punkts ist erforderlich, weil in den USA das Tausender trennzeichen und das Dezimaltrennzeichen genau andersherum benutzt werden als in Deutschland üblich. Tausender werden durch ein Komma getrennt und Dezimalstellen durch einen Punkt. Die Zahl 1.234,56 wird in den USA also 1,234.56 geschrieben.

C# behandelt alle konstanten Gleitkommazahlen als Typ `double`. Wenn Sie eine konstante Gleitkommazahl als `float` interpretieren möchten, müssen Sie das Zeichen `F` hinter die Zahl stellen.



In den folgenden Ausgaben setzen wir dann die Ausgabe der Genauigkeit auf insgesamt 18 Stellen. Das erfolgt etwas versteckt durch die Angabe :G18 hinter dem Platzhalter `{0}`. Damit Sie die Stelle genau erkennen, haben wir sie in der folgenden Anweisung fett markiert.

```
Console.WriteLine("Große Zahlen (10000000000):
{0:G18}", floatVariable);
```

Durch diese **Zahlenformatzeichenfolge** wird der Compiler angewiesen, die Zahl in einem allgemeinen Format (das `G`) mit 18 Stellen (die `18`) auszugeben. Bei den Stellen werden dabei sämtliche Stellen mitgezählt – also sowohl die Stellen vor und nach dem Komma.

Die Zahl `10 000 000 000` erscheint danach in der gewohnten Darstellung. Beim Datentyp `float` wird die Ausgabe trotzdem nach acht Nachkommastellen beendet. Wie Sie sehen, ist aber die letzte Stelle bereits nicht mehr korrekt. Die Genauigkeit des Datentyps reicht nicht mehr aus. Beim Datentyp `double` dagegen werden lediglich zwei Stellen nach dem Komma mehr ausgegeben. Sonst hat sich nichts geändert.

Interessant sind dann noch einmal die letzten beiden Anweisungen

```
floatVariable = 1001.333333333333333F;
Console.WriteLine("Als float: {0:G18}", floatVariable);
```

Hier weisen wir der Variablen `floatVariable` einen Wert mit vier Stellen vor dem Komma beziehungsweise dem Dezimaltrennzeichen zu. An der folgenden Ausgabe

```
Als float: 1001,33331
```

können Sie dann erkennen, wie sich dadurch auch die Genauigkeit hinter dem Komma ändert. Es werden nur noch fünf Ziffern ausgegeben – die letzte davon wieder ungenau.

Für Rechnungen, die extrem genau sein müssen, stellt Ihnen C# noch den Datentyp `decimal` zur Verfügung. Er beansprucht 128 Bits Speicherplatz und kann bis zu 28 Stellen genau darstellen. Wenn Sie mehr zu diesem Datentyp wissen wollen, lesen Sie bitte in der Hilfe nach. Sie finden die Informationen, wenn Sie nach dem Begriff `System.Decimal` suchen.

So viel zu den Datentypen für Gleitkommazahlen.

Kommen wir noch einmal kurz zurück zur **Zahlenformatzeichenfolge**. Sie können hier zum Beispiel auch festlegen, dass eine Zahl in der Form `x.xxx.xxx,xx` ausgegeben wird. Dazu verwenden Sie das Symbol `N` und geben – falls gewünscht – wieder die Genauigkeit an.

Die Anweisung

```
Console.WriteLine("Große Zahlen (1000000000) :
{0:N0}", floatVariable);
```

gibt die Zahl `10 000 000 000` zum Beispiel in der Form `10.000.000.000` auf dem Bildschirm aus. Sie können die Angabe der Genauigkeit auch weglassen. Dann werden Standardwerte benutzt – zum Beispiel zwei Nachkommastellen.

Außerdem können Sie auch die Ausrichtung und die Breite der Darstellung verändern. Die Anweisung

```
Console.WriteLine("Große Zahlen (1000000000) : {0,20:N0}",
floatVariable);
```

stellt für die Ausgabe 20 Stellen zur Verfügung und richtet den Wert rechtsbündig aus. Damit werden vor die Ausgabe des Wertes in unserem Beispiel zunächst einmal sechs Leerzeichen gesetzt.

Das .NET Framework kennt noch eine ganze Reihe weiterer Zahlenformatzeichenfolgen. Über das Symbol `E` können Sie einen Wert zum Beispiel in der Exponentialdarstellung ausgeben lassen. Genauere Beschreibungen finden Sie in der Hilfe unter dem Stichwort **Zahlenformatzeichenfolgen**. Lesen Sie die Dokumente durch und probieren Sie die verschiedenen Einstellungen einfach einmal im vorigen Code aus.

4.2.3 Datentypen für Zeichen und Zeichenketten

Für Zeichen und Zeichenketten stellt Ihnen C# die Datentypen `char` und `string` zur Verfügung. Der Datentyp `char` kann genau ein Zeichen speichern, der Typ `string` dagegen nahezu beliebig viele. Beide Datentypen arbeiten mit dem Unicode-Zeichensatz.

Der Unicode-Zeichensatz basiert auf einer 16-Bit-Darstellung und kann damit mindestens 65 536 unterschiedliche Zeichen abbilden. Weitere Informationen zu Unicode und unterschiedlichen Unicode-Codierungen finden Sie zum Beispiel im Internet unter der Adresse <https://de.wikipedia.org/wiki/Unicode>.



Bitte beachten Sie, dass ein einzelnes Zeichen durch Apostrophe '`'` umfasst wird, eine Zeichenkette dagegen durch Anführungszeichen "`"`".

Schauen wir uns die beiden Datentypen jetzt an einem Beispiel an. Erklärungen zu den Besonderheiten erhalten Sie wieder im Anschluss an den Code.

```
/* ##### Die Datentypen string und char ##### */
using System;
namespace Cshp02d_04_04
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            char charVariable;
            string stringVariable;
            //Werte zuweisen und ausgeben
            charVariable = 'A';
            //DAS KLAPPT NICHT!
            //charVariable = "A";
            stringVariable = "Ich bin eine Zeichenkette";

            Console.WriteLine("Das Zeichen: {0}", charVariable);
            Console.WriteLine("Die Zeichenkette: {0}",
            stringVariable);

            stringVariable = "Zeichenketten lassen sich ändern!";
            Console.WriteLine("Die Zeichenkette: {0}",
            stringVariable);
        }
    }
}
```

```
        stringVariable = "Auch das " + "Verketten funktioniert!";
        Console.WriteLine("Die Zeichenkette: {0}",
        stringVariable);
    }
}
```

Code 4.4: Die Datentypen string und char

In den ersten beiden Anweisungen vereinbaren wir je eine Variable vom Typ `char` und eine vom Typ `string`. Danach weisen wir den beiden Variablen Werte zu und geben sie aus. Hier müssen Sie eigentlich nur darauf achten, dass Sie beim Typ `char` die Apostrophe verwenden und beim Typ `string` die Anführungszeichen.



Noch einmal, weil es sehr wichtig ist:

Zeichenketten werden mit **Anführungszeichen** eingeschlossen. Einzelne Zeichen werden mit **Apostrophen** eingeschlossen.

Wann welches Zeichen verwendet werden muss, können Sie sich auch an der Übersetzung der Namen merken. `char` steht als Abkürzung für *character* (engl.: „Zeichen“), `string` bedeutet übersetzt so viel wie „Schnur“ oder „Band“.

Achten Sie bei einzelnen Zeichen bitte auch darauf, dass Sie den Apostroph benutzen und nicht die Akzentzeichen. Sie finden den Apostroph auf der Tastatur ganz rechts in der dritten Reihe von unten – direkt über der Umschalttaste  und neben der Taste .

Interessant ist die Anweisung

```
stringVariable = "Auch das " + "Verketten funktioniert!";
```

Hier verbinden wir zwei Zeichenketten über den Operator + und weisen das Ergebnis dann der string-Variablen zu.

Hinweis:

Das .NET Framework kennt noch zahlreiche weitere Möglichkeiten, Zeichenketten zu verändern und zu verarbeiten. Damit werden wir uns ebenfalls später im Detail beschäftigen.

Probieren Sie den vorigen Code jetzt aus. Entfernen Sie nach dem ersten Test auch einmal die Kommentarzeichen vor der Zeile

```
//charVariable = "A";
```

Sie werden sehen, die Übersetzung wird mit einer Fehlermeldung abgebrochen. Dabei spielt es auch keine Rolle, dass hier eigentlich nur ein Zeichen zugewiesen wird. Durch die Anführungszeichen interpretiert der Compiler das Zeichen in jedem Fall als Zeichenkette.

Merken Sie sich also:

Ausdrücke wie "A" werden vom Compiler immer als Zeichenkette behandelt – auch wenn es sich scheinbar nur um ein einziges Zeichen handelt.



4.2.4 Der Datentyp für logische Werte

Kommen wir nun noch zum Datentyp `bool` für logische Werte. Er kann nur zwei Werte speichern: `true` für wahr und `false` für falsch.

Hinweis:

Der Name des Datentyps geht auf den Mathematiker und Logiker George Boole zurück. Er entwickelte um 1850 eine mathematische Form der Logik – die **Aussagenlogik** oder **boolesche Algebra**. Damit werden Sie sich im weiteren Verlauf ausführlich auseinandersetzen.

Wichtig wird der Typ `bool` vor allem in Bedingungen, die wir Ihnen später vorstellen werden. Der folgende Code dient daher einfach nur als kurze Demonstration.

```
/* ######
Der Datentyp bool
##### */

using System;

namespace Cshp02d_04_05
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            bool boolVariable;

            //Werte zuweisen und ausgeben
            boolVariable = true;
            Console.WriteLine("Der Wert: {0}", boolVariable);

            boolVariable = false;
            Console.WriteLine("Der Wert: {0}", boolVariable);
        }
    }
}
```

Code 4.5: Der Datentyp `bool`

Hinweis:

Lassen Sie sich von der Ausgabe des Programms nicht verwirren. Die Konstanten für die Zuweisung auf einen `bool`-Typ lauten `true` und `false` – jeweils mit kleingeschriebenem Anfangsbuchstaben. Ausgegeben werden die Werte aber als **True** und **False** – also mit großgeschriebenem Anfangsbuchstaben.

4.3 Die Datentypen des .NET Frameworks

Neben den Datentypen von C# können Sie in einem Programm auch noch die Datentypen des .NET Frameworks verwenden. Diese Datentypen unterscheiden sich vor allem im Namen von den Datentypen des .NET Frameworks. Eine Übersicht der C#-Typen mit der jeweiligen .NET Framework-Entsprechung finden Sie in der folgenden Tabelle.

Tab. 4.2: Die Datentypen von C# und ihre .NET Framework-Entsprechungen

C#-Datentyp	.NET Framework-Entsprechung
bool	System.Boolean
char	System.Char
string	System.String
byte	System.Byte
sbyte	System.SByte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
float	System.Single
double	System.Double
decimal	System.Decimal

Ob Sie die Datentypen des .NET Frameworks oder die Datentypen von C# benutzen, macht im Prinzip keinen Unterschied. Sie können die Datentypen auch problemlos in einem Programm mischen.

Abschließend noch ein Hinweis:

Sie können bei lokalen Variablen die Angabe eines Datentyps auch durch das Schlüsselwort `var` ersetzen. C# ermittelt dann den Datentyp anhand der ersten Wertzuweisung automatisch. Diese Wertzuweisung muss aber direkt bei der Vereinbarung erfolgen. Möglich wäre damit also auch die folgende Form:

```
var test = 10;
```

Als Datentyp würde hier `int` verwendet.

Der Einsatz des Schlüsselworts `var` ist zwar sehr bequem, kann aber zu schlecht lesbarem Quelltext führen. Denn es ist ja nicht sofort klar, welchen Datentyp die Variable hat.

Das Ableiten eines Datentyps aus dem Wert wird Typinferenz oder Typableitung genannt.



So viel an dieser Stelle zu den Datentypen. Im folgenden Kapitel beschäftigen wir uns mit der Zusammenarbeit unterschiedlicher Datentypen und der Veränderung von Datentypen.

Zusammenfassung

C# stellt Ihnen Datentypen für ganze Zahlen, für Gleitkommazahlen sowie für Zeichen und Zeichenketten zur Verfügung. Außerdem können Sie über einen speziellen Datentyp auch Wahrheitswerte wie `true` oder `false` verarbeiten.

Die Datentypen für Zahlen haben unterschiedliche Wertebereiche und Genauigkeiten.

Einzelne Zeichen müssen durch Apostrophe umfasst werden, Zeichenketten dagegen durch Anführungszeichen.

Über Zahlenformatzeichenfolgen können Sie die Ausgabe formatieren.

Aufgaben zur Selbstüberprüfung

- 4.1 Was unterscheidet die Datentypen `int` und `uint`?

- 4.2 Welcher ganzzahlige Datentyp von C# hat den größten maximalen Wert?

- 4.3 Sie arbeiten mit einer Variablen vom Datentyp `int`, die den Wert 2 147 483 647 hat. Welchen Wert hat die Variable, wenn Sie 1 addieren? Warum?

- 4.4 Welchen Datentyp verwenden Sie für ein einzelnes Zeichen? Welchen Datentyp verwenden Sie für eine Zeichenkette?

- 4.5 Schreiben Sie ein kleines Programm, das den Wert 1,23456789 mit 10 Stellen Genauigkeit über eine float-Variable ausgeben soll. Welcher Wert wird tatsächlich ausgegeben? Warum?

5 Arbeiten mit Variablen unterschiedlicher Datentypen

Bisher haben wir in unseren Programmen immer mehr oder weniger sorgfältig darauf geachtet, dass der Wert, den wir einer Variablen zugewiesen haben, auch zum Datentyp der Variablen passte. Häufig werden Sie in der Praxis aber auch den Fall haben, dass Sie mit Variablen unterschiedlicher Typen arbeiten müssen – zum Beispiel, wenn Sie einen double-Typ durch einen int-Typ dividieren wollen. Was Sie dabei beachten müssen, erfahren Sie in diesem Kapitel. Außerdem zeigen wir Ihnen, wie Sie den Datentyp eines Wertes ausdrücklich verändern können.

Beginnen wollen wir aber mit einem anderen – vergleichsweise „harmlosen“ – Problem.

5.1 Die Zuweisung von float-Werten

Wie Sie ja bereits wissen, geht der Compiler bei der Zuweisung von konstanten Gleitkommawerten immer davon aus, dass es sich um den Typ double handelt. Damit der Wert als float interpretiert wird, können Sie die Zeichen f oder F hinter die Zahl stellen.

Den Unterschied sehen Sie noch einmal in den beiden folgenden Zeilen

```
//Interpretation als double
variable = 1001.33;
//Interpretation als float durch das nachgestellte F
variable = 1001.33F;
```

5.2 Zuweisungen zwischen Variablen unterschiedlichen Typs

Schauen wir uns jetzt an, was bei der Zuweisung von Werten zwischen Variablen mit unterschiedlichen Typen geschieht. Fangen wir hierzu mit einem einfachen Beispiel an – der direkten Zuweisung.

Diese direkte Zuweisung ist nur dann möglich, wenn der Zieldatentyp „größer“ ist als der ursprüngliche Datentyp. Sie können also einer double-Variablen ohne Weiteres eine int-Variablen zuweisen oder einer int-Variablen eine byte-Variablen. Der Compiler formt dabei den Datentyp automatisch in das passende Format um.

Die andere Richtung – also die Zuweisung eines „größeren“ Typs auf einen „kleineren“ – funktioniert dagegen auf direktem Wege nicht. Hier meldet der Compiler einen Fehler, dass er den Typ nicht konvertieren kann. Der Grund für diesen Fehler ist auch schnell gefunden. Denn bei der Zuweisung eines „größeren“ Typs auf einen kleineren müssen zwangsläufig Informationen verloren gehen, da die Wertebereiche nicht passen. Wenn Sie beispielsweise einen double-Typ auf einen float-Typ zuweisen, wird nicht nur der darstellbare Zahlenraum kleiner, sondern auch die Genauigkeit geringer. Und dadurch können sehr unangenehme Probleme entstehen.

In dem folgenden Fragment sind daher auch nur die ersten Zuweisungen möglich. Die letzten drei Zuweisungen dagegen führen zu einer Fehlermeldung, da hier der Zieldatentyp „kleiner“ ist als der ursprüngliche Datentyp.

```

static void Main(string[] args)
{
    long longZahl = 1234567891234567;
    int intZahl = 100;
    double doubleZahl = 1.2;
    byte byteZahl = 10;
    //diese Zuweisungen sind möglich
    longZahl = intZahl;
    doubleZahl = intZahl;
    doubleZahl = longZahl;
    longZahl = byteZahl;
    //diese dagegen nicht
    intZahl = longZahl;
    intZahl = doubleZahl;
    byteZahl = intZahl;
}

```

Code 5.1: Zuweisungen zwischen Variablen unterschiedlichen Typs
(es handelt sich um ein Fragment, das sich nicht übersetzen lässt)

5.3 Typecasting

Bei der Arbeit mit unterschiedlichen Datentypen gibt es noch ein weiteres Problem, das sehr unangenehm werden kann. Schauen wir uns dazu einmal den folgenden Code an. Es sollen mehrere Divisionen mit unterschiedlichen Typen durchgeführt werden.

```

/* ##### Divisionen mit verschiedenen Datentypen #####
using System;

namespace Cshp02d_05_02
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            int intVariable1, intVariable2;
            double doubleVariable1, doubleVariable2, doubleVariable3;

            //Wertzuweisungen
            intVariable1 = 10;
            doubleVariable1 = 10;
            intVariable2 = 3;
            doubleVariable2 = 3;

            //double / double
            doubleVariable3 = doubleVariable1 / doubleVariable2;
            Console.WriteLine("double / double =
{0}", doubleVariable3);
            //int / double
            doubleVariable3 = intVariable1 / doubleVariable2;
        }
    }
}

```

```
        Console.WriteLine("int / double = {0}",doubleVariable3);
        //double / int
        doubleVariable3 = doubleVariable1 / intVariable2;
        Console.WriteLine("double / int = {0}",doubleVariable3);
        //int / int
        doubleVariable3 = intVariable1 / intVariable2;
        Console.WriteLine("int / int = {0}",doubleVariable3);
    }
}
```

Code 5.2: Divisionen mit verschiedenen Datentypen

In dem Code werden zunächst fünf Variablen vereinbart – zwei vom Typ `int` und drei vom Typ `double`. Danach werden dann jeweils zwei Variablen vom Typ `double` und `int` die Werte 10 und 3 zugewiesen. Anschließend werden diese Werte mehrfach dividiert, dabei jeweils der `double`-Variablen `doubleVariable3` zugewiesen und ausgegeben.

Auf den ersten Blick sieht der Code ganz gut aus. Wenn Sie ihn aber ausführen, werden Sie eine Überraschung erleben. Die Ausgabe sieht nämlich so aus:

```
double / double = 3,33333333333333  
int / double = 3,33333333333333  
double / int = 3,33333333333333  
int / int = 3
```

Bei den ersten drei Divisionen ist das Ergebnis korrekt. Dabei spielt es auch keine Rolle, dass wir in der zweiten und der dritten Division mit zwei unterschiedlichen Typen arbeiten. Der Compiler baut den `int`-Typ offensichtlich intern in einen `double`-Typ um und benutzt dann diesen Typ für die Division.

Bei Rechenoperationen mit unterschiedlichen Typen versucht der Compiler selbst, alle Typen auf den jeweils größten beziehungsweise genauesten Typ umzubauen.

Das Ergebnis der letzten Division dagegen ist falsch. Denn hier werden zwei `int`-Typen dividiert. Damit hat auch das Ergebnis den Typ `int` – egal, welchen Typ die Variable hat, der das Ergebnis zugewiesen wird. Denn es wird erst gerechnet und dann zugewiesen. Da aber bereits bei der Rechnung alle Nachkommastellen verloren gehen, hat die folgende Zuweisung auf einen Typ `double` keine Wirkung mehr. Die Nachkommastellen sind abgeschnitten und das Ergebnis damit falsch. Daher wird bei der letzten Ausgabe auch nur der Wert 3 ausgegeben.

Bei Rechenoperationen wird der Typ des Ergebnisses durch die Typen der Operanden bestimmt – und nicht durch den Typ der Variable, die das Ergebnis aufnimmt.

Eine Möglichkeit, diesem Problem aus dem Weg zu gehen, wäre es, statt der ganzzahligen Typen einen `double`- oder `float`-Typ zu benutzen. Das ist aber in der Praxis nicht immer möglich. Trotzdem müssen Sie nicht damit leben, dass Ihr Computer scheinbar

nicht korrekt rechnen kann. Denn Sie können den Compiler anweisen, den Typ eines Ausdrucks oder einer Variablen zu verändern. Dazu benutzen Sie das **explizite Typecasting** – häufig auch kurz einfach nur Casting⁵ genannt.

Das Casting erfolgt durch die Angabe des Zieltyps in runden Klammern vor dem Ausdruck.



Beispiel 5.1:

```
(double) intVariable1
```

würde den Wert der Variablen `intVariable1` in den Typ `double` umwandeln. Der eigentliche Datentyp der Variablen bleibt dabei aber unverändert. Die Umwandlung gilt also nur für die Stelle, an der das Casting erfolgt.

Sehen wir uns das Typecasting in einem Beispielprogramm an. Im folgenden Code casten wir einige Variablen und auch komplette Ausdrücke. Was dabei genau passiert, erklären wir Ihnen wieder nach dem Code.

```
/* ##### Typecasting #####
#####
using System;

namespace Cshp02d_05_03
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            int intVariable1, intVariable2;
            double doubleVariable1, doubleVariable2, doubleVariable3;

            //Wertzuweisungen
            intVariable1 = 10;
            doubleVariable1 = 10;
            intVariable2 = 3;
            doubleVariable2 = 3;

            //beide double werden zum int, das Ergebnis wird falsch
            //bitte jeweils in einer Zeile eingeben
            doubleVariable3 = (int)doubleVariable1 /
                (int)doubleVariable2;
            Console.WriteLine("(int)double / (int)double =
{0}", doubleVariable3);
            //ein int wird zum double, das Ergebnis stimmt
            doubleVariable3 = (double)intVariable1 / intVariable2;
            Console.WriteLine("(double)int / int = {0}",
            doubleVariable3);
        }
    }
}
```

5. Wörtlich übersetzt bedeutet *cast* so viel wie „Guss“ oder „Gussform“. Gemeint ist damit, dass der Typ in eine bestimmte Form gezwungen wird.

```
//noch einmal zur Kontrolle ohne Casting, das
//Ergebnis ist wieder falsch
doubleVariable3 = intVariable1 / intVariable2;
Console.WriteLine("int / int = {0}", doubleVariable3);
//der gesamte Ausdruck wird gecastet
doubleVariable3 = (double)(intVariable1 / intVariable2);
//bitte in einer Zeile eingeben
Console.WriteLine("(double)(int / int) = {0}",
doubleVariable3);
}
}
```

Code 5.3: Typecasting

Das Programm erzeugt die folgenden Ausgaben:

```
(int)double / (int)double = 3  
(double)int / int = 3,33333333333333  
int / int = 3  
(double)(int / int) = 3
```

In der ersten Zeile wird das Ergebnis der Berechnung

```
doubleVariable3 = (int)doubleVariable1 / (int)doubleVariable2;
```

angezeigt. Es ist falsch, weil wir hier beide `double`-Typen in `int`-Typen umwandeln. Damit werden für die Rechnung ganze Zahlen benutzt und im Ergebnis fehlen die Nachkommastellen.

Die zweite Zeile dagegen zeigt das korrekte Ergebnis an. Hier wird einer der beiden int-Typen in einen double-Typ umgewandelt. Das führt dazu, dass auch der andere Typ automatisch umgewandelt wird.

Die dritte Zeile dagegen zeigt wieder das falsche Ergebnis an. Denn hier verwenden wir beide `int`-Typen wieder ohne Casting. Sie sehen also, das Casting wirkt nur exakt an der Stelle, an der Sie es verwenden. Der eigentliche Typ wird nicht verändert.

Die vierte Zeile der Ausgabe lohnt ein etwas genaueres Hinsehen. Hier wird das Ergebnis der Anweisung

```
doubleVariable3 = (double) (intVariable1 / intVariable2);
```

ausgegeben. Das Casting (double) (intVariable1 / intVariable2) sieht vielleicht auf den ersten Blick ganz brauchbar aus, liefert aber trotzdem nur ein Ergebnis ohne Nachkommastellen. Denn umgewandelt werden hier nicht die beiden Variablen in dem Ausdruck oder der gesamte Ausdruck, sondern nur das **Ergebnis** des Ausdrucks. Es wird also erst gerechnet und dann umgewandelt. Da aber bei der Division der beiden ganzzahligen Typen die Nachkommastellen verloren gehen, hat die anschließende Umwandlung keinerlei Effekt.

Ein Casting ist auch dann zwingend erforderlich, wenn Sie mit den Typen `byte` und `short` rechnen und das Ergebnis wieder einem dieser Typen zuweisen wollen. Denn die Typen `byte` und `short` werden vom Compiler immer automatisch auf den Typ `int` umgebaut – und damit scheitert dann die Zuweisung auf die Typen `byte` und

short, da das Ergebnis ja zu groß ist. Sie müssen also das Ergebnis des gesamten Ausdrucks explizit in den Typ byte beziehungsweise short umbauen – so wie in dem folgenden Fragment:

```
static void Main(string[] args)
{
    byte byteZahl1, byteZahl2, byteZahl3;
    byteZahl2 = 10;
    byteZahl3 = 20;
    //das geht nicht!!!
    byteZahl1 = byteZahl2 + byteZahl3;
    //und das auch nicht
    byteZahl1 = (byte)byteZahl2 + (byte)byteZahl3;
    //so ist es möglich
    byteZahl1 = 10 + 20;
    //und auch über ein Casting des gesamten Ausdrucks
    byteZahl1 = (byte)(byteZahl2 + byteZahl3);
}
```

Code 5.4: Casting bei Berechnungen mit den Typen byte und short
(es handelt sich um ein Fragment, das sich nicht übersetzen lässt)

Auch bei der Arbeit mit long-Werten können unangenehme Überraschungen auftreten. So führt zum Beispiel die folgende Anweisung zu einer Fehlermeldung des Compilers – auch dann, wenn es sich bei der Variablen grosseZahl um einen long-Typ handelt.

```
grosseZahl = 2147483647 + 1;
```

Denn die beiden Zahlen werden vom Compiler als int-Werte interpretiert. Und damit findet durch die Addition ein Überlauf statt, obwohl die Zuweisung auf den richtigen Typ long erfolgt. Erst nach einem Casting oder der ausdrücklichen Markierung als long durch den Buchstaben L oder l hinter einem der Werte funktioniert die Zuweisung. Korrekt wären also zum Beispiel:

```
grosseZahl = 2147483647 + 1L;
```

oder

```
grosseZahl = (long)2147483647 + 1;
```

Tipp:

Der Buchstabe l wird beim flüchtigen Lesen schnell mit der Ziffer 1 verwechselt. Sie sollten daher für die Kennzeichnung als long den großen Buchstaben L benutzen.

Allerdings hat das Casting auch seine Grenzen. So gehen zum Beispiel bei der Umwandlung einer Gleitkommazahl in eine ganze Zahl die Nachkommastellen verloren, da der Zieldatentyp sie ja nicht abbilden kann. Außerdem kann es durchaus passieren, dass sich durch das Casting der Wert verändert – nämlich dann, wenn Sie einen Wert umwandeln, der für den Zieldatentyp zu groß ist. Sie sehen, hier drohen genau die Probleme, die der Compiler bei einer direkten Zuweisung verhindert.

Außerdem können Sie auch keine Zeichenkette in eine Zahl umwandeln oder andersherum. Dazu müssen Sie spezielle Methoden verwenden – zum Beispiel Convert.ToInt32(). Diese Methode konvertiert den Ausdruck, der in Klammern angegeben

wird, in einen `Int32`-Typ beziehungsweise in einen `int`-Typ. Damit können Sie dann auch mit Zeichenketten „richtig“ rechnen. Eine Demonstration finden Sie im folgenden Code:

```
/*
#####
Umwandlung über Convert
#####
*/

using System;

namespace Cshp02d_05_05
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            int intVariable;
            string stringVariable1, stringVariable2;

            //Wertzuweisungen
            stringVariable1 = "10";
            stringVariable2 = "20";

            //SO GEHT ES NICHT!
            //intVariable = (int)(stringVariable1) +
            //((int)(stringVariable2));
            //aber so!
            //bitte in einer Zeile eingeben
            intVariable = Convert.ToInt32(stringVariable1) +
            Convert.ToInt32(stringVariable2);
            Console.WriteLine("Das Ergebnis = {0}", intVariable);
        }
    }
}
```

Code 5.5: Die Konvertierung von Zeichenketten in Zahlen

Bitte beachten Sie:

Die Methoden der Klasse `Convert` haben ihren Namen von den .NET Framework-Typen. Die Methode heißt also `Convert.ToInt32()` und nicht `Convert.ToInt()`.



Jetzt können Sie natürlich fragen: „Warum der ganze Aufwand? Wenn die beiden Zeichenketten direkt als Zahlen vereinbart werden, brauche ich das doch alles nicht.“ Im Moment haben Sie sicherlich Recht. Aber wie Sie im nächsten Kapitel noch sehen werden, ist die Umwandlung von Zeichenketten in Zahlen durchaus wichtig.

So viel zum Arbeiten mit Variablen unterschiedlicher Datentypen. Im nächsten Kapitel werden wir uns um die Eingabe kümmern.

Zusammenfassung

Wenn Sie in einem Ausdruck Variablen mit einem unterschiedlichen Typ verwenden, benutzt der Compiler immer den größeren beziehungsweise den genaueren der eingesetzten Typen.

Mit dem expliziten Typecasting können Sie den Datentyp einer Variablen oder eines Ausdrucks gezielt verändern.

Aufgaben zur Selbstüberprüfung

- 5.1 Welche Ergebnisse liefern die folgenden Ausdrücke, wenn `doubleVariable` vom Typ `double` ist? Warum?

```
doubleVariable = 100 / 3;  
doubleVariable = 100 / 3.0;
```

- 5.2 Sie haben einer Variablen vom Typ `double` den Wert `1.4444444444444444` zugewiesen. Was geschieht, wenn Sie diesen Wert einer Variablen vom Typ `float` zuweisen wollen? Begründen Sie bitte kurz Ihre Antwort.

- 5.3 Liefert der folgende Ausdruck ein Ergebnis mit Nachkommastellen? Bei den beiden Variablen handelt es sich um `int`-Typen. Begründen Sie bitte Ihre Antwort.

```
(double) (intVariable1 / intVariable2);
```

6 Eingabe

Bisher haben wir in unseren Programmen immer mit Werten gearbeitet, die im Quelltext festgelegt wurden. Eine direkte Veränderung während der Programm ausführung war nicht möglich. Sie mussten immer erst den Quelltext überarbeiten und dann das Programm kompilieren. Solche „Einbahnstraßen“-Programme sind natürlich in vielen Fällen nicht brauchbar. Deshalb zeigen wir Ihnen in diesem Kapitel, wie Sie Daten mit Eingabeanweisungen einlesen können.

Das .NET Framework kennt für Konsolenanwendungen insgesamt drei Eingabeanweisungen:

- `Console.Read()`,
- `Console.ReadKey()` und
- `Console.ReadLine()`.

Die Anweisungen `Console.Read()` und `Console.ReadKey()` lesen jeweils ein einziges Zeichen ein. Der einzige Unterschied liegt darin, dass Sie bei `Console.Read()` die Eingabe noch mit der Eingabetaste abschließen müssen, bei `Console.ReadKey()` dagegen nicht.

Die Anweisung `Console.ReadLine()` kann auch längere Eingaben verarbeiten, die aus mehreren Zeichen bestehen. Genau wie bei `Console.Read()` muss auch hier die Eingabe mit der Eingabetaste beendet werden.

Die Unterschiede zwischen den drei Anweisungen können Sie sich auch recht einfach an den Übersetzungen merken: `ReadKey` bedeutet frei übersetzt so viel wie „Lese Taste“, `Read` steht für „Lese“ und `ReadLine` für „Lese Zeile“.

Allerdings sind alle drei Eingabeanweisungen im praktischen Einsatz etwas „sperrig“ und führen auch schnell zu Programmabstürzen. Das liegt vor allem an den Datentypen, die die drei Methoden zurückliefern.

Die Methode `Console.ReadKey()` liefert die gedrückte Taste als einen Typ `System.ConsoleKeyInfo` zurück – allerdings nicht direkt, sondern in der Eigenschaft `KeyChar` von `ConsoleKeyInfo`.

Die Methode `Console.Read()` liefert nicht – wie vielleicht zu erwarten – ein Zeichen, sondern einen `Int32`-Wert zurück und die Methode `Console.ReadLine()` schließlich liefert eine Zeichenkette vom Typ `string`.

Was auf den ersten Blick nicht weiter problematisch aussieht, kann in der Praxis für recht seltsame Phänomene sorgen. Schauen wir uns das einmal am folgenden Code an. Hier werden einige Daten über die Anweisungen `Console.Read()` und `Console.ReadLine()` verarbeitet.

```
/* #####
Eingabe I
#####
using System;
```

```
namespace Cshp02d_06_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            int eingabel;
            string eingabe2, eingabe3;

            //ein Zeichen über Read() lesen und ausgeben
            //bitte in einer Zeile eingeben
            Console.WriteLine("Geben Sie ein Zeichen ein.
Drücken Sie dann die Eingabetaste.");
            eingabel = Console.Read();

            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben das Zeichen {0}
eingegeben.", eingabel);
            //die Eingabetaste aus dem Puffer holen
            //sonst wird die nächste Eingabe scheinbar
            //übersprungen
            Console.ReadLine();

            //Zahlen einlesen
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie bitte die erste Zahl
ein. Drücken Sie dann die Eingabetaste. ");
            eingabe2 = Console.ReadLine();
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie bitte die zweite Zahl ein.
Drücken Sie dann die Eingabetaste. ");
            eingabe3 = Console.ReadLine();
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Addition der Zahlen {0}
und {1} ergibt = {2}", eingabe2, eingabe3,
eingabe2 + eingabe3);

            //Zeichenketten einlesen
            Console.Write("Wie heißen Sie? ");
            eingabe2 = Console.ReadLine();
            Console.Write("Wen wollen Sie grüßen? ");
            eingabe3 = Console.ReadLine();
            //bitte in einer Zeile eingeben
            Console.WriteLine("Hallo {0}. Es grüßt Dich
{1}.", eingabe3, eingabe2);
        }
    }
}
```

Code 6.1: Eingaben über Console.Read() und Console.ReadLine()

Hinweise:

Die Anweisung

```
Console.ReadLine();
```

nach dem ersten Einlesen ist erforderlich, weil die Eingabetaste, die für das Abschließen der ersten Eingabe erforderlich ist, von der Methode `Console.Read()` nicht verarbeitet wird und noch weiter im Tastaturpuffer steht. Ohne diese Anweisung würde die Eingabetaste direkt von den folgenden Eingabeanweisungen verarbeitet und es entstünde der Eindruck, dass das Einlesen schlicht und einfach übersprungen würde.

Mit der Methode `Console.ReadKey()` können Sie ein Konsolenprogramm, das Sie durch einen Doppelklick im Explorer von Windows gestartet haben, so lange geöffnet halten, bis der Anwender eine Taste drückt. Allerdings muss dann beim Ausführen über die Eingabeaufforderung oder beim Start direkt aus der Entwicklungs-Umgebung zweimal eine beliebige Taste gedrückt werden, um das Programm zu beenden.

Der Programmablauf könnte so aussehen:

```
Geben Sie ein Zeichen ein. Drücken Sie dann die Eingabetaste.  
q  
Sie haben das Zeichen 113 eingegeben.  
Geben Sie bitte die erste Zahl ein. Drücken Sie dann die Eingabetaste.12  
Geben Sie bitte die zweite Zahl ein. Drücken Sie dann die Eingabetaste. 18  
Die Addition der Zahlen 12 und 18 ergibt = 1218  
Wie heißen Sie? Christoph  
Wen wollen Sie grüßen? Welt  
Hallo Welt. Es grüßt Dich Christoph.
```

Das Einlesen funktioniert problemlos, allerdings erzeugen die ersten beiden Ausgaben nichts Brauchbares. Statt des Zeichens wird bei der ersten Ausgabe der numerische Code des Zeichens ausgegeben und bei der zweiten Ausgabe werden die beiden Werte aneinandergehängt. Lediglich das Einlesen und Ausgeben der Zeichenketten ganz am Ende des Programms funktionieren wie gewünscht.

Die Gründe für die ersten beiden „seltsamen“ Ausgaben sind schnell gefunden – wenn Sie sich noch einmal die Datentypen der beiden Methoden in Erinnerung rufen. Die Methode `Console.Read()` liefert einen `Int32`-Typ zurück, der dann auch als Zahl ausgegeben wird. Die Methode `Console.ReadLine()` dagegen liefert Zeichenketten vom Typ `string` zurück. Diese Zeichenketten können Sie zwar „addieren“, dabei werden aber die Werte einfach aneinandergehängt.

Um diese „seltsamen“ Ausgaben zu vermeiden, müssen Sie die eingelesenen Daten konvertieren – zum Beispiel mit `Convert.ToInt32()` oder `Convert.ToChar()`.


Bitte denken Sie daran:

Die Namen der Methoden werden von den .NET Framework-Typen abgeleitet. Die Methode heißt `Convert.ToChar()` und nicht `Convert.Tochar()`.

Das könnte dann zum Beispiel so aussehen:

```
/* ##### Eingabe II (mit Konvertierung) #####
using System;
namespace Cshp02d_06_02
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            char eingabel;
            int eingabe2, eingabe3;

            //ein Zeichen lesen und ausgeben, jetzt über
            //ReadLine() mit Konvertierung
            //bitte in einer Zeile eingeben
            Console.WriteLine("Geben Sie ein Zeichen ein.");
            Drücken Sie dann die Eingabetaste.");
            eingabel = Convert.ToChar(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben das Zeichen {0}
eingegeben.", eingabel);

            //Zahlen einlesen
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie bitte die erste Zahl ein. Drücken
Sie dann die Eingabetaste. ");
            eingabe2 = Convert.ToInt32(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie bitte die zweite Zahl
ein. Drücken Sie dann die Eingabetaste. ");
            eingabe3 = Convert.ToInt32(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Addition der Zahlen {0}
und {1} ergibt = {2}", eingabe2, eingabe3,
eingabe2 + eingabe3);
        }
    }
}
```

Code 6.2: Einlesen mit Konvertierung

Den `int`-Typ für das erste Einlesen haben wir durch einen `char`-Typ ersetzt und die beiden `string`-Typen für die Berechnungen durch `int`-Typen.

Das Zeichen lassen wir dann über `Console.ReadLine()` einlesen. Damit ersparen wir uns das „Abholen“ der Eingabetaste aus dem Tastaturpuffer, das im letzten Code noch erforderlich war. Da die Methode `Console.ReadLine()` aber einen `string`-Typ liefert, konvertieren wir das Ergebnis der Methode durch den Ausdruck `Convert.ToChar(Console.ReadLine())`.

Eine ähnliche Technik setzen wir dann beim Einlesen der beiden Zahlen ein. Hier konvertieren wir die beiden Eingaben durch `Convert.ToInt32(Console.ReadLine())` in einen `Int32`-Typ, mit dem wir dann auch rechnen können.

Hinweis:

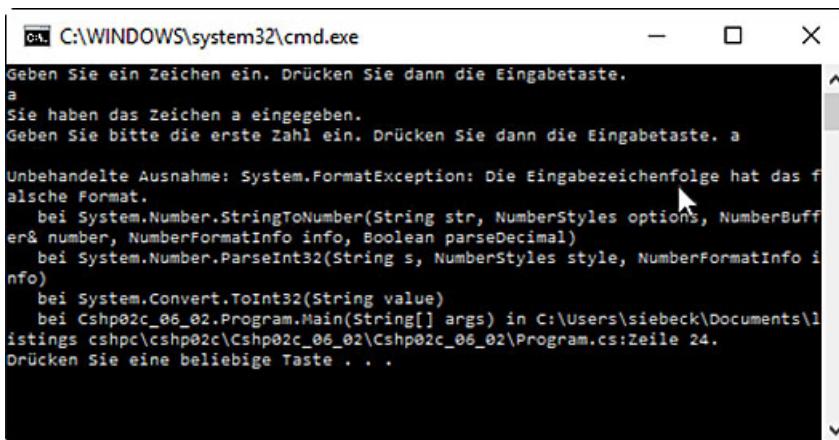
Sie können die Eingaben auch in weitere Datentypen konvertieren – zum Beispiel in den Typ `Double` beziehungsweise `double` oder den Typ `Single` beziehungsweise `float`. Dazu verwenden Sie die Anweisungen `Convert.ToDouble()` beziehungsweise `Convert.ToSingle()`.

Bitte beachten Sie aber, dass bei der Eingabe von Nachkommastellen das Komma als Trennzeichen erwartet wird und nicht etwa der Punkt.

Das klappt auch alles wunderbar – wenn der Anwender nicht aus Versehen zum Beispiel bei einer der letzten beiden Eingaben ein Zeichen eingibt oder bei einer Eingabe einfach nur die Eingabetaste drückt. Dann nämlich stürzt das Programm ab, weil die Konvertierung nicht mehr funktioniert.

Probieren Sie das einfach einmal aus. Speichern Sie zur Sicherheit noch einmal alle Änderungen. Lassen Sie dann den vorigen Code ausführen und geben Sie zum Beispiel bei der zweiten Eingabe einen Buchstaben ein. Sobald Sie die Eingabetaste drücken, nimmt das Unglück seinen Lauf.

Das Programm wird beendet. In der Eingabeaufforderung sehen Sie auch, wo das Problem aufgetreten ist.



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The window contains the following text:
Geben Sie ein Zeichen ein. Drücken Sie dann die Eingabetaste.
a
Sie haben das Zeichen a eingegeben.
Geben Sie bitte die erste Zahl ein. Drücken Sie dann die Eingabetaste. a
Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das falsche Format.
bei System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
bei System.Convert.ToInt32(String value)
bei Cshp02c_06_02.Program.Main(String[] args) in C:\Users\siebeck\Documents\1
istings\cshp02c\cshp02c\cshp02c_06_02\Cshp02c_06_02\Program.cs:Zeile 24.
Drücken Sie eine beliebige Taste . . .

Abb. 6.1: Die Meldung in der Eingabeaufforderung (interessant ist vor allem der Text am Mauszeiger)

In der ersten Zeile hinter der Eingabe teilt Ihnen Visual Studio mit, dass die Eingabezeichenfolge das falsche Format hat.

Im Moment können wir daran auch nichts ändern. Sie müssen sich also erst einmal damit abfinden, dass falsche Eingaben beim Konvertieren zum Programmabsturz führen können. Im weiteren Verlauf werden wir uns aber bei den Schleifen – der Wiederholung von Anweisungen – noch einmal um das Problem kümmern.

Zusammenfassung

Das .NET Framework kennt drei unterschiedliche Eingabeanweisungen für die Eingabeaufforderung:

- `Console.Read()`,
- `Console.ReadKey()` und
- `Console.ReadLine()`.

Die Eingabeanweisungen unterscheiden sich in der Anzahl der gelesenen Zeichen und im Datentyp, der zurückgegeben wird.

Aufgaben zur Selbstüberprüfung

- 6.1 Welcher wesentliche Unterschied besteht zwischen den Eingabeanweisungen `Console.ReadKey()` und `Console.Read()`?

- 6.2 Welche Eingabeanweisung müssen Sie verwenden, wenn Sie mehrere Zeichen gleichzeitig einlesen wollen?

- 6.3 Schreiben Sie ein Programm, das drei Zahlen über die Tastatur einliest. Die erste Eingabe soll vom Typ `int` sein, die zweite und dritte Eingabe vom Typ `double`. Addieren Sie dann die drei Eingaben und lassen Sie das Ergebnis auf dem Bildschirm ausgeben.

7 Die Gestaltung von Konsolenanwendungen

Zum Abschluss dieses Studienheftes wollen wir Ihnen noch kurz zeigen, wie Sie Ihre Konsolenanwendungen etwas „netter“ gestalten können.

In der Standardeinstellung wird die Eingabeaufforderung von Windows mit schwarzem Hintergrund und weißer Schrift dargestellt. Über die Anweisungen `Console.ForegroundColor` und `Console.BackgroundColor`⁶ können Sie aber auch andere Farben einstellen. Die Angabe der Farbe erfolgt dabei als Konstante in der Form `ConsoleColor.<Name>`. Die Anweisung

```
Console.ForegroundColor = ConsoleColor.Blue;
```

setzt zum Beispiel die Textfarbe in der Eingabeaufforderung auf Blau.

Hinweis:

Änderungen der Hintergrundfarbe werden erst dann für das gesamte Fenster durchgeführt, wenn Sie den Fensterinhalt einmal mit der Anweisung `Console.Clear()` löschen.

Neben dem Ändern der Farbe können Sie aber auch den Text in der Titelleiste des Konsolenfensters verändern, die Position des Fensters festlegen und noch einiges Weiteres mehr. Im praktischen Einsatz finden Sie einige Anweisungen im folgenden Code. Weitere Anweisungen schlagen Sie bitte in der Hilfe nach. Das entsprechende Dokument finden Sie schnell, wenn Sie die Einfügemarke auf den Text `Console.` in einem Code stellen und dann die Taste **F1** drücken.

```
/* ##### Gestaltung von Konsolenprogrammen ##### */
using System;
namespace Cshp02d_07_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung der Variablen
            string eingabe1, eingabe2;

            //den Titel setzen
            Console.Title = "Nun sieht es netter aus!";

            //die Farben verändern
            Console.BackgroundColor = ConsoleColor.Blue;
            Console.ForegroundColor = ConsoleColor.Yellow;
        }
    }
}
```

6. *Foreground color* bedeutet übersetzt so viel wie „Vordergrundfarbe“ und *background color* „Hintergrundfarbe“.

```
//alles löschen, damit die Änderungen wirksam werden  
Console.Clear();  
  
//Zeichenketten einlesen  
Console.Write("Wie heißen Sie? ");  
eingabe1 = Console.ReadLine();  
Console.Write("Wen wollen Sie grüßen? ");  
eingabe2 = Console.ReadLine();  
//bitte in einer Zeile eingeben  
Console.WriteLine("Hallo {0}. Es grüßt Dich  
{1}.", eingabe2, eingabe1);  
}  
}  
}
```

Code 7.1: Gestaltung von Konsolenprogrammen

„Spielen“ Sie einfach einmal ein wenig zur Entspannung mit den verschiedenen Möglichkeiten herum. Sie haben es sich nach dem Durcharbeiten dieses Studienheftes verdient.

Schlussbetrachtung

Herzlichen Glückwunsch!

Sie haben erfolgreich die ersten Schritte beim Erstellen von C#-Programmen unternommen. Sie können jetzt Werte über die Tastatur einlesen, Rechenoperationen mit diesen Werten ausführen und die Ergebnisse wieder auf dem Bildschirm ausgeben.

Noch ein Tipp: Nutzen Sie die Codes aus diesem Studienheft ruhig auch für eigene Experimente. Probieren Sie aus, welche Wirkung Änderungen am Quelltext haben. Denken Sie aber in jedem Fall daran, Ihre Quelltexte vor dem Kompilieren zur Sicherheit zu speichern.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 In jedem C#-Programm muss die Methode `Main()` enthalten sein. Sie bildet den Startpunkt für die Ausführung des Programms.
- 1.2 Der Anweisungsteil einer Methode muss durch `{` und `}` geklammert werden. Die einzelnen Anweisungen werden durch `;` getrennt.
- 1.3 Die beiden Möglichkeiten sind die Kommentarzeichen `//` und `/* */`. Mit den Zeichen `//` wird ein Kommentar in einer Zeile eingefügt. Die Zeichen `/* */` werden benutzt, um einen Kommentarblock einzufügen.

Kapitel 2

- 2.1 a) `\n`
b) `\t`
c) `\\"\\`
- 2.2 Die Lösung könnte so aussehen:

```
/*
#####
Musterlösung Aufgabe II.2
#####
using System;

namespace LoesII2
{
    class Program
    {
        static void Main(string[] args)
        {
            //bitte jeweils in einer Zeile eingeben
            Console.Write("Dies ist ein C#- Programm,\n");
            Console.Write("das mithilfe von Escape-Sequenzen
sogar Zeichen\n");
            Console.Write("wie \" \" \" , \" \\ \" und
\" \' \", \n");
            Console.Write("die in C# eine besondere Bedeutung
haben, ausgibt.\n");
        }
    }
}
```

Wichtig sind vor allem die korrekte Position der Escape-Sequenz `\n` und die Ausgabe der besonderen Zeichen über die entsprechenden Escape-Sequenzen. Das Zeichen `'` kann in dem Beispiel auch ohne `\` dargestellt werden, weil es innerhalb der Zeichenkette steht.

Kapitel 3

- 3.1 Das Ergebnis ist 54. Das Programm kann zum Beispiel so aussehen:

```
/* ##### Musterlösung Aufgabe III.1 #####
using System;
namespace LoesIII
{
    class Program
    {
        static void Main(string[] args)
        {
            //bitte in einer Zeile eingeben
            Console.WriteLine("Das Ergebnis ist: {0}", (((20 * 4)
                %3 + 8) *10 + 8) / 2);
        }
    }
}
```

- 3.2
- ungültig, das erste Zeichen darf keine Ziffer sein.
 - gültig
 - gültig
 - gültig
 - ungültig, `using` ist ein Schlüsselwort und kann nicht verwendet werden
 - ungültig, das Zeichen `*` darf nicht verwendet werden
 - ungültig, eine Leerstelle darf nicht verwendet werden

- 3.3 Der Operator heißt Zuweisungsoperator und wird mit dem Zeichen `=` dargestellt.

- 3.4 Das Schlüsselwort heißt `const`.

- 3.5

Schritt	Ausgewertete Operatoren	Ausgewerteter Ausdruck
1	<code>++, --</code>	<code>++a, --b</code>
2	<code>*</code>	<code>++a * --b</code>
3	<code>/</code>	<code>++a * --b / c</code>
4	<code>+</code>	<code>++a * --b / c + d</code>
5	<code>=</code>	<code>= ++a * --b / c + d</code>

Kapitel 4

- 4.1 Der Datentyp `int` kann negative Zahlen darstellen, der Datentyp `uint` nicht.
- 4.2 Den größten maximalen Wert bei den ganzzahligen Datentypen kann `ulong` darstellen.
- 4.3 Das Ergebnis ist $-2\ 147\ 483\ 648$. Der Wert $2\ 147\ 483\ 647$ ist der maximale Wert für `int`. Bei der Addition von 1 erfolgt ein Überlauf und es wird mit dem kleinsten möglichen Wert weitergearbeitet.
- 4.4 Der Datentyp für einzelne Zeichen heißt `char`, der Datentyp für Zeichenketten `string`.
- 4.5 Das Programm könnte so aussehen:

```
/* ##### Musterlösung Aufgabe IV.5 #####
using System;

namespace LoesIV5
{
    class Program
    {
        static void Main(string[] args)
        {
            float floatVar = 1.23456789F;
            Console.WriteLine("floatVar ist: {0:G10}", floatVar);
        }
    }
}
```

Bei der Ausgabe sind die beiden letzten Stellen zufällig, da der Typ `float` nicht über eine Genauigkeit von 10 Stellen verfügt.

Kapitel 5

- 5.1 Der erste Ausdruck liefert 33 als Ergebnis. Die Nachkommastellen werden ignoriert, da es sich um zwei ganze Zahlen handelt. Die Zuweisung auf die Variable vom Typ `double` spielt keine Rolle.

Der zweite Ausdruck liefert 33,333333333333 als Ergebnis. Durch die Schreibweise 3.0 wird der zweite Wert als `double` behandelt. Dadurch wird auch der erste Wert vom Compiler umgewandelt und der gesamte Ausdruck ist vom Typ `double`.

- 5.2 Die Zuweisung wird vom Compiler verhindert, da Informationen verloren gehen könnten.
- 5.3 Nein, der Ausdruck liefert kein Ergebnis mit Nachkommastellen. Es wird erst gerechnet und dann umgewandelt. Die Nachkommastellen gehen aber bereits bei der Berechnung verloren.

Kapitel 6

- 6.1 Die Eingabe bei `Console.ReadKey()` muss nicht mit der Eingabetaste abgeschlossen werden.
- 6.2 Für mehrere Zeichen müssen Sie die Eingabeanweisung `Console.ReadLine()` verwenden.
- 6.3 Das Programm könnte so aussehen:

```
/* ##### Musterlösung Aufgabe VI.3 #####
using System;
namespace LoesVI3
{
    class Program
    {
        static void Main(string[] args)
        {
            int intVariable1;
            double doubleVariable1, doubleVariable2;
            Console.Write("Geben Sie die erste Zahl ein: ");
            intVariable1 = Convert.ToInt32 (Console.ReadLine());
            Console.Write("Geben Sie die zweite Zahl ein: ");
            //bitte in einer Zeile eingeben
            doubleVariable1 = Convert.ToDouble(
                (Console.ReadLine()));
            Console.Write("Geben Sie die dritte Zahl ein: ");
            //bitte in einer Zeile eingeben
            doubleVariable2 = Convert.ToDouble(
                (Console.ReadLine()));
            //bitte in einer Zeile eingeben
            Console.WriteLine("Das Ergebnis von {0} + {1} + {2}
ist {3}", intVariable1, doubleVariable1,
doubleVariable2, intVariable1 + doubleVariable1 +
doubleVariable2);
        }
    }
}
```

B. Glossar

.NET Framework	<p>Das .NET Framework schafft Rahmenbedingungen für das plattformunabhängige Programmieren. Dazu werden die Programme zunächst in eine Zwischensprache (CIL) übersetzt. Diese Zwischensprache wird dann auf jedem einzelnen Computer durch einen <i>Just-in-Time-Compiler</i> (JIT) in ein ausführbares Programm umgewandelt. Der Just-in-Time-Compiler selbst ist Bestandteil einer Laufzeitumgebung – der <i>Common Language Runtime</i> (CLR).</p> <p>Das .NET Framework stellt außerdem zahlreiche vorgefertigte Funktionen und Steuerelemente für die Entwicklung zur Verfügung.</p>
American Standard Code for Information Interchange	Siehe ASCII
Anweisungsteil	<p>Der Anweisungsteil ist der Teil einer Methode, der die Anweisungen enthält. Der Anweisungsteil wird immer durch geschweifte Klammern umfasst.</p> <p>Die Anweisungen im Anweisungsteil müssen durch ein Semikolon getrennt werden.</p>
Arithmetik	Die Arithmetik ist ein Teilgebiet der Mathematik, das sich mit Zahlen und den für sie geltenden Rechenregeln befasst.
ASCII	<p>ASCII steht für <i>American Standard Code for Information Interchange</i>.</p> <p>ASCII bildet Zeichen über einen Zahlencode ab.</p>
Bezeichner	<p>Ein Bezeichner ist der Name eines Datenobjekts.</p> <p>Für die Vergabe von Bezeichnern gibt es feste Regeln. So dürfen Sie zum Beispiel keine Schlüsselwörter verwenden und müssen einen Bezeichner mit einem Buchstaben oder dem Unterstrich <code>_</code> beginnen.</p>
Binärer Operator	Ein binärer Operator ist ein Operator mit zwei Operanden.
Binäres System	Ein binäres System ist ein System, das genau zwei Zustände abbilden kann.
Bit	Ein Bit ist die kleinste Einheit in der elektronischen Datenverarbeitung. Es entspricht einer Ziffer im Dualsystem beziehungsweise dem Zustand „Strom an“ oder „Strom aus“.
Byte	Ein Byte fasst acht Bits zusammen.
Casting	Siehe <i>Typecasting</i> .

CIL	Die CIL (<i>Common Intermediate Language</i>) ist eine Zwischensprache für die Ausführung von <i>.NET Framework</i> -Anwendungen. Aus der Zwischensprache werden durch den <i>Just-in-Time</i> -Compiler ausführbare Programme erzeugt.
CLI	CLI steht für <i>Common Language Infrastructure</i> . Es handelt sich um eine gemeinsame Basis für das Erstellen von Computerprogrammen – unabhängig von einer bestimmten Programmiersprache und auch von einer bestimmten Plattform.
CLR	Die CLR (<i>Common Language Runtime</i>) ist die Laufzeitumgebung für <i>.NET Framework</i> -Anwendungen. Sie enthält unter anderem den JIT-Compiler. Die Laufzeitumgebung muss auf jedem Rechner vorhanden sein, der ein Programm, das für das <i>.NET Framework</i> erstellt wurde, ausführen soll.
Common Intermediate Language	Siehe CIL.
Common Language Infrastructure	Siehe CLI.
Common Language Runtime	Siehe CLR.
Datentyp	Der Datentyp beschreibt, welche Informationen mit einem Datenobjekt verarbeitet werden. C# kennt unter anderem verschiedene Datentypen für ganze Zahlen, Gleitkommazahlen und Zeichen. Zusätzlich können beliebige Datentypen durch den Programmierer definiert werden.
Dekrementoperator	Mit dem Dekrementoperator -- können Sie den Wert eines ganzzahligen Datentyps um 1 verringern.
Dualsystem	Das Dualsystem arbeitet mit den beiden Ziffern 0 und 1 und dem Stellenwert 2. Eine Zahl wird durch Potenzen des Stellenwerts 2 dargestellt.
Eingabeaufforderung	Die Eingabeaufforderung ist ein Teil von Windows, über den Konsolenprogramme gestartet werden können.
Escape-Sequenzen	Escape-Sequenzen bilden Steuerzeichen und Sonderzeichen ab. Eine Escape-Sequenz beginnt immer mit dem Zeichen \.
Initialisierung	Die Initialisierung ist die erste Zuweisung eines Wertes an ein Datenobjekt.

Inkrementoperator	Mit dem Inkrementoperator <code>++</code> können Sie den Wert eines ganzzahligen Datentyps um 1 erhöhen.
JIT-Compiler	Ein JIT-Compiler (<i>Just in Time</i>) erzeugt aus den Anweisungen einer Zwischensprache ausführbare Programme.
Just-in-Time-Compiler	Siehe JIT-Compiler.
Kommentar	Ein Kommentar ist ein Teil eines Quelltextes, der nicht ausgeführt wird. Kommentare werden durch <code>//</code> oder <code>/* */</code> gekennzeichnet.
Konsolenprogramm	Ein Konsolenprogramm ist ein Programm, das unter der Eingabeaufforderung von Windows läuft.
Konstante	Eine Konstante ist ein Datenobjekt, das seinen Wert einmal zugewiesen bekommt und danach nicht mehr ändern kann. Der Begriff Konstante wird auch allgemein für feste Werte benutzt – zum Beispiel <code>1234</code> oder „Hallo“. Solche Konstanten werden auch Literal genannt.
Literal	Ein Literal ist ein fester Wert, der einem Datenobjekt zugewiesen wird.
Main()	<code>Main()</code> ist die Hauptmethode eines C#-Konsolenprogramms.
Modulo	Modulo liefert den Rest einer Division.
Namensraum	Über den Namensraum kann die Gültigkeit von Bezeichnern festgelegt werden.
Operand	Ein Operand ist der Teil eines Ausdrucks, auf den ein Operator wirkt.
Operator	Ein Operator ist ein Symbol, das eine bestimmte Operation auslöst – zum Beispiel eine Rechenoperation. C# kennt verschiedene Arten von Operatoren – zum Beispiel <ul style="list-style-type: none"> • arithmetische Operatoren, • logische Operatoren und • Vergleichsoperatoren.
Reelle Zahlen	Eine reelle Zahl ist eine Zahl mit Nachkommastellen.
Reservierte Wörter	Reservierte Wörter sind Zeichenketten, die intern vom Compiler verwendet werden. Solche Zeichenketten dürfen Sie nicht selbst benutzen. Reservierte Wörter sind zum Beispiel <code>using</code> oder <code>while</code> .

String	<i>String</i> ist eine andere Bezeichnung für Zeichenkette.
Symbolische Konstante	Siehe Konstante.
Ternärer Operator	Der ternäre Operator ist ein Operator mit drei Operanden.
Typableitung	Siehe Typinferenz.
Typinferenz	Bei der Typinferenz kann der Datentyp einer Variablen aus der ersten Zuweisung abgeleitet werden.
Unärer Operator	Ein unärer Operator ist ein Operator, der nur einen Operanden hat.
Unicode	<i>Unicode</i> ist ein Zeichensatz für die Darstellung von alphanumerischen Werten. Er kann mindestens 65 536 unterschiedliche Zeichen abbilden.
UTF-8	UTF-8 ist eine Variante des <i>Unicode</i> -Zeichensatzes.
Variable	Eine Variable ist ein Datenobjekt, das seinen Wert beliebig verändern kann.
Vereinfachende Operatoren	Vereinfachende Operatoren ermöglichen eine verkürzte Schreibweise in der folgenden Form: <code>zahl +=5; //entspricht zahl = zahl + 5;</code>
Vergleichsoperator	Mit dem Vergleichsoperator == werden Werte verglichen. Der Operator wird schnell mit dem Zuweisungsoperator = verwechselt.
Zahlenformatzeichenfolge	Über die Zahlenformatzeichenfolge können Sie dem Compiler mitteilen, in welcher Form ein numerischer Wert ausgegeben werden soll.
Zeichenkette	Eine Zeichenkette ist eine beliebige Folge von Zeichen. Eine Zeichenkette wird in C# durch Anführungszeichen umfasst.
Zuweisungsoperator	Der Zuweisungsoperator = wird benutzt, um einem Datenobjekt einen Wert zuzuweisen.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch.*

Spracheinführung, Objektorientierung, Programmiertechniken.
8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019.* Ideal für Programmieranfänger.
6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 4.1	Bitweise Darstellung einer Zahl mit einfacher Genauigkeit	40
Abb. 6.1	Die Meldung in der Eingabeaufforderung (interessant ist vor allem der Text am Mauszeiger)	67

E. Tabellenverzeichnis

Tab. 2.1	Steuerzeichen mit Escape-Sequenzen	14
Tab. 2.2	Escape-Sequenzen für besondere Zeichen	15
Tab. 3.1	Arithmetische Operatoren in C#	17
Tab. 3.2	Rangfolge der C#-Operatoren.....	18
Tab. 3.3	Beispiele für Bezeichner	26
Tab. 4.1	Die Datentypen von C# für ganze Zahlen	41
Tab. 4.2	Die Datentypen von C# und ihre .NET Framework-Entsprechungen	52

F. Codeverzeichnis

Code 1.1	Das kleinste mögliche C#-Programm unter Visual Studio	3
Code 1.2	Grundsätzliche Struktur eines C#-Programms mit mehreren Anweisungen	5
Code 1.3	C#-Programm mit Kommentaren	6
Code 2.1	Das „Hallo Welt“-Programm.....	9
Code 2.2	Textausgabe mit und ohne Zeilenumbruch	12
Code 2.3	Textausgabe mit Zeilenumbrüchen über die Escape-Sequenz \n	13
Code 3.1	Arithmetische Operationen in C#	19
Code 3.2	Verkürzter Quelltext für Code 3.1	21
Code 3.3	Beispiel für die Verwendung von Variablen	24
Code 3.4	Komplexeres Beispiel für die Arbeit mit Variablen	28
Code 3.5	Beispiel für weitere Operatoren	32
Code 3.6	Beispiel für die Verwendung von symbolischen Konstanten	34
Code 4.1	Eine Liste der ganzzahligen Datentypen	42
Code 4.2	Ein provoziertes Überlauf	44
Code 4.3	Datentypen für Gleitkommazahlen im praktischen Einsatz	46
Code 4.4	Die Datentypen string und char	50
Code 4.5	Der Datentyp bool	51
Code 5.1	Zuweisungen zwischen Variablen unterschiedlichen Typs (es handelt sich um ein Fragment, das sich nicht übersetzen lässt)	56
Code 5.2	Divisionen mit verschiedenen Datentypen	57
Code 5.3	Typecasting	59
Code 5.4	Casting bei Berechnungen mit den Typen byte und short (es handelt sich um ein Fragment, das sich nicht übersetzen lässt)	60
Code 5.5	Die Konvertierung von Zeichenketten in Zahlen	61
Code 6.1	Eingaben über Console.Read() und Console.ReadLine()	64
Code 6.2	Einlesen mit Konvertierung	66
Code 7.1	Gestaltung von Konsolenprogrammen	70

G. Sachwortverzeichnis

A	
Anweisungsteil	4
Arithmetik	17
ASCII	40
ASCII-Code	40
B	
Bezeichner	23
Bildschirmausgabe	9
Bit	38
Byte	38
C	
Casting	58
D	
Darstellung	
binäre	37
Datentyp	23, 37
des .NET Frameworks	52
für ganze Zahlen	41
für Gleitkommazahlen	44
für logische Werte	51
für Zeichen und Zeichenketten	49
Dekrement-Operator	30
Dezimalsystem	38
Dualsystem	37
E	
Eingabe	63
Escape-Sequenz	12
Exponent	39
G	
Gleitkommawert	39
I	
Initialisierung	29
Inkrement-Operator	30
K	
Kommentar	5
Kommentierung	
Regeln für die	7
Konsolenanwendung	
die Gestaltung von	69
Konstante	33
symbolische	33
L	
Literal	33
M	
Mantisse	39
Methode	4
N	
Namenskonvention	25
Namensraum	9
Norm IEEE 754r	40
Normierung	39
O	
Operator	
arithmetischer	17
binärer	19
ternärer	19
unärer	19
vereinfachender	31
P	
Priorität	18
S	
Schlüsselwörter	25
String	11
T	
Typableitung	53
Typecasting	
explizites	58
Typinferenz	53

U

- Überlauf 43
Unicode 40, 49

V

- Variable 22
lokale 29
Vereinbarung
einer Variablen 24

Z

- Zahlenformatzeichenfolge 48
Zeichenkette 11
interpolierte 22
Zuweisungsoperator 24

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP02D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

1. Schreiben Sie ein Programm, das folgende Ausgabe am linken Rand des Konsolefensters erzeugt:

```
"#
#
#
#
#
#
"
#
#
#
#
#
#
"
#
#
#
#
#
#
"
```

Verwenden Sie für die Positionierung der Zeichen keine Leerzeichen. Der genaue Abstand zwischen den Zeichen ist Ihnen freigestellt. Lediglich die Ausrichtung der Zeichen sollte so aussehen wie im Beispiel.

Kennzeichnen Sie Ihren Code mit dem folgenden Programmkopf:

```
/* #####Einsendeaufgabe 2.1#####
##### */
```

30 Pkt.

2. Schreiben Sie ein Programm, das zwei `int`-Werte über die Tastatur einliest und in `int`-Variablen ablegt. Der erste eingelesene Wert soll durch den zweiten eingelesenen Wert dividiert werden. Die Ausgabe soll als Typ `double` erfolgen. Achten Sie dabei unbedingt darauf, dass auch für Divisionen wie $10 / 3$ die Stellen nach dem Komma im Ergebnis erscheinen.

Kennzeichnen Sie Ihren Code mit dem folgenden Programmkopf:

```
/* ##### Einstudeaufgabe 2.2 ##### */

```

40 Pkt.

3. Geben Sie für die folgenden Wertebereiche einen geeigneten Datentyp an. Verwenden Sie dabei den kleinstmöglichen Datentyp und achten Sie auf das Vorzeichen.

Wertebereich	Datentyp
50.33 bis 10012.61	
-5 bis 33 000	
0.1234567123 bis 1	
K bis Z	
$5.66 \cdot 10^{40}$ bis $5.66 \cdot 10^{50}$	

10 Pkt.

4. Das folgende Programm führt Berechnungen mit Stunden und Minuten durch. Ändern Sie den Quelltext so, dass statt der Zahlen entsprechende symbolische Konstanten verwendet werden.

```
/* ##### Einstudeaufgabe 2.4 ##### */
using System;

namespace Aufgabe0204
{
    class Program
    {
        static void Main(string[] args)
        {
            int variable;
            variable = 60;
            //bitte in einer Zeile eingeben
            Console.WriteLine("Eine Stunde hat {0} Minuten.", variable);
            variable = 60 * 24;
        }
    }
}
```

```
Console.WriteLine("Ein Tag hat {0} Minuten.",  
    variable);  
variable = 60 * 24 * 7;  
//bitte in einer Zeile eingeben  
Console.WriteLine("Eine Woche hat {0} Minuten.",  
    variable);  
}  
}  
}
```

20 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Logische Operatoren und Vergleichsoperatoren,
Kontrollstrukturen und Schleifen, Methoden

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP03D

Objektorientierte Software-Entwicklung mit C#

Logische Operatoren und Vergleichsoperatoren, Kontrollstrukturen und Schleifen, Methoden

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Logische Operatoren und Vergleichsoperatoren, Kontrollstrukturen und Schleifen, Methoden

Inhaltsverzeichnis

Einleitung	1
1 Logische Operatoren und Vergleichsoperatoren	3
1.1 Vergleichsoperatoren	3
1.2 Logische Operatoren	5
1.3 Kombination von logischen Operatoren mit Vergleichsoperatoren	10
Zusammenfassung	13
2 Kontrollstrukturen	15
2.1 Einfache Verzweigung mit if.....	15
2.2 Alternative Verzweigung mit if ... else	20
2.3 Geschachtelte Verzweigungen	22
2.4 Mehrfachauswahl mit switch ...case	26
Zusammenfassung	33
3 Schleifen	36
3.1 Kopfgesteuerte Schleife mit while	37
3.2 Fußgesteuerte Schleife mit do ... while	39
3.3 Zählschleife mit for	43
3.4 Die Anweisungen break und continue bei Schleifen	47
3.4.1 break	48
3.4.2 continue	52
3.5 Exkurs: Auffangen von ungültigen Eingaben	54
Zusammenfassung	55
4 Methoden	58
4.1 Methoden in C#	58
4.2 Methoden mit Rückgabewert	63
4.3 Methoden mit Argumenten	69
4.4 Tipps zum Arbeiten mit Methoden	73
Zusammenfassung	75

Schlussbetrachtung	77
---------------------------------	----

Anhang

A. Lösungen der Aufgaben zur Selbstüberprüfung	78
B. Glossar	83
C. Literaturverzeichnis	87
D. Abbildungsverzeichnis	88
E. Tabellenverzeichnis	89
F. Codeverzeichnis	90
G. Medienverzeichnis	91
H. Sachwortverzeichnis	92
I. Einsendeaufgabe	95

Einleitung

In diesem Studienheft stellen wir Ihnen zunächst noch einige weitere Operatoren für Vergleiche und logische Operationen vor. Anschließend erfahren Sie, wie Sie den Ablauf Ihrer Programme in Abhängigkeit von Bedingungen steuern können und wie Sie Anweisungen mit Schleifen wiederholen. Abschließend beschäftigen wir uns mit Methoden.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie Operatoren für Vergleiche verwenden,
- wie Sie logische Operatoren wie UND und NICHT einsetzen,
- wie Sie logische Operatoren und Vergleichsoperatoren kombinieren,
- wie Sie mit der `if`-Anweisung Befehle abhängig von Bedingungen ausführen lassen,
- wie Sie geschachtelte Verzweigungen erstellen,
- wie Sie Mehrfachauswahlen mit der Anweisung `switch ... case` erstellen,
- welche verschiedenen Schleifen C# unterstützt,
- wie Sie Anweisungen mit Schleifen wiederholen lassen,
- wie Sie die Verarbeitung von Schleifen mit `break` und `continue` beeinflussen,
- wie Sie in einer Schleife sicherstellen, dass Daten in der korrekten Form eingelesen werden,
- wie Sie mit Methoden Quelltext einmal erstellen und dann beliebig oft verwenden,
- wie Sie Rückgabewerte aus Methoden verarbeiten und
- wie Sie Werte an Methoden übergeben.

Beginnen wir mit den logischen Operatoren und den Vergleichsoperatoren.

Christoph Siebeck

1 Logische Operatoren und Vergleichsoperatoren

In diesem Kapitel werden wir uns mit zwei weiteren Operatortypen beschäftigen: den **Vergleichsoperatoren** und den **logischen Operatoren**.

1.1 Vergleichsoperatoren

Vergleichsoperatoren werden – wie der Name schon sagt – für den Vergleich von Daten benutzt. Sie können zum Beispiel zwei Zahlen miteinander vergleichen, eine Zahl mit dem Wert einer Variablen oder auch die Werte in zwei Variablen.

C# unterstützt folgende Vergleichsoperatoren:

Tab. 1.1: Vergleichsoperatoren von C#

Operator	Bedeutung
<code>==</code>	Es wird überprüft, ob die beiden Operanden gleich sind.
<code>!=</code>	Es wird überprüft, ob die beiden Operanden ungleich sind.
<code><</code>	Es wird überprüft, ob der linke Operand kleiner ist als der rechte Operand.
<code>></code>	Es wird überprüft, ob der linke Operand größer ist als der rechte Operand.
<code><=</code>	Es wird überprüft, ob der linke Operand kleiner oder gleich dem rechten Operanden ist.
<code>>=</code>	Es wird überprüft, ob der linke Operand größer oder gleich dem rechten Operanden ist.

Bitte beachten Sie:

Der Vergleichsoperator `==` besteht aus **zwei** Zeichen. Achten Sie sorgfältig darauf, dass Sie ihn nicht mit dem Zuweisungsoperator `=` verwechseln.



Achten Sie auch darauf, Operatoren, die aus zwei Zeichen bestehen, zusammenzuschreiben. Beim Operator `<=` darf zum Beispiel zwischen den beiden Zeichen kein Leerzeichen stehen.

Tipp:

Wenn Sie sich den Unterschied zwischen den Operatoren `<` und `>` nicht merken können, gibt es eine einfache „Eselsbrücke“: Die Spitze zeigt immer zu dem kleineren der beiden Werte beziehungsweise zu dem Wert, der auf kleiner als überprüft werden soll.

Das Ergebnis eines Vergleichs ist entweder **wahr** oder **falsch** beziehungsweise `true` oder `false`.

Einige Beispiele für den Einsatz von Vergleichsoperatoren finden Sie im folgenden Code.

```
/*
#####
Beispiele für Vergleichsoperatoren
#####
*/

using System;

namespace Cshp03d_01_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl1, zahl2;
            zahl1 = 1;
            zahl2 = 0;

            //ein direkter Vergleich von zwei Zahlen
            Console.WriteLine("5 < 10 \t{0}", 5 < 10);

            //ein Vergleich einer Zahl mit einer Variablen
            Console.WriteLine("5 > {0} \t{1}", zahl1, 5 > zahl1);

            //Vergleiche von zwei Variablen
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("{0} == {1} \t{2}", zahl1, zahl2,
            zahl1 == zahl2);
            Console.WriteLine("{0} != {1} \t{2}", zahl1, zahl2,
            zahl1 != zahl2);
        }
    }
}
```

Code 1.1: Beispiele für Vergleichsoperatoren



Zur Auffrischung:

Mit der Anweisung `using System;` wird der Namensraum `System` für den gesamten Code vereinbart. Dadurch ersparen Sie sich die ständige Eingabe von `System.` vor vielen Anweisungen.

Die Anweisung `namespace Cshp03d_01_01` vereinbart einen eigenen Namensraum für das Programm.

Das Programm sollte die folgende Ausgabe erzeugen:

```
5 < 10 True
5 > 1 True
1 == 0 False
1 != 0 True
```

Schauen wir uns an, wie diese Ausgaben zustande kommen.

Zuerst vergleichen wir mit dem Ausdruck `5 < 10` (5 kleiner als 10) zwei Zahlen. Das Ergebnis ist wahr. Deshalb wird `True` ausgegeben.

Danach vergleichen wir die Zahl 5 mit dem Wert der Variablen `zahl1`. Der Ausdruck ergibt `5 > 1` (5 größer als 1) und ist ebenfalls wahr.

Anschließend vergleichen wir die beiden Variablen `zahl1` und `zahl2` miteinander. Der Ausdruck `zahl1 == zahl2` (`zahl1` gleich `zahl2`) liefert `1 == 0` und ist falsch. Deshalb wird hier `False` ausgegeben. Der Ausdruck `zahl1 != zahl2` (`zahl1` ungleich `zahl2`) dagegen liefert `1 != 0` und ist wahr. Also wird `True` ausgegeben.

Probieren Sie einmal aus, was geschieht, wenn Sie den Operator `==` in der vorletzten Anweisung durch den Zuweisungsoperator `=` ersetzen. Sie werden sehen, der Compiler akzeptiert die Anweisung ohne Murren. Als Ergebnis wird dann allerdings

```
1 == 0 0
```

ausgegeben. Denn jetzt liefert der Ausdruck `zahl1 = zahl2` ja nicht mehr `true` oder `false`, sondern das Ergebnis der Zuweisung – also den numerischen Wert 0. Entsprechend wird dann in der letzten Anweisung auch der Wert 0 als Ergebnis der Zuweisung ausgegeben.

Noch einmal, weil es schnell für Fehler sorgen kann:



Achten Sie vor allem bei Ihren ersten Versuchen sehr sorgfältig darauf, dass Sie für Vergleiche den Operator `==` benutzen und nicht versehentlich den Operator `=`. Denn auch eine Zuweisung liefert ein Ergebnis – nämlich den Wert, der zugewiesen wurde.

So viel zu den Vergleichsoperatoren. Kommen wir nun zu den logischen Operatoren.

1.2 Logische Operatoren

Mit logischen Operatoren können Sie Ausdrücke – zum Beispiel Ergebnisse von Vergleichen – miteinander verknüpfen. Die logischen Operatoren liefern genau wie die Vergleichsoperatoren entweder wahr bzw. `true` oder falsch bzw. `false`.

Sie brauchen logische Operatoren, wenn Sie Anweisungen wie „Wenn es **nicht** regnet, dann gehe ich spazieren“ in Ihren Programmen umsetzen wollen. Wie das geht, werden Sie noch in diesem Heft lernen. Hier wollen wir Ihnen die drei wichtigsten logischen Operatoren daher zunächst einmal nur vorstellen. Sie finden sie in der folgenden Tabelle:

Tab. 1.2: Logische Operatoren von C#

Operator	Bedeutung
!	Logische Verneinung (NICHT, NOT)
&&	Logisches Und (UND, AND)
	Logisches Oder (ODER, OR)

Hinweis:

Das Zeichen | erhalten Sie mit der Tastenkombination `Alt Gr` + `z|`.

**Bitte beachten Sie:**

Die logischen Operatoren `&&` und `||` bestehen aus **zwei** Zeichen und nicht aus einem. Mit den Operatoren `&` und `|` führen Sie bitweise Verknüpfungen durch beziehungsweise Sie verknüpfen Bedingungen ohne optimierte Auswertung. Damit wollen wir uns hier aber nicht weiter beschäftigen.

Schauen wir uns die logischen Operatoren der Reihe nach an. Beginnen wir mit der **logischen Verneinung** – dem logischen NICHT.

NICHT ist der einfachste logische Operator. Er dreht das Ergebnis einer logischen Aussage um – macht also aus wahr falsch und aus falsch wahr. Im praktischen Einsatz sehen Sie den Operator `!` im folgenden Code. Das Programm verwendet die gleichen Ausdrücke wie der vorige Code und kehrt die Wahrheitswerte über den Operator `!` um:

```
/* ##### Logisches NICHT #####
using System;
namespace Cshp03d_01_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl1, zahl2;
            zahl1 = 1;
            zahl2 = 0;

            //ein direkter Vergleich von zwei Zahlen
            Console.WriteLine("!(5 < 10) \t{0}", !(5 < 10));

            //bitte jeweils in einer Zeile eingeben
            //ein Vergleich einer Zahl mit einer Variablen
            Console.WriteLine("!(5 >{0}) \t{1}", zahl1, !(5 > zahl1));
            //Vergleiche von zwei Variablen
            Console.WriteLine("!( {0} == {1}) \t{2}", zahl1,
            zahl2,! (zahl1 == zahl2));
            Console.WriteLine("!( {0}!= {1}) \t{2}", zahl1,
            zahl2,! (zahl1 != zahl2));
        }
    }
}
```

Code 1.2: Logisches NICHT

Die Ausgabe sollte jetzt so aussehen:

```
!(5 < 10) False
!(5 > 1) False
!(1 == 0) True
!(1 != 0) False
```

Die Wahrheitswerte werden also umgedreht. Der Ausdruck `5 < 10` (5 kleiner als 10) ist eigentlich wahr und liefert das Ergebnis `True`. Durch den Operator `!` wird der Wahrheitswert umgedreht und zu `False`.

Bitte beachten Sie:



Damit der vorige Code übersetzt wird, müssen Sie die gesamten Ausdrücke über den Operator `!` umdrehen. Der Operator `!` muss also jeweils vor der Klammer `(` stehen. Andernfalls würden Sie versuchen, den Wahrheitswert einer Zahl umzudrehen – und das ist in C# nicht möglich.

Achten Sie deshalb beim Einsatz von Operatoren sehr sorgfältig auf die Klammern.

So viel zur logischen Verneinung.

Das **logische ODER** `||` verknüpft zwei Aussagen und liefert immer dann den Wert wahr zurück, wenn mindestens eine der Aussagen wahr ist. Umgangssprachlich lässt sich das logische ODER zum Beispiel so darstellen:

„Wenn die Sonne scheint **oder** wenn es warm ist, dann kann ich ein T-Shirt anziehen.“

Um ein T-Shirt anzuziehen, reicht es, wenn **entweder** die Sonne scheint **oder** wenn es warm ist.

Es gibt nur einen einzigen Fall, bei dem das logische ODER den Wert falsch zurückliefert – nämlich dann, wenn beide Aussagen den Wert falsch haben.

Die verschiedenen möglichen Kombinationen von zwei Wahrheitswerten durch das logische ODER finden Sie in der folgenden **Wahrheitstabelle** zusammengefasst.

Tab. 1.3: Wahrheitstabelle für das logische ODER

A	B	<code>A B</code>
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Eine ähnliche Tabelle erzeugt auch der folgende Code. Es verknüpft die Werte `true` und `false` mehrfach logisch ODER.

```
/*
#####
Logisches ODER
#####
*/
using System;
```

```
namespace Cshp03d_01_03
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("A \t B \t\t A || B");
            Console.WriteLine("wahr \t wahr \t\t {0}", (true || true));
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("wahr \t falsch \t\t {0}", (true || false));
            Console.WriteLine("falsch \t wahr \t\t {0}", (false || true));
            Console.WriteLine("falsch \t falsch \t\t {0}", (false || false));
        }
    }
}
```

Code 1.3: Logisches ODER**Noch einmal zur Erinnerung:**

Die logischen Werte heißen `true` und `false` mit kleinem Anfangsbuchstaben. Ausgegeben werden aber immer `True` beziehungsweise `False` mit großem Anfangsbuchstaben.

Jetzt bleibt uns noch der letzte logische Operator – nämlich `&&` für das **logische UND**.

Dieser Operator liefert nur dann wahr, wenn auch die beiden verbundenen Ausdrücke wahr sind. Für jeden anderen Fall wird falsch geliefert. Umgangssprachlich lässt sich das logische UND so darstellen:

„Wenn der Stecker in der Steckdose steckt **und** wenn das Radio eingeschaltet ist, höre ich Musik.“

Hier müssen also beide Ausdrücke wahr sein. Wenn nur der Stecker in der Steckdose steckt, das Radio aber nicht eingeschaltet ist, hören Sie keine Musik. Und wenn nur das Radio eingeschaltet ist, aber der Stecker nicht eingesteckt ist, hören Sie ebenfalls keine Musik.

Die Wahrheitstabelle für das logische UND sieht also so aus:

Tab. 1.4: Wahrheitstabelle für das logische UND

A	B	A && B
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Das Beispielprogramm sieht fast genauso aus wie bei dem logischen ODER:

```
/*
#####
Logisches UND
#####
*/
using System;

namespace Cshp03d_01_04
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("A \t B \t\t A && B");
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("wahr \t wahr \t\t {0}", (true &&
            true));
            Console.WriteLine("wahr \t falsch \t\t {0}", (true &&
            false));
            Console.WriteLine("falsch \t wahr \t\t {0}", (false &&
            true));
            Console.WriteLine("falsch \t falsch \t\t {0}", (false &&
            false));
        }
    }
}
```

Code 1.4: Logisches UND

Bitte beachten Sie:

Die Auswertung von logischen Verknüpfungen über die Operatoren `&&` und `||` wird nicht immer vollständig ausgeführt, sondern intern optimiert. Bei einer logischen UND-Verknüpfung wird die Auswertung zum Beispiel abgebrochen, sobald der erste Ausdruck falsch ergibt. Denn dann wird ja auch zwangsläufig das Ergebnis der gesamten Verknüpfung falsch, da bei einer logischen UND-Verknüpfung alle Ausdrücke wahr sein müssen, damit das Gesamtergebnis wahr ist.



Bei einer logischen ODER-Verknüpfung wird die Auswertung abgebrochen, sobald der erste Ausdruck wahr ergibt. Denn dann wird auch zwangsläufig das Gesamtergebnis wahr.

Wenn Sie dagegen die Operatoren `&` und `|` benutzen, werden sämtliche Ausdrücke ausgewertet. Sie sollten daher – wann immer möglich – die Operatoren `&&` und `||` für logische Verknüpfungen verwenden. Damit wird zum einen Ihr Programm etwas schneller, zum anderen droht keine Gefahr, dass Sie die Operatoren für die logischen Verknüpfungen mit den Operatoren für die bitweisen Verknüpfungen verwechseln.

1.3 Kombination von logischen Operatoren mit Vergleichsoperatoren

Die logischen Operatoren und die Vergleichsoperatoren können Sie beliebig miteinander kombinieren. So lassen sich zum Beispiel Ausdrücke wie „A ist kleiner als B und größer als C“ oder „A ist gleich B und B ist ungleich C“ erstellen.

Schauen wir uns dazu einmal das folgende Beispielprogramm an. Es liest drei Zahlen ein und verknüpft dann die Werte dieser Zahlen über Ausdrücke:

```
/* ##### Kombination von logischen Operatoren mit Vergleichsoperatoren ##### */
Kombination von logischen Operatoren mit Vergleichsoperatoren
##### */

using System;

namespace Cshp03d_01_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int wertA, wertB, wertC;
            bool ergebnis;

            //Eingabeteil
            Console.Write("Geben Sie den Wert für A ein: ");
            wertA = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie den Wert für B ein: ");
            wertB = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie den Wert für C ein: ");
            wertC = Convert.ToInt32(Console.ReadLine());

            //Ausgabeteil
            ergebnis = (wertA < wertB) && (wertB < wertC);
            //bitte in einer Zeile eingeben
            Console.WriteLine("A ist kleiner als B und B ist kleiner als C: {0}", ergebnis);

            ergebnis = (wertA < wertB) || (wertB < wertC);
            //bitte in einer Zeile eingeben
            Console.WriteLine("A ist kleiner als B oder B ist kleiner als C: {0}", ergebnis);

            ergebnis = (wertA == wertB) || !(wertB < wertC);
            //bitte in einer Zeile eingeben
            Console.WriteLine("A ist gleich B oder B ist nicht kleiner als C: {0}", ergebnis);
        }
    }
}
```

Code 1.5: Kombination von logischen Operatoren mit Vergleichsoperatoren

Gehen wir den Code der Reihe nach durch:

Zunächst werden drei Variablen vom Typ `int` vereinbart und eine vom Typ `bool`. Das kennen Sie bereits.

Mit den folgenden Anweisungen lesen wir dann Werte für die drei Variablen `wertA`, `wertB` und `wertC` über die Tastatur ein. Das ist ebenfalls nichts Neues.

Zur Auffrischung:

Die Anweisung `Console.ReadLine()` liefert eine Zeichenkette vom Typ `string` zurück. Diese Zeichenkette müssen Sie ausdrücklich über die Anweisung `Convert.ToInt32()` in eine Zahl konvertieren.

Bitte beachten Sie, dass das Programm abstürzt, wenn Sie für einen der Werte keine ganze Zahl eingeben oder beim Einlesen einfach nur die Eingabetaste drücken.



Interessanter wird es dann im Ausgabeteil. Hier weisen wir der Variablen `ergebnis` dreimal den Wert unterschiedlicher Ausdrücke zu und lassen das Ergebnis mit einem beschreibenden Text auf dem Bildschirm ausgeben.

Der erste Ausdruck ist `(wertA < wertB) && (wertB < wertC)`. Er ist dann wahr, wenn sowohl `wertA < wertB` als auch `wertB < wertC` gilt. Der Compiler wertet zunächst die beiden Vergleiche in den Klammern aus und verknüpft die Ergebnisse dann mit einem logischen UND. Wenn Sie zum Beispiel die Zahlen 10 für `wertA`, 11 für `wertB` und 12 für `wertC` eingeben, erfolgt die Auswertung so:

$10 < 11$ – also ist der linke Teil wahr.

$11 < 12$ – also ist der rechte Teil auch wahr.

Wahr `&&` wahr ergibt wieder wahr. Damit ist also auch das Ergebnis des gesamten Ausdrucks wahr.

Nehmen wir drei andere Zahlen als Beispiel: `wertA` bekommt den Wert 10, `wertB` den Wert 9 und `wertC` den Wert 12. Die Auswertung sieht dann so aus:

$10 < 9$ – der Ausdruck ist falsch.

$9 < 12$ – also ist der rechte Teil wahr.

Falsch `&&` wahr ergibt falsch. Also ist das Gesamtergebnis falsch.



Noch einmal zur Erinnerung:

Intern wird der zweite Ausdruck gar nicht mehr überprüft. Da bereits der erste Ausdruck falsch ist, ist auch das Ergebnis der gesamten UND-Verknüpfung falsch. Der Wert der folgenden Ausdrücke spielt dabei keine Rolle.

Der zweite Ausdruck `(wertA < wertB) || (wertB < wertC)` unterscheidet sich vom ersten Ausdruck nur durch den Operator `||`. Hier werden die beiden Ausdrücke also logisch ODER verknüpft. Das Ergebnis dieses Ausdrucks ist dann wahr, wenn entweder der linke Teil oder der rechte Teil wahr ist oder wenn beide Teile wahr sind.

Schauen wir uns auch diesen Ausdruck mit den Zahlen von oben an. Beginnen wir mit den Werten 10 für `wertA`, 11 für `wertB` und 12 für `wertC`.

$10 < 11$ – also ist der linke Teil wahr.

$11 < 12$ – also ist der rechte Teil auch wahr.

Wahr `||` wahr ergibt wieder wahr. Also ist auch hier das Ergebnis wahr, obwohl wir den Operator `||` benutzt haben.



Auch hier wird der zweite Ausdruck intern nicht weiter überprüft. Da bereits der erste Ausdruck wahr ist, spielt das Ergebnis der weiteren Ausdrücke bei einer logischen ODER-Verknüpfung keine Rolle mehr.

Schauen wir uns jetzt die Werte 10 für `wertA`, 9 für `wertB` und 12 für `wertC` an.

$10 < 9$ – der Ausdruck ist falsch.

$9 < 12$ – also ist der rechte Teil wahr.

Falsch `||` wahr ergibt wahr. Also ist auch hier das Ergebnis wahr.

Der dritte Ausdruck (`wertA == wertB`) `||` `!(wertB < wertC)` enthält einen NICHT-Operator, der auf den gesamten rechten Teil wirkt. Der Ausdruck `wertB < wertC` wird damit negiert – also umgedreht.

Mit den Werten 10 für `wertA`, 11 für `wertB` und 12 für `wertC` würde die Auswertung so laufen:

$10 == 11$ ist falsch. Also ist der linke Teil falsch.

$11 < 12$ ist wahr und wird negiert. Damit ist der rechte Teil ebenfalls falsch.

Falsch `||` falsch ergibt falsch – also ist das Gesamtergebnis falsch.

Wenn wir die Werte 10 für `wertA`, 9 für `wertB` und 12 für `wertC` verwenden, sieht die Auswertung für den letzten Ausdruck so aus:

$10 == 9$ ist falsch. Damit ist der linke Teil falsch.

$9 < 12$ ist wahr und wird negiert. Damit ist der rechte Teil ebenfalls falsch.

Auch hier ist das Gesamtergebnis wieder falsch.

Damit Sie ein Gefühl für logische Ausdrücke bekommen, sollten Sie jetzt ein wenig mit dem Programm aus dem vorigen Code experimentieren. Geben Sie unterschiedliche Zahlen ein und ersetzen Sie die vorgegebenen Ausdrücke zum Beispiel durch die folgenden Ausdrücke:

```
(wertA != wertB) && (wertA < wertB)
(wertA <= wertB) || (wertA < wertC)
((wertA != wertB) && (wertA < wertB)) || (wertB == wertC)
```

Versuchen Sie nachzuvollziehen, was diese Ausdrücke bedeuten. Der letzte Ausdruck sieht auf den ersten Blick kompliziert aus, sollte Ihnen aber keine großen Schwierigkeiten bereiten, wenn Sie ihn in Umgangssprache „übersetzen“ und die Klammern beachten.

Wenn Sie mit dem Ausdruck nicht klarkommen: Er bedeutet: ((wertA ungleich wertB) UND (wertA kleiner wertB)) ODER (wertB gleich wertC).

Zusammenfassung

Mit Vergleichsoperatoren können Sie Daten – zum Beispiel Zahlen oder Werte in Variablen – vergleichen.

Mit logischen Operatoren können Sie Aussagen miteinander verknüpfen.

Das Ergebnis eines Vergleichs oder einer logischen Verknüpfung ist entweder wahr oder falsch.

Vergleichsoperatoren und logische Operatoren können Sie nahezu beliebig miteinander kombinieren.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Mit welchem Operator können Sie überprüfen, ob zwei Zahlen gleich sind? Mit welchem Operator können Sie überprüfen, ob eine Zahl größer ist als eine andere?

- 1.2 Geben Sie für die folgenden Ausdrücke jeweils an, ob das Ergebnis wahr oder falsch ist:

a) $1 \leq 2$

b) $1 < 1$

c) `2 != 1`

d) `wahr && falsch`

e) `wahr || falsch`

f) `wahr && !(2 < 1)`

g) `1 < 0 || !(2 != 0)`

- 1.3 Erstellen Sie eine Wahrheitstabelle für den Ausdruck `!(A || B)`. Beispiele für solche Wahrheitstabellen finden Sie im Kapitel.

2 Kontrollstrukturen

*Mit den logischen Ausdrücken aus den letzten Kapiteln konnten wir bisher vor allem die Ergebnisse auf dem Bildschirm ausgeben lassen. Das ist natürlich nicht sonderlich spannend. Richtig interessant werden logische Ausdrücke, wenn Sie sie benutzen, um den Ablauf Ihrer Programme über **Kontrollstrukturen** zu steuern. Dann können Sie nämlich die Ausführung von Anweisungen an Bedingungen knüpfen oder auch Anweisungen gezielt wiederholen. Wie das genau funktioniert, lernen Sie in diesem Kapitel.*

Normalerweise werden Anweisungen bei einfachen Programmen exakt in der Reihenfolge abgearbeitet, in der sie im Quelltext stehen. Über Kontrollstrukturen können Sie die Reihenfolge aber auch verändern – zum Beispiel die Ausführung einer Anweisung von einer Bedingung abhängig machen oder Anweisungen gezielt wiederholen.



2.1 Einfache Verzweigung mit if

Beginnen wir mit der einfachsten Kontrollstruktur – der einfachen `if`-Verzweigung. Eine `if`-Verzweigung (übersetzt etwa „wenn ... dann“) wird eingesetzt, wenn eine oder mehrere Anweisungen abhängig von einer Bedingung ausgeführt werden sollen.

Schauen wir uns die `if`-Verzweigung zunächst an einem Beispiel aus der Umgangssprache an. Dargestellt wird ein Tagesablauf:

Aufstehen

Arbeiten

Nach Hause gehen

WENN die Sonne scheint, **DANN** joggen

Abendessen kochen

Die `if`-Verzweigung finden Sie hier bei der vierten Aktion: **WENN** die Sonne scheint, **DANN** joggen.

Wenn die Sonne nicht scheint – die Bedingung also nicht zutrifft, wird die Aktion „joggen“ nicht ausgeführt, sondern es geht direkt mit der fünften Aktion weiter: Das Abendessen wird gekocht.

Diese einfache `if`-Verzweigung lässt sich grafisch so darstellen:

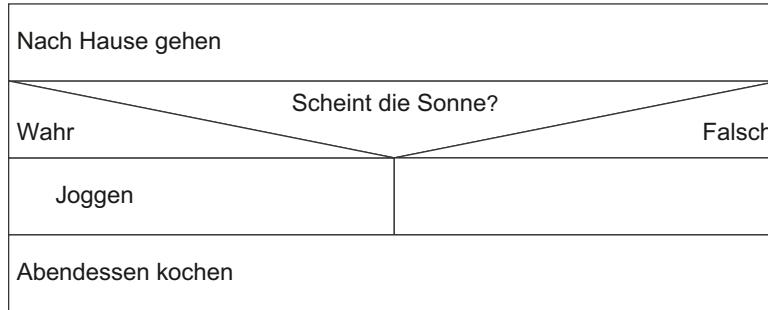


Abb. 2.1: Eine einfache `if`-Verzweigung

Hinweis:

Für die grafische Darstellung der Kontrollstrukturen benutzen wir in diesem Studienheft Nassi-Shneiderman-Diagramme – auch Struktogramme genannt. Andere Formen für die Darstellung von Programmabläufen – wie zum Beispiel Aktivitätsdiagramme der UML – werden Sie später kennenlernen, wenn wir unsere Beispielanwendung entwickeln.

Schauen wir uns jetzt die `if`-Verzweigung in einem C#-Programm an. Der folgende Code soll ermitteln, ob eine eingegebene Zahl größer ist als 5. Damit Sie die `if`-Verzweigung sofort erkennen, haben wir sie im Code durch einen Kommentar markiert und fett hervorgehoben.

```

/*
#####
Einfache if-Verzweigung
#####
*/
using System;

namespace Cshp03d_02_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());
            //die Verzweigung
            if (zahl > 5)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");

            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}
  
```

Code 2.1: Einfache `if`-Verzweigung

Die meisten Anweisungen dürften Ihnen bekannt sein. Neu sind nur die Zeilen

```
if (zahl > 5)  
    Console.WriteLine("Sie haben eine Zahl größer als 5  
eingegeben.");
```

Schauen wir sie uns etwas genauer an:

In der ersten Zeile folgt nach dem Schlüsselwort `if` in runden Klammern der Ausdruck, der die Verzweigung steuert.

Zur Erinnerung:

Bei den Schlüsselwörtern handelt es sich um reservierte Wörter, die vom Compiler für bestimmte Zwecke verwendet werden – zum Beispiel `using` oder eben `if`. Schlüsselwörter werden in der Standardeinstellung vom Editor in blauer Schrift dargestellt.



In unserem Beispiel überprüfen wir mit dem Ausdruck `(zahl > 5)`, ob der Wert der Variablen `zahl`, den wir mit der Anweisung davor über die Tastatur eingelesen haben, größer ist als 5.

Der Ausdruck in einer `if`-Verzweigung – die **Bedingung** – muss entweder logisch wahr oder logisch falsch zurückliefern. Beispiele für solche Ausdrücke haben Sie im vorigen Kapitel kennengelernt.



Falls die Bedingung wahr ist, wird die nachfolgende Anweisung ausgeführt. In unserem Beispiel wird also der Satz `Sie haben eine Zahl größer als 5 eingegeben` ausgegeben.

Beachten Sie bitte, dass es keine Rolle spielt, ob die Anweisung, die ausgeführt werden soll, in der nächsten oder in derselben Zeile wie die Bedingung steht. Sie könnten die `if`-Verzweigung aus dem vorigen Code also auch so schreiben:

```
if (zahl > 5) Console.WriteLine("Sie haben eine Zahl größer als  
5 eingegeben.");
```

Bitte beachten Sie unbedingt:

Sie dürfen **kein** Semikolon hinter den Ausdruck in einer `if`-Anweisung setzen – auch dann nicht, wenn Sie mehr als eine Zeile benutzen. Andernfalls werden nämlich die `if`-Anweisung und die dazugehörige Anweisung getrennt.



Probieren Sie das einfach einmal in dem vorigen Code aus. Setzen Sie hinter die Zeile mit der `if`-Anweisung ein Semikolon und lassen Sie das Programm dann ausführen. Sie werden sehen, die Ausgabe `Sie haben eine Zahl größer als 5 eingegeben` erscheint dann immer – auch wenn Sie eine Zahl kleiner oder gleich 5 eingeben.

Wenn Sie nach der Änderung genau hinsehen, werden Sie auch eine Warnung des Compilers finden, dass möglicherweise eine falsche leere Anweisung vorliegt.

Sie können in der `if`-Verzweigung auch mehr als eine Anweisung ausführen lassen, wenn der Ausdruck wahr ist. Dazu müssen Sie die Anweisungen mit geschweiften Klammern `{ }` in einem **Anweisungsblock** zusammenfassen.

Sehen wir uns dazu das folgende Beispiel an. Es errechnet in einem Anweisungsblock zusätzlich noch die Differenz zwischen dem eingegebenen Wert und dem Wert 5:

```
/*
if-Verzweigung mit Anweisungsblock
*/
using System;

namespace Cshp03d_02_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.WriteLine("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());

            //die Verzweigung, diesmal mit einem Anweisungsblock
            if (zahl > 5)
            {
                //bitte jeweils in einer Zeile eingeben
                Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");
                Console.WriteLine("Die Differenz zwischen {0} und 5 ist
{1}.", zahl, zahl-5);
            }
            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}
```

Code 2.2: `if`-Verzweigung mit Anweisungsblock

Wie Sie sehen, handelt es sich beim vorigen Code um eine leicht veränderte Variante des Code 2.1. Es ist eine zweite Anweisung hinzugekommen, die nur innerhalb der `if`-Verzweigung ausgeführt wird. Zusammen mit der bereits im Code 2.1 vorhandenen Anweisung bilden die beiden Anweisungen jetzt einen Anweisungsblock:

```
{
    Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");
    Console.WriteLine("Die Differenz zwischen {0} und 5 ist {1}.",
zahl, zahl-5);
}
```

Die geschweiften Klammern sorgen dabei dafür, dass die beiden Anweisungen vom Compiler als ein logischer Block betrachtet werden. Sie werden nur dann ausgeführt, wenn die Bedingung wahr ist. Wenn Sie die Klammern weglassen, nehmen Sie die zweite Anweisung aus dem Block heraus. Sie wird dann immer ausgeführt – egal, welchen Wert die Bedingung hat.

Achten Sie daher bei Anweisungsblöcken sehr sorgfältig auf die Position der geschweiften Klammern. Denken Sie bitte auch daran, dass für jede öffnende Klammer { auch eine schließende Klammer } vorhanden sein muss.



Fassen wir die Syntax der `if`-Verzweigung noch einmal zusammen.

Tab. 2.1: Syntax der if-Verzweigung

```
if (Ausdruck)           if (Ausdruck)
    Anweisung;          {
                        Anweisung1;
                        Anweisung2;
                        ...
    }
```

Hinter dem Schlüsselwort `if` steht in runden Klammern der Ausdruck, durch den die Verzweigung gesteuert wird – die Bedingung. Die Bedingung liefert entweder wahr oder falsch. Hinter dem Ausdruck folgt entweder eine Anweisung (links in der Tabelle) oder ein Anweisungsblock (rechts in der Tabelle).

Exkurs: Die Tücken der Operatoren = und ==

Am Code 2.2 können Sie auch noch einmal selbst ausprobieren, wie unangenehm ein Verwechseln der Operatoren `=` und `==` sein kann. Ersetzen Sie den Operator `>` in dem Ausdruck `(zahl > 5)` durch den Operator `==`. Dann werden die Anweisungen in dem Block nur noch ausgeführt, wenn Sie die Zahl 5 eingeben.

Löschen Sie dann eins der Gleichheitszeichen in dem Ausdruck. Sie werden sehen, der Compiler beschwert sich sofort, dass er einen `int`-Typ nicht in einen `bool`-Typ umwandeln kann. Denn der Ausdruck `zahl = 5` liefert ja nicht `true` oder `false` als Ergebnis, sondern den numerischen Wert 5.

Solche Fehler sind gerade mit wenig Programmiererfahrung schnell passiert. Achten Sie daher sorgfältig darauf, dass Sie in Vergleichen mit dem Operator `==` arbeiten, und sehen Sie sich im Zweifelsfall die Fehlermeldung von Visual Studio genau an.

So viel zur einfachen `if`-Verzweigung. Kommen wir jetzt zur `if ... else`-Verzweigung.

2.2 Alternative Verzweigung mit if ... else

Die if ... else-Verzweigung (übersetzt etwa „wenn ... dann – sonst“) wird eingesetzt, wenn Sie neben den Anweisungen, die ausgeführt werden sollen, wenn die Bedingung wahr ist, auch Anweisungen benötigen, die nur dann ausgeführt werden sollen, wenn die Bedingung falsch ist.

Erweitern wir das Beispiel mit dem Tagesablauf aus dem letzten Kapitel einmal um eine if ... else-Verzweigung. Es könnte dann so aussehen:

Aufstehen

Arbeiten

Nach Hause gehen

WENN die Sonne scheint, **DANN** joggen, **SONST** fernsehen

Abendessen kochen

Im Unterschied zur einfachen if-Verzweigung wird hier in jedem Fall eine „Freizeitaktion“ ausgeführt. Die Entscheidung wird davon abhängig gemacht, ob die Sonne scheint. Wenn die Bedingung „Sonne scheint“ erfüllt ist, wird die Aktion „joggen“ ausgeführt. Wenn die Bedingung „Sonne scheint“ dagegen nicht erfüllt ist, wird die Aktion „fernsehen“ ausgeführt.

Grafisch lässt sich diese if ... else-Verzweigung so darstellen:

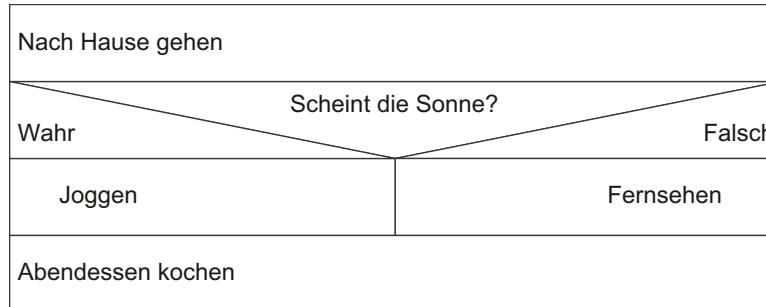


Abb. 2.2: Eine if ... else-Verzweigung

Jetzt fragen Sie sich vielleicht: „Warum soll ich einen eigenen SONST-Zweig einbauen, wenn ich auch eine Anweisung direkt nach dem WENN verwenden kann? Diese Anweisung direkt nach dem WENN wird doch auch ausgeführt, wenn der Ausdruck nicht zutrifft.“

Der Unterschied zwischen den beiden Konstruktionen ist zwar klein, aber bei der Ausführung des Programms erheblich. Sehen wir uns den Tagesablauf einmal an, wenn die Aktion „fernsehen“ nicht im SONST-Zweig steht, sondern als eigene Anweisung direkt hinter dem WENN-Zweig:

Aufstehen

Arbeiten

Nach Hause gehen

WENN die Sonne scheint, DANN joggen

Fernsehen

Abendessen kochen

Wenn Sie diesen Tagesablauf nachspielen, werden Sie den Unterschied sofort sehen: Die Aktion „fernsehen“ ist jetzt völlig unabhängig von der Bedingung bei WENN. Sie wird **immer** ausgeführt – egal, ob die Bedingung zutrifft oder nicht.

Noch deutlicher wird der Unterschied in der grafischen Darstellung. Links sehen Sie den Ablauf mit `if ... else`, rechts dagegen den Ablauf nur mit `if`.

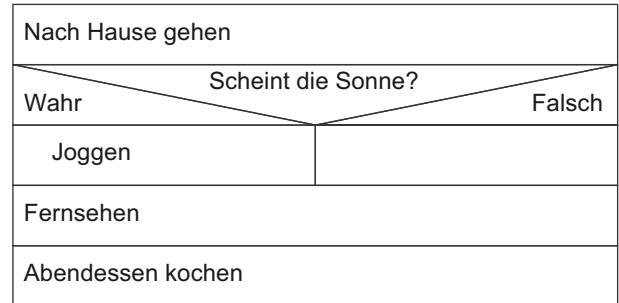
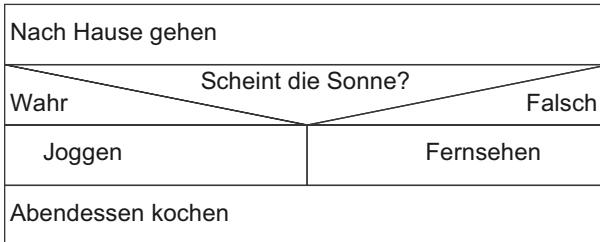


Abb. 2.3: Vergleich von `if ... else` (links) und `if` (rechts)

Achten Sie vor allem darauf, welche Anweisung jeweils nach der Aktion „joggen“ ausgeführt wird. Links wird nach dem Joggen das Abendessen gekocht, rechts dagegen wird nach dem Joggen erst einmal ferngesehen.

Sehen wir uns jetzt die `if ... else`-Verzweigung in einem praktischen Beispiel an. Das folgende Programm erweitert den Code 2.1 und gibt jetzt auch eine Meldung aus, wenn eine Zahl kleiner oder gleich 5 eingegeben wird.

```

/*
#####
if ... else-Verzweigung
#####
*/
using System;

namespace Cshp03d_02_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.WriteLine("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());

            //die Verzweigung, diesmal mit else
            if (zahl > 5)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");
        }
    }
}
  
```

```

        else
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben eine Zahl kleiner als 5
                oder 5 eingegeben.");
            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}

```

Code 2.3: if ... else-Verzweigung

Wie schon bei der einfachen if-Verzweigung können Sie auch hinter else sowohl eine einzelne Anweisung als auch einen Anweisungsblock angeben. Der Anweisungsblock muss dann aber wieder mit den Klammern {} gekennzeichnet werden.

Die Syntax der if ... else-Verzweigung finden Sie noch einmal zusammengefasst in der folgenden Tabelle.

Tab. 2.2: Syntax der if ... else-Verzweigung

if (Ausdruck)	if (Ausdruck)
Anweisung;	{
else	Anweisung1;
Anweisung;	Anweisung2;
	...
	}
	else
	{
	Anweisung1;
	Anweisung2;
	...
	}

2.3 Geschachtelte Verzweigungen

Sie können if-Verzweigungen auch schachteln – also in einer if-Verzweigung mit einer weiteren if-Verzweigung eine weitere Bedingung prüfen. Eine geschachtelte if-Verzweigung könnte so aussehen:

```

if (Ausdruck1)
    if (Ausdruck2)
        Anweisung1;

```

Zuerst wird überprüft, ob Ausdruck1 wahr ist. Wenn das der Fall ist, wird überprüft, ob Ausdruck2 ebenfalls wahr ist. Wenn auch das der Fall ist, wird die Anweisung ausgeführt.

Ist Ausdruck1 dagegen falsch, wird Ausdruck2 gar nicht erst überprüft. Das lässt sich auch sehr einfach in der folgenden Abbildung nachverfolgen.

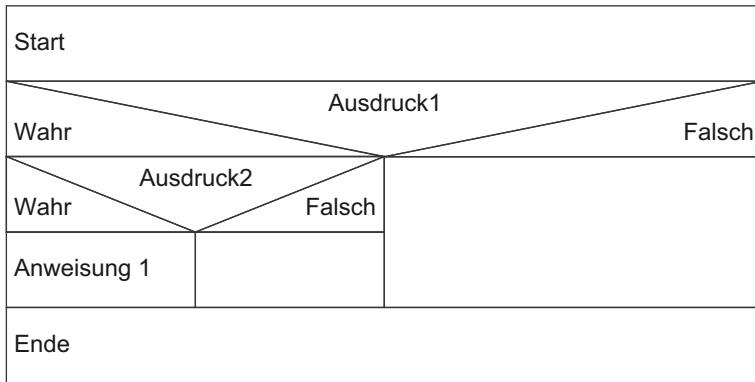


Abb. 2.4: Eine geschachtelte Verzweigung

Da bei einer geschachtelten Verzweigung beide Bedingungen wahr sein müssen, können Sie sie in vielen Fällen auch durch eine logische UND-Verknüpfung ersetzen. Für das Beispiel von oben könnte diese Verknüpfung zum Beispiel so aussehen:

```
if (Ausdruck1 && Ausdruck2)
  Anweisung1;
```

Hier wird ebenfalls überprüft, ob sowohl `Ausdruck1` als auch `Ausdruck2` wahr sind. Nur dann wird die Anweisung ausgeführt.

Die Variante mit der logischen UND-Verknüpfung ist zwar etwas schwieriger zu lesen, dafür aber sehr viel kompakter. Wie Sie bereits an diesem sehr einfachen Beispiel sehen, gibt es beim Programmieren häufig mehrere Möglichkeiten, zum Ziel zu kommen.

Eine wichtige praktische Bedeutung hat die folgende Variante, die noch etwas weiter schachtelt:

```
if (Ausdruck1)
  Anweisung1;
else
  if (Ausdruck2)
    Anweisung2;
  else
    Anweisung3;
```

Hier enthält die erste `if`-Anweisung im `else`-Zweig eine weitere `if`-Anweisung – nämlich `if (Ausdruck2)`.

Der Durchlauf durch diese Konstruktion ist etwas komplizierter als bei den anderen Beispielen: Zuerst wird in der ersten Zeile überprüft, ob `Ausdruck1` wahr ist. Wenn das der Fall ist, wird die Anweisung `Anweisung1` ausgeführt. Danach wird der gesamte Block verlassen, da ja nur noch Anweisungen für den `else`-Zweig folgen, der zu `if (Ausdruck1)` gehört. Es wird also weder `Ausdruck2` geprüft noch werden die Anweisungen `Anweisung2` oder `Anweisung3` ausgeführt.

Ist `Ausdruck1` dagegen falsch, wird im `else`-Zweig mit der Anweisung

```
if (Ausdruck2)
```

weitergemacht.

Ist Ausdruck2 wahr, wird die Anweisung Anweisung2 ausgeführt. Ist Ausdruck2 dagegen falsch, wird der else-Zweig ausgeführt, der zu if (Ausdruck2) gehört – also die Anweisung Anweisung3.

Zum leichteren Verständnis des Ablaufs auch hier wieder die grafische Darstellung:

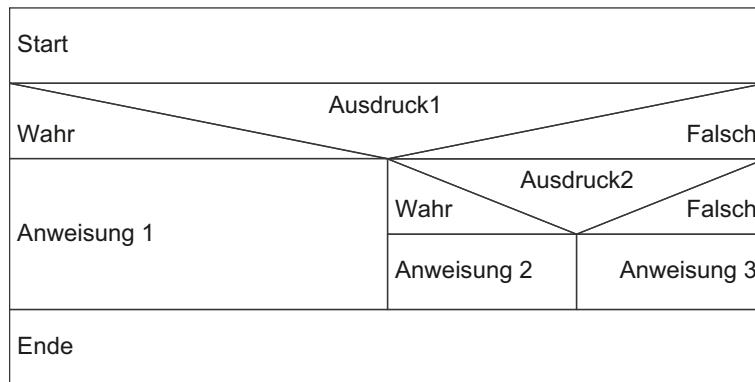


Abb. 2.5: Geschachtelte Verzweigungen mit if ... else



Durch die Kombination von geschachtelten if ... else-Konstruktionen, in denen die verschiedenen Zweige selbst weitere if ... else-Konstruktionen enthalten, lassen sich Entscheidungsketten in einem Programm realisieren.

Sie können die zweite if-Anweisung auch direkt hinter das else schreiben – zum Beispiel so:

```

if (Ausdruck1)
    Anweisung1;
else if (Ausdruck2)
    ...
  
```

Achten Sie dann aber darauf, dass Sie die beiden Schlüsselwörter else und if durch ein Leerzeichen trennen müssen. Ein Schlüsselwort elseif kennt C# nicht.

Tipp:

Achten Sie bei allen if ... else-Konstruktionen auf die Einrückungen. Damit erhöhen Sie die Lesbarkeit des Quelltextes erheblich. Im folgenden Beispiel sind die Einrückungen zum Beispiel nicht ganz korrekt.

```

if (Ausdruck1)
    Anweisung1;
else
    if (Ausdruck2)
        Anweisung2;
    else
        Anweisung3;
  
```

Beim flüchtigen Hinsehen entsteht so der Eindruck, als befände sich die zweite if-Anweisung auf der gleichen Ebene wie die erste if-Anweisung. Tatsächlich aber gehört sie zum else-Zweig der ersten if-Anweisung.

Normalerweise setzt der Editor von Visual Studio die Einrückungen automatisch richtig, sobald Sie eine Zeile mit einem if oder einem else abschließen. Sie können die Einrückungen aber auch selbst mit der Taste  setzen.

Sehen wir uns die Entscheidungsketten mit if ... else jetzt an einem Beispiel an. Der folgende Code ermittelt über geschachtelte Abfragen, welche Zahl eingegeben wurde.

```
/* ##### Entscheidungsketten mit if ... else ##### */
using System;
namespace Cshp03d_02_04
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());

            if (zahl == 5)
                Console.WriteLine("Sie haben die 5 eingegeben.");
            else
                if (zahl < 5)
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben eine Zahl kleiner als 5
eingegeben.");
                else
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");
            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}
```

Code 2.4: Entscheidungsketten mit if ... else

Schauen wir uns an, wie das Programm genau arbeitet. Zuerst müssen Sie wieder eine Zahl eingegeben. Anschließend werden mithilfe der geschachtelten Verzweigung drei Fälle unterschieden.

1. Die Zahl ist gleich 5.
2. Die Zahl ist kleiner als 5.
3. Alles andere – also die Zahl ist größer als 5.

Fall 1 wird durch `if (zahl == 5)` abgedeckt. Sollte `zahl` ungleich 5 sein, geht die erste Verzweigung in ihren `else`-Teil. Dort erfolgt dann die Abfrage für Fall 2 mit `if (zahl < 5)`. Wenn die Zahl auch nicht kleiner als 5 war, geht die zweite Verzweigung in ihren `else`-Teil. Damit wird dann auch Fall 3 abgedeckt.

Zum leichteren Nachvollziehen finden Sie den Ablauf auch noch einmal in der folgenden Abbildung.

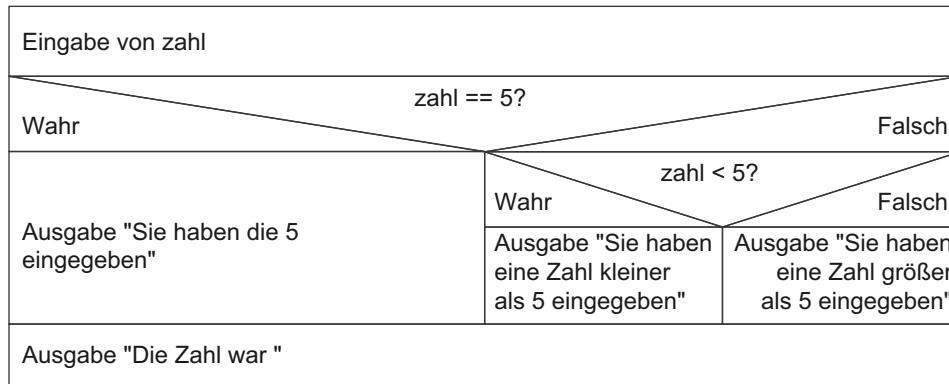


Abb. 2.6: Die Entscheidungskette aus Code 2.4

2.4 Mehrfachauswahl mit `switch ... case`

Wie Sie gerade selbst gesehen haben, sind geschachtelte Verzweigungen recht vielseitig. Allerdings geht schnell der Überblick verloren, wenn Sie nicht sehr sorgfältig arbeiten.

C# kennt aber noch eine Möglichkeit, gezielt auf verschiedene Werte zu reagieren – die Mehrfachauswahl über `switch ... case`. Sie ist wie folgt aufgebaut:

```

switch (Ausdruck)
{
    case Konstante1:
        Anweisung1;
        break;
    case Konstante2:
        Anweisung2;
        break;
    ...
    default:
        Anweisung;
        break;
}
  
```

Grafisch lässt sich die `switch ... case`-Konstruktion zum Beispiel so darstellen:

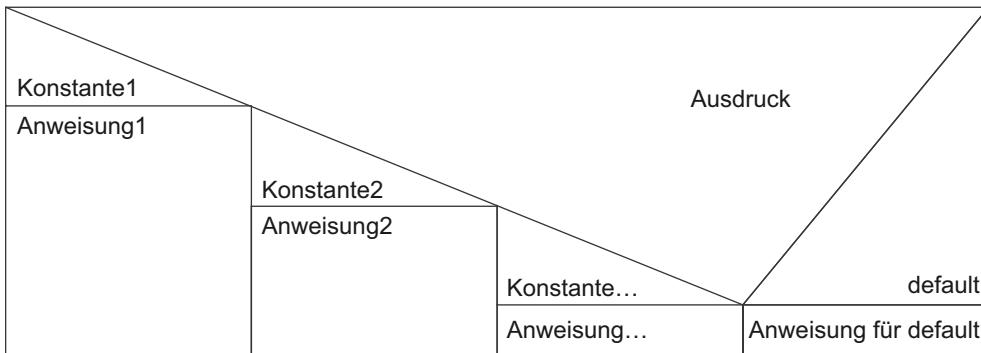


Abb. 2.7: Die `switch ... case`-Konstruktion

Eingeleitet wird die Konstruktion von dem Schlüsselwort `switch`¹. Auf das Schlüsselwort folgt in Klammern entweder ein Wert vom Typ `char`, ein ganzzahliger Typ oder eine Zeichenkette. Einen Typ mit Kommastellen dagegen können Sie nicht verwenden.

Anschließend folgt eine öffnende geschweifte Klammer und es beginnt der **Auswertungsblock**. Im Auswertungsblock stehen die einzelnen Fälle, zwischen denen Sie unterscheiden wollen – die **case-Marken** oder **case-Zweige**.

Jeder Zweig wird dabei einzeln mit dem Schlüsselwort `case`² eingeleitet. Dann folgen eine Konstante und ein Doppelpunkt. Danach kommen die Anweisungen, die in diesem Fall ausgeführt werden sollen. Abgeschlossen werden die Anweisungen für den Zweig durch das Schlüsselwort `break`³ und ein Semikolon.

Ausgeführt wird ein `case`-Zweig, wenn der Ausdruck hinter `switch` mit der Konstante des Zweigs übereinstimmt. Die Ausführung der Anweisungen wird dabei beim nächsten `break` beendet.

Bitte beachten Sie:

Die Anweisungen in den einzelnen `case`-Zweigen müssen nicht von geschweiften Klammern umgeben sein.

Für jede Konstante darf es nur genau einen `case`-Zweig geben. Sie können eine Konstante also nicht mehrfach aufführen.



Bleibt noch das Schlüsselwort `default`⁴ am Ende des Auswertungsblocks. Hier können Sie Anweisungen angeben, die ausgeführt werden, wenn keiner der Fälle davor zutrifft. Der Einsatz von `default` ist optional; das heißt, Sie können `default` auch weglassen. Damit sparen Sie zwar etwas Tipparbeit, laufen aber besonders bei umfangreichen `switch ... case`-Konstruktionen Gefahr, dass das Programm in einen undefinierten Zustand gerät, wenn keiner der `case`-Zweige zutrifft.

1. *Switch* bedeutet übersetzt so viel wie „Schalter“.

2. *Case* bedeutet übersetzt so viel wie „Fall“.

3. *Break* bedeutet übersetzt so viel wie „Unterbrechung“.

4. *Default* bedeutet frei übersetzt so viel wie „Standard“.

Daher sollten Sie den `default`-Zweig vor allem bei Ihren ersten Versuchen immer einbauen – auch wenn Sie hier nur eine Kontrollmeldung ausgeben lassen, dass keiner der `case`-Zweige durchlaufen wurde.

Ein wichtiger Tipp:

Denken Sie an die `break`-Anweisungen am Ende der Zweige. Wenn das `break` fehlt, meldet der Compiler einen Fehler. Achten Sie daher auch darauf, dass Sie nicht nur die `case`-Zweige mit `break` abschließen, sondern auch den `default`-Zweig.

Schauen wir uns die `switch ... case`-Konstruktion jetzt an einem praktischen Beispiel an. Der folgende Code erzeugt eine „Menükarte“ mit drei Einträgen und liefert abhängig von Ihrer Auswahl einen Text zurück.

```
/* #####switch ... case#####
##### * /
```

```
using System;

namespace Cshp03d_02_05
{
    class Program
    {
        static void Main(string[] args)
        {
            char essenWahl;

            Console.WriteLine("Sie haben folgende Auswahl: \n");
            Console.WriteLine("a Schweineschnitzel mit Nudeln");
            Console.WriteLine("b Wiener Schnitzel mit Pommes");
            //bitte in einer Zeile eingeben
            Console.WriteLine("c Vegetarische Hackbällchen mit Reis\n");
            Console.Write("Was möchten Sie essen? ");

            essenWahl = Convert.ToChar(Console.Read());

            //die Auswertung von essenWahl
            switch(essenWahl)
            {
                //der case-Zweig für a
                case 'a':
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben Schweineschnitzel mit Nudeln gewählt!");
                    Console.WriteLine("Kommt sofort!");
                    break;
                //der case-Zweig für b
                case 'b':
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben Wiener Schnitzel mit Pommes gewählt!");
                    Console.WriteLine("Das dauert einen Moment!");
                    break;
            }
        }
    }
}
```

```
//der case-Zweig für c
case 'c':
    //bitte in einer Zeile eingeben
    Console.WriteLine("Sie haben vegetarische Hackbällchen
mit Reis gewählt!");
    Console.WriteLine("Einen Augenblick!");
    break;
//für alles andere
default:
    //bitte in einer Zeile eingeben
    Console.WriteLine("Sie haben keine gültige Auswahl
getroffen!");
    Console.WriteLine("Dann gibt es eben nichts.");
    break;
}
}
}
```

Code 2.5: switch ... case-Konstruktion

Die drei Gerichte werden im Code durch die drei `case`-Zweige unterschieden. Wenn Sie etwas anderes als a, b oder c eingeben, werden die Anweisungen hinter `default` ausgeführt.

Allerdings werden die Anweisungen im `default`-Block auch dann ausgeführt, wenn Sie zum Beispiel A eingeben. Denn aus Sicht des Programms handelt es sich bei a und A um zwei verschiedene Werte – und lediglich für den Wert a ist ja ein `case`-Zweig vorhanden. Sie könnten nun natürlich auch für die entsprechenden Großbuchstaben getrennte `case`-Zweige erstellen und die Anweisungen dann dort noch einmal aufführen. Es geht aber auch sehr viel komfortabler – nämlich mit einem kleinen „Kniff“.

Ergänzen Sie einfach unmittelbar vor dem `case`-Zweig für den Kleinbuchstaben einen `case`-Zweig für den dazugehörigen Großbuchstaben. Diesen Zweig lassen Sie komplett leer – also ohne Anweisungen und auch ohne `break`. Für die Buchstaben A und a könnte das so aussehen:

```
switch (essenWahl)
{
    case 'A':
    case 'a':
        Console.WriteLine("Sie haben Schweineschnitzel mit Nudeln gewählt!");
        Console.WriteLine("Kommt sofort!");
        break;
    case 'b':
        ...
}
```

Wenn der Buchstabe A eingegeben wird, beginnt das Programm mit der Bearbeitung bei der entsprechenden `case`-Marke – also `case 'A':`. Dann werden alle Anweisungen bis zum nächsten `break` ausgeführt – also die Anweisungen, die zur `case`-Marke `case 'a':` gehören.



Dieser „Kniff“ funktioniert nur dann, wenn die `case`-Marken direkt aufeinanderfolgen.

Auch `switch ... case`-Konstruktionen lassen sich schachteln. Das ist zum Beispiel dann interessant, wenn Sie einen Hauptfall in weitere Unterfälle untergliedern wollen. Im folgenden Code werden zum Beispiel für zwei Gerichte noch Beilagen angeboten. Außerdem haben wir den Code so erweitert, dass jetzt auch Großbuchstaben berücksichtigt werden.

```
/* ##### geschachtelte switch ... case-Konstruktion ##### */
using System;
namespace Cshp03d_02_06
{
    class Program
    {
        static void Main(string[] args)
        {
            char essenWahl, beilagenWahl;

            Console.WriteLine("Sie haben folgende Auswahl: \n");
            Console.WriteLine("a Schweineschnitzel");
            Console.WriteLine("b Wiener Schnitzel\n");
            Console.Write("Was möchten Sie essen? ");

            essenWahl = Convert.ToChar(Console.Read());
            //den Tastaturpuffer leeren
            Console.ReadLine();

            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie können folgende Beilagen wählen:
            \n");
            Console.WriteLine("c Pommes");
            Console.WriteLine("d Reis\n");
            Console.Write("Welche Beilage möchten Sie? ");

            beilagenWahl = Convert.ToChar(Console.Read());

            //die Auswertung von essenWahl
            switch (essenWahl)
            {
                case 'A':
                case 'a':
```

```
//die Auswertung von beilagenWahl
switch (beilagenWahl)
{
    case 'C':
    case 'c':
        //bitte in einer Zeile eingeben
        Console.WriteLine("Sie haben Schweineschnitzel mit
        Pommes gewählt!");
        break;
    case 'D':
    case 'd':
        //bitte in einer Zeile eingeben
        Console.WriteLine("Sie haben Schweineschnitzel mit
        Reis gewählt!");
        break;
    default:
        Console.WriteLine("Diese Beilage gibt es nicht.");
        break;
} //hier endet die Auswertung von beilagenWahl
break; //hier enden die case-Marken für a und A
case 'B':
case 'b':
    //die Auswertung von beilagenWahl
    switch (beilagenWahl)
    {
        case 'C':
        case 'c':
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben Wiener Schnitzel mit
            Pommes gewählt!");
            break;
        case 'D':
        case 'd':
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben Wiener Schnitzel mit
            Reis gewählt!");
            break;
        default:
            Console.WriteLine("Diese Beilage gibt es nicht.");
            break;
    } //hier endet die Auswertung von beilagenWahl
    break; //hier enden die case-Marken für b und B
default: //für die Auswertung von essenWahl
    //bitte in einer Zeile eingeben
    Console.WriteLine("Dieses Gericht steht nicht auf der
    Karte.");
    break;
} //hier endet die Auswertung von essenWahl
}
```

Code 2.6: Verschachteltes switch ... case

Damit Sie die verschiedenen Ebenen leichter nachvollziehen können, haben wir den Anfang und das Ende jeweils mit Kommentaren gekennzeichnet. Welche verschiedenen Fälle das Programm abdeckt, zeigt Ihnen die folgende Abbildung:

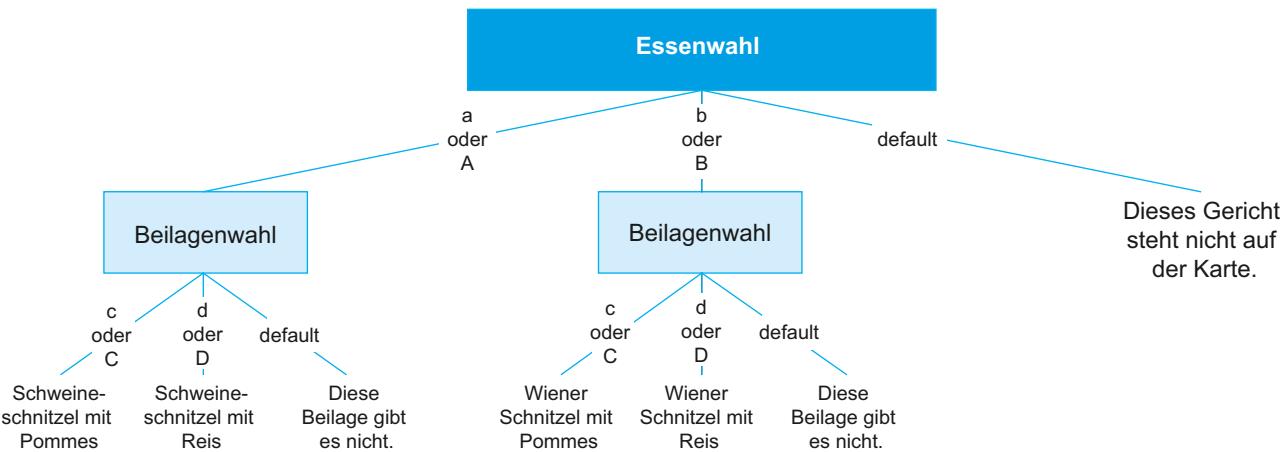


Abb. 2.8: Darstellung der möglichen Fälle aus Code 2.6

Genau wie verschachtelte `if ... else`-Konstruktionen können auch geschachtelte `switch ... case`-Konstruktionen sehr schnell unübersichtlich werden. Versuchen Sie daher, die Schachtelungen auf so wenig Ebenen wie eben möglich zu beschränken.

Zum Schluss dieses Kapitels fassen wir noch einmal ein paar wichtige Hinweise zur `if`-Konstruktion und zur `switch ... case`-Konstruktion zusammen:

- Wenn Sie eine Mehrfachauswahl programmieren wollen, können Sie die `switch ... case`-Konstruktion nur dann verwenden, wenn Sie mit Werten arbeiten, die sich durch eine ganze Zahl, ein Zeichen oder eine Zeichenkette darstellen lassen. Bei Zahlen mit Kommastellen müssen Sie geschachtelte `if ... else`-Verzweigungen benutzen.
- Denken Sie in einer `switch ... case`-Konstruktion an das `break` in den `case`-Zweigen und dem `default`-Zweig!
- Achten Sie peinlich genau darauf, dass die Konstanten in den `case`-Zweigen zum Typ des Ausdrucks bei `switch` passen. Wenn Sie zum Beispiel ganze Zahlen verarbeiten, dürfen Sie diese Zahlen **nicht** mit Apostrophen umfassen. Der Compiler behandelt dann nämlich die Konstanten der `case`-Marken als Zeichen und es gibt keine passende `case`-Marke – egal, was Sie eingeben.

Das folgende Codefragment führt zum Beispiel immer die Anweisungen im `default`-Zweig aus – auch dann, wenn Sie 1 oder 2 eingeben:

```

int zahlWahl;
Console.WriteLine("Geben Sie eine Zahl ein: ");
zahlWahl = Convert.ToInt32(Console.ReadLine());
switch(zahlWahl)
{
    case '1':
        Console.WriteLine("Sie haben 1 eingegeben!");
        break;
  
```

```
case '2':  
    Console.WriteLine("Sie haben 2 eingegeben!");  
    break;  
default:  
    Console.WriteLine("Sie haben weder 1 noch 2 eingegeben!");  
    break;  
}
```

- Achten Sie sehr genau auf die Klammern bei geschachtelten `if ... else`-Konstruktionen. Verwenden Sie lieber zu viele Klammern als zu wenig. Besonders bei starken Schachtelungen besteht gerade zu Beginn die Gefahr, dass Sie den Überblick verlieren. Klammern Sie daher am Anfang am besten jeden Zweig – auch wenn es eigentlich nicht zwingend erforderlich ist.

Denken Sie aber bitte daran: Der Compiler überprüft lediglich die Syntax Ihrer Programme. Das heißt, er zählt mit, ob Sie für jede öffnende Klammer auch eine schließende verwenden. Ob Sie die Klammern logisch an die richtige Stelle setzen, kann der Compiler nicht prüfen.

- Achten Sie bei `if ... else`-Verzweigungen genau darauf, zu welchem `if` ein `else`-Teil gehört.

Zusammenfassung

Mit einer `if`-Verzweigung können Sie Anweisungen abhängig von einer Bedingung ausführen lassen.

Über den `else`-Zweig können Sie bei `if` Anweisungen angeben, die ausgeführt werden sollen, wenn die Bedingung **nicht** zutrifft.

Sowohl im `if`-Zweig als auch im `else`-Zweig können Sie mehrere Anweisungen mit den Klammern `{ }` zu einem Anweisungsblock zusammenfassen.

Für Mehrfachauswahlen bei ganzzahligen Werten, Zeichen oder Zeichenketten können Sie die `switch ... case`-Konstruktion verwenden.

Sowohl die `if`-Anweisung als auch die `switch ... case`-Konstruktion können Sie schachteln.

Aufgaben zur Selbstüberprüfung

- 2.1 Schreiben Sie ein Programm, das zwei ganze Zahlen x und y über die Tastatur einliest und anschließend ausgibt, ob x größer ist als y oder umgekehrt. Benutzen Sie dafür eine `if ... else`-Verzweigung.

- 2.2 Ergänzen Sie die `switch ... case`-Konstruktion in dem folgenden Quelltext. Geben Sie bitte in den Lücken die jeweils erforderliche Anweisung an. Ein `default`-Zweig ist in dem Programm nicht vorgesehen.

```
/* #####switch ... case mit Lücken#####
#####using System;
namespace LoesII2
{
    class Program
    {
        static void Main(string[] args)
        {
            int essenWahl;

            Console.WriteLine("Wählen Sie bitte ein Essen aus:
\n");
            Console.WriteLine("1 Jägerschnitzel mit Pommes");
            Console.WriteLine("2 Currywurst mit Pommes");
            Console.WriteLine("3 Bratwurst mit Brötchen\n");
            Console.WriteLine("Was möchten Sie essen? ");

            essenWahl = Convert.ToInt32(Console.ReadLine());

            switch _____
            {
                _____
```

```
Console.WriteLine("Sie haben ein Jägerschnitzel  
mit Pommes gewählt.");  
_____;  
  
Console.WriteLine("Sie haben eine Currywurst mit  
Pommes gewählt.");  
  
case _____ :  
    Console.WriteLine("Sie haben eine Bratwurst mit  
Brötchen gewählt.");  
  
    _____  
}  
}  
}
```

3 Schleifen

*Im letzten Kapitel haben Sie gelernt, wie Sie Verzweigungen und Mehrfachauswahlen in Ihren Programmen benutzen. In diesem Kapitel werden Sie ein weiteres wichtiges Konstrukt zum Erstellen anspruchsvoller Programme kennenlernen: die **Schleifen**.*



Mit Schleifen können Sie Teile eines Programms wiederholen lassen. Dabei führt der Rechner die Anweisungen im **Schleifenkörper** wiederholt in Abhängigkeit von **Bedingungen** aus. Die Bedingungen stehen entweder am Anfang der Schleife im **Schleifenkopf** oder am Ende der Schleife im **Schleifenfuß**.

Hinweis:

Die Bedingungen, die die Ausführung einer Schleife steuern, werden auch **Abbruchbedingungen** genannt. C# wertet Abbruchbedingungen allerdings genau andersherum aus, als es dem „normalen“ Sprachgebrauch entspricht. Wenn die Abbruchbedingung wahr ist, wird die Schleife ausgeführt. Ist die Abbruchbedingung dagegen falsch, wird die Schleife nicht ausgeführt. Wir benutzen daher in diesem Kapitel einfach den Ausdruck „Bedingung“.

Grundsätzlich unterscheidet C# drei Schleifentypen:

- die **kopfgesteuerte Schleife**, bei der die Bedingung am Anfang steht,
- die **fußgesteuerte Schleife**, bei der die Bedingung am Ende steht, und
- die **Zählschleife**, einen Sonderfall der kopfgesteuerten Schleife.

Wir werden im Verlauf des Kapitels nacheinander auf diese Schleifentypen eingehen.

Außerdem werden Sie zwei Befehle zur Schleifensteuerung kennenlernen: `break` und `continue`.

Bevor Sie sich mit den Schleifen beschäftigen, noch ein wichtiger Hinweis:

Übernehmen Sie die Programme dieses Kapitels sehr sorgfältig. Bei Fehlern in einer Schleife droht eine **Endlosschleife**. Das heißt, das Programm wiederholt die Schleifenanweisungen immer wieder und kann nicht korrekt beendet werden. Sie müssen es dann „gewaltsam“ abbrechen – zum Beispiel mit der Tastenkombination **Strg + C**. Im schlimmsten Fall bleibt Ihnen bei einer Endlosschleife nichts anderes übrig, als den Rechner neu zu starten. Achten Sie deshalb vor allem bei den Bedingungen sorgfältig darauf, dass Ihnen keine Fehler unterlaufen.

Schauen wir uns jetzt die verschiedenen Schleifen in C# der Reihe nach an. Beginnen wir mit der kopfgesteuerten Schleife mit `while`.

3.1 Kopfgesteuerte Schleife mit while

Die kopfgesteuerte Schleife mit while ist die einfachste und allgemeinste Schleife von C#.

Sie hat die folgende Syntax:

```
while (Ausdruck) //Schleifenkopf  
    Anweisung;     //Schleifenkörper
```

Zuerst wird der Ausdruck im Schleifenkopf ausgewertet. Wenn er wahr ist, wird die Anweisung im Schleifenkörper ausgeführt. Danach springt das Programm wieder zum Schleifenkopf und der Ausdruck wird erneut ausgewertet. Dieser Prozess wiederholt sich so lange, bis der Ausdruck falsch wird. Dann wird das Programm mit der nächsten Anweisung hinter dem Schleifenkörper fortgesetzt.

Bitte beachten Sie:

Hinter dem Schleifenkopf dürfen Sie **kein** Semikolon setzen. Damit trennen Sie den Schleifenkopf von der eigentlichen Schleife und es wird nur noch der Schleifenkopf ausgeführt. Da sich dabei aber der Ausdruck nicht verändern kann, wird der Schleifenkopf endlos wiederholt.



Grafisch lässt sich die while-Schleife so darstellen:



Abb. 3.1: Die while-Schleife

Merken Sie sich für die while-Schleife:

Die Schleife wird so lange ausgeführt, wie der Ausdruck im Schleifenkopf wahr ist.



Beachten Sie bitte, dass sich bei der while-Schleife die Reihenfolge bei der Ausführung der Anweisungen verändert. Bisher wurden in unseren Programmen immer alle Anweisungen von „oben“ nach „unten“ abgearbeitet. Jetzt tritt zum ersten Mal der Fall ein, dass nach einer Anweisung mit einer Anweisung weitergearbeitet wird, die sich im Quelltext vor der ausgeführten Anweisung befindet – eben mit der Überprüfung der Bedingung für die Schleife.

Wie schon bei der `if`-Verzweigung können Sie auch bei der `while`-Schleife mehrere Anweisungen in einem Anweisungsblock zusammenfassen. Dann sieht die `while`-Schleife so aus:

```
while (Ausdruck) //Schleifenkopf
{
    //Anfang des Schleifenkörpers
    Anweisung1;
    Anweisung2;
    ...
}
//Ende des Schleifenkörpers
```

Sehen wir uns die `while`-Schleife jetzt in einem Programm an. Der folgende Code zählt auf dem Bildschirm bis 10.

```
/* #####
while-Schleife
#####
*/
using System;

namespace Cshp03d_03_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int schleifenVariable = 0;

            //die Schleife
            while (schleifenVariable <= 10)
            {
                //der aktuelle Wert wird ausgegeben
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert {0}",
                    schleifenVariable);
                //schleifenVariable wird erhöht
                schleifenVariable++;
            }
        }
    }
}
```

Code 3.1: while-Schleife

Schauen wir uns die Schleife im Detail an:

Im Schleifenkopf `while (schleifenVariable <= 10)` wird der Ausdruck `schleifenVariable <= 10` ausgewertet. Solange dieser Ausdruck wahr ist, werden die Anweisungen im Schleifenkörper ausgeführt. In unserem Beispiel läuft die Schleife also, bis `schleifenVariable` den Wert 11 erreicht hat.



Eine Variable, die im Ausdruck einer Schleife überprüft wird, wird auch **Schleifenvariable** genannt.

Im Schleifenkörper selbst werden zwei Anweisungen ausgeführt:

```
Console.WriteLine("Die Variable hat jetzt den Wert {0}",  
    schleifenVariable);  
  
    schleifenVariable++;
```

Die erste Anweisung gibt den aktuellen Wert von `schleifenVariable` aus. In der zweiten Anweisung wird der Wert von `schleifenVariable` bei jedem Durchlauf um den Wert 1 erhöht.

Zwei Sachen sind bei einer `while`-Schleife sehr wichtig:

- 1) Sie müssen sicherstellen, dass die Schleifenvariable korrekt initialisiert ist – also den richtigen Wert zugewiesen bekommt. Wenn Sie zum Beispiel `schleifenVariable` im vorigen Code mit dem Wert 1 initialisieren, erfolgt ein Schleifendurchlauf weniger.
- 2) Sie müssen den Wert der Schleifenvariablen innerhalb der Schleife verändern. Das übernimmt in unserem Beispiel die Anweisung

```
    schleifenVariable++;
```

im Schleifenkörper.

Wenn Sie diese Veränderung nicht vornehmen, ergibt die Auswertung der Bedingung im Schleifenkopf immer wieder denselben Wert – und damit hätten Sie eine Endlosschleife konstruiert.

Probieren Sie das ruhig einmal aus. Kommentieren Sie die Anweisung zum Verändern von `schleifenVariable` aus und lassen Sie das Programm noch einmal ausführen. Denken Sie vor dem Start bitte daran, alle noch nicht gespeicherten Daten zu sichern. Sie werden sehen, das Programm läuft so lange weiter, bis Sie es mit der Tastenkombination **Strg** + **C** abbrechen.

Testen Sie auch einmal, was geschieht, wenn Sie hinter den Schleifenkopf im vorigen Code ein Semikolon setzen. Nach dem Start des Programms erscheint dann nur ein leeres Fenster für die Eingabeaufforderung mit einer blinkenden Einfügemarkierung. Scheinbar passiert also gar nichts. Tatsächlich aber läuft die Schleife endlos – allerdings wird lediglich der Ausdruck immer wieder überprüft. Wenn Sie genau hinsehen, erkennen Sie auch, dass der Compiler Ihnen eine entsprechende Warnung anzeigt – nämlich zu einer falschen leeren Anweisung.

So viel zur kopfgesteuerten Schleife mit `while`.

3.2 Fußgesteuerte Schleife mit `do ... while`

Die `do ... while`-Schleife arbeitet ähnlich wie die `while`-Schleife. Allerdings erfolgt die Auswertung der Bedingung erst im Schleifenfuß – daher auch der Name **fußgesteuerte Schleife**.

Die Syntax der `do ... while`-Schleife sieht so aus:

```
do          //Schleifenkopf  
    Anweisung; //Schleifenkörper  
  
    while (Ausdruck); //Schleifenfuß
```

Zuerst wird die Anweisung im Schleifenkörper ausgeführt. Anschließend wird der Ausdruck im Schleifenfuß geprüft. Wenn dieser Ausdruck wahr ist, wird die Anweisung im Schleifenkörper wiederholt, andernfalls wird die Anweisung nach dem Schleifenfuß ausgeführt.



Bitte beachten Sie:

Bei einer `do ... while`-Schleife **müssen** Sie den Schleifenfuß mit einem Semikolon abschließen. Hier ist ja die gesamte Schleife zu Ende. Wenn Sie das Semikolon weglassen, meldet der Compiler einen Syntaxfehler.

Da die Überprüfung der Bedingung erst am Ende der Schleife erfolgt, wird der Schleifenkörper bei der `do ... while`-Schleife – anders als bei der `while`-Schleife – mindestens einmal durchlaufen. Eine fußgesteuerte Schleife sollten Sie daher immer dann einsetzen, wenn Sie sichergehen wollen, dass die Schleife mindestens einmal abgearbeitet wird.



Merken Sie sich für die do ... while-Schleife:

Die Schleife wird in jedem Fall mindestens einmal durchlaufen.

Grafisch lässt sich die `do ... while`-Schleife so darstellen:

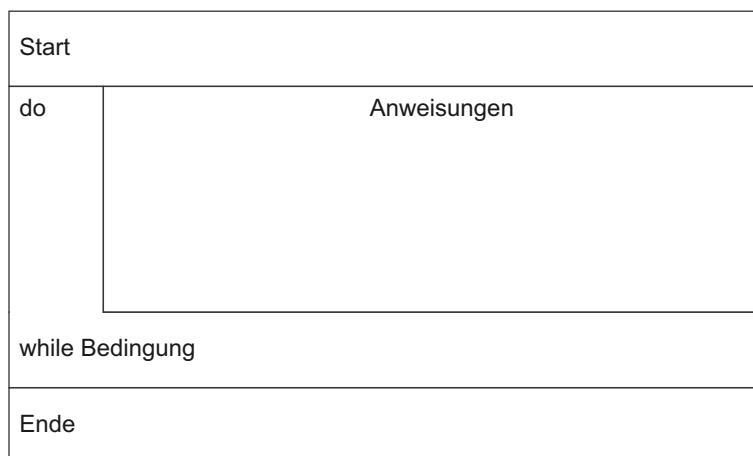


Abb. 3.2: Die `do ... while`-Schleife

Auch im Schleifenkörper der `do ... while`-Schleife können Sie einen Anweisungsblock verwenden. Dann sieht die Syntax so aus:

```
do           //Schleifenkopf
{
           //Anfang des Schleifenkörpers
    Anweisung1;
    Anweisung2;
    ...
} while(Ausdruck); //Ende des Schleifenkörpers und Schleifenfuß
```

Schauen wir uns auch die `do ... while`-Schleife an einem Beispiel an. Im folgenden Code werden so lange Zahlen eingelesen, bis Sie einen Wert größer als 10 eingeben.

```
/* ##### do ... while-Schleife ##### */
do ... while-Schleife
##### */

using System;

namespace Cshp03d_03_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int schleifenVariable = 0;

            do
            {
                //bitte in einer Zeile eingeben
                Console.Write("Bitte geben Sie einen Wert größer als 10
ein. ");
                schleifenVariable = Convert.ToInt32(Console.ReadLine());
            } while (schleifenVariable <= 10);
            Console.WriteLine("Danke.");
        }
    }
}
```

Code 3.2: do ... while-Schleife

Im Schleifenkörper finden Sie eine Aufforderung zur Eingabe einer Zahl und die Anweisung zum Einlesen der Zahl. Diese Anweisungen werden so lange ausgeführt, bis der Ausdruck `schleifenVariable <= 10` falsch wird – also bis eine Zahl größer als 10 eingegeben wird.

Die Veränderung der Variablen `schleifenVariable` erfolgt in unserem Beispiel durch das Einlesen des Wertes innerhalb der Schleife. Damit ist also sichergestellt, dass keine Endlosschleife entstehen kann – außer der Anwender gibt immer wieder einen Wert kleiner als 11 ein.

Da die Schleife im vorigen Code mindestens einmal durchlaufen wird, könnten Sie hier auch auf die Initialisierung der Schleifenvariablen vor der Schleife verzichten. Die Initialisierung würde ja beim ersten Durchlauf der Schleife erfolgen. Damit wäre auch sichergestellt, dass die Schleifenvariable bei der ersten Auswertung des Ausdrucks im Schleifenfuß einen definierten Wert hat.

Grundsätzlich lässt sich jede kopfgesteuerte Schleife in eine fußgesteuerte ändern und umgekehrt. Beim vorigen Code ist diese Änderung ganz einfach. Sie können die `while`-Anweisung, die im Schleifenfuß steht, in den Schleifenkopf schreiben und lassen das `do` weg. Denken Sie dabei bitte aber auch daran, das Semikolon hinter dem Ausdruck bei `while` zu löschen.

Weitere Änderungen sind nicht nötig, weil die Variable `schleifenVariable` zu Beginn mit 0 initialisiert wird und der Ausdruck `schleifenVariable <= 10` damit wahr ist. Der Schleifenkörper wird also in jedem Fall mindestens einmal durchlaufen. Der vorige Code sieht mit einer kopfgesteuerten Schleife so aus:

```
/*
#####
Die umgebaute Schleife
#####
*/
using System;

namespace Cshp03d_03_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int schleifenVariable = 0;

            //der Fuß wird jetzt zum Kopf
            //Denken Sie daran, das Semikolon zu löschen!
            while (schleifenVariable <= 10)
            {
                //bitte in einer Zeile eingeben
                Console.Write("Bitte geben Sie einen Wert größer als 10
ein. ");
                schleifenVariable = Convert.ToInt32(Console.ReadLine());
            }
            Console.WriteLine("Danke.");
        }
    }
}
```

Code 3.3: Die umgebaute Schleife

Auch die `while`-Schleife aus dem Code 3.1 lässt sich durch eine `do ... while`-Schleife ersetzen.

Bevor Sie weiterlesen ...

Versuchen Sie erst einmal selbst, den Umbau durchzuführen.

Der Code sieht dann so aus:

```
/*
#####
Noch eine umgebaute Schleife
#####
*/
using System;

namespace Cshp03d_03_04
{
    class Program
    {
```

```
static void Main(string[] args)
{
    int schleifenVariable = 0;

    do
    {
        //bitte in einer Zeile eingeben
        Console.WriteLine("Die Variable hat jetzt den Wert {0}",
        schleifenVariable);
        schleifenVariable++;
    } while (schleifenVariable <= 10);
}
```

Code 3.4: Noch eine umgebaute Schleife

Ob Sie nun eine `while`- oder `do ... while`-Schleife verwenden, ist in vielen Fällen auch eine Frage des persönlichen Geschmacks. `do ... while`-Schleifen werden aber zum Beispiel häufig für das Einlesen von Werten verwendet, da hier ja die Eingabe des Anwenders bereits vor der Überprüfung der Bedingung zur Verfügung steht.

So viel an dieser Stelle zur `do ... while`-Schleife. Am Ende dieses Kapitels werden wir diese Schleifenform noch einmal in einem Exkurs verwenden und Ihnen zeigen, wie Sie Eingabefehler abfangen können.

3.3 Zählschleife mit `for`

Mit der kopfgesteuerten `while`- und der fußgesteuerten `do ... while`-Schleife können Sie eigentlich bereits alle Aufgaben umsetzen, die den Einsatz einer Schleife erfordern. Trotzdem gibt es in C# noch eine dritte Variante: die Zählschleife mit `for`. Sie kommt vor allem dann zum Einsatz, wenn Sie bereits vor dem Ausführen der Schleife wissen, wie oft die Anweisungen wiederholt werden müssen.

Das Grundprinzip einer `for`-Schleife unterscheidet sich nicht wesentlich von den beiden anderen Schleifenformen. Die Schleife wird ebenfalls so lange ausgeführt, bis eine Bedingung falsch ist. Die Syntax ist allerdings wesentlich kompakter. Ein Beispiel:

```
for (i = 0; i <= 10; i++)
    Anweisung;
```

Hinter dem Schlüsselwort `for` stehen in Klammern drei Ausdrücke, die das Verhalten der Schleife steuern.

Der **erste Ausdruck** `i = 0`, ist der **Vorlauf**. Er wird einmal vor Beginn der Schleife ausgeführt. In unserem Beispiel wird hier die Schleifenvariable `i` mit dem Wert 0 initialisiert.

Der **zweite Ausdruck** `i <= 10`, ist die **Bedingung** oder der **Testausdruck**. Er wird vor Beginn jedes Schleifendurchlaufs überprüft. Ergibt der Ausdruck falsch, wird die Ausführung der Schleife abgebrochen. In unserem Beispiel wird die Schleife also so lange ausgeführt, bis `i` den Wert 11 hat.

Der **dritte Ausdruck** `i++` wird als **Nachlauf** bezeichnet. Er wird am Ende jedes Schleifendurchlaufs ausgeführt. In unserem Beispiel wird die Schleifenvariable `i` also bei jedem Durchlauf um den Wert 1 erhöht.



Bitte beachten Sie:

Die drei Ausdrücke müssen jeweils durch ein Semikolon voneinander getrennt werden.

Die Anweisung, die die Schleife ausführen soll, steht erst **hinter** diesen drei Ausdrücken. Dabei können Sie – wie gewohnt – auch wieder mehrere Anweisungen in einem Anweisungsblock zusammenfassen.

Grafisch lässt sich die `for`-Schleife so darstellen:

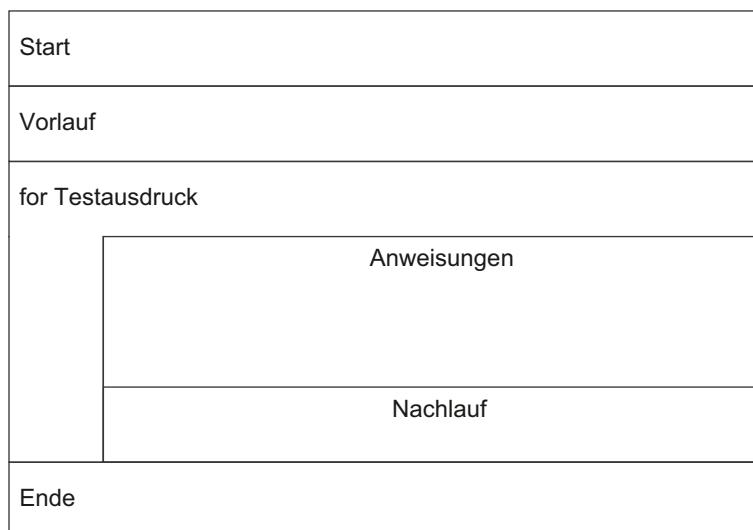


Abb. 3.3: Die `for`-Schleife



Bitte beachten Sie:

Die drei Ausdrücke in einer `for`-Schleife werden nicht gleichberechtigt abgearbeitet – auch wenn sie alle zusammen in einer Zeile stehen.

Der Vorlauf wird lediglich **einmal** bearbeitet – zu Beginn der Schleife. Bei der eigentlichen Ausführung der Schleife wird er **nicht** wiederholt.

Der Testausdruck wird **vor** jedem Schleifendurchlauf überprüft.

Der Nachlauf wird **nach** den Anweisungen der Schleife durchgeführt – obwohl er im Quelltext **vor** den Anweisungen steht.

Schauen wir uns die `for`-Schleife jetzt in einem C#-Programm an. Es gibt die Zahlen von 0 bis 10 auf dem Bildschirm aus:

```
/* #####for-Schleife#####
##### */
using System;

namespace Cshp03d_03_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;

            for(i = 0; i <= 10; i++)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert
{0}",i);
        }
    }
}
```

Code 3.5: for-Schleife

Dieser Code hat exakt dieselbe Wirkung wie Code 3.1. Damit Sie die beiden Codes leichter vergleichen können, drucken wir Code 3.1 hier noch einmal mit einigen Kommentaren ab und ändern außerdem den Namen der Schleifenvariablen in `i`.

```
/* #####Von 0 bis 10 mit while#####
##### */
using System;

namespace Cshp03d_03_06
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;          //der Vorlauf
            while (i <= 10)   //der Testausdruck
            {
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert
{0}",i);
                i++;           //der Nachlauf
            }
        }
    }
}
```

Code 3.6: Von 0 bis 10 mit while

Der Vorlauf `i = 0;` steht im vorigen Code vor der Schleife und wird einmal vor Schleifenbeginn ausgeführt. Der Testausdruck `i <= 10` wird vor jedem Schleifendurchlauf überprüft, weil er im Schleifenkopf der `while`-Schleife steht. Der Nachlauf `i++;` wird am Ende nach jedem Schleifendurchlauf ausgeführt.

Im direkten Vergleich ist die Version mit `while` allerdings etwas länger und auch unübersichtlicher als die Version mit der `for`-Schleife. Denn im Code 3.5 wird ja allein durch einen kurzen Blick auf die Zeile mit `for` klar, wie die Schleifenvariable initialisiert wird, wie oft die Schleife läuft und wie die Veränderung der Schleifenvariable erfolgt. Beim Code 3.6 dagegen müssen Sie sich diese Informationen an mehreren Stellen zusammensuchen.

Code 3.5 lässt sich sogar noch etwas kompakter gestalten. Sie können die Schleifenvariable auch direkt im Vorlauf vereinbaren. Damit sparen Sie sich die Anweisung `int i;`. Der veränderte Code sieht dann so aus:

```
/*
#####
Ein sehr kompakte for-Schleife
#####
*/
using System;
namespace Cshp03d_03_07
{
    class Program
    {
        static void Main(string[] args)
        {
            for(int i = 0; i <= 10; i++)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert
{0}", i);
        }
    }
}
```

Code 3.7: Eine sehr kompakte `for`-Schleife

Bitte beachten Sie aber, dass die Schleifenvariable `i` jetzt nur noch in der Schleife bekannt ist. Wenn Sie zum Beispiel versuchen, den aktuellen Wert von `i` außerhalb der Schleife auszugeben, erhalten Sie eine Fehlermeldung, dass der Name `i` im aktuellen Kontext nicht vorhanden ist. Warum diese Fehlermeldung erscheint, erfahren Sie später, wenn wir uns mit dem Gültigkeitsbereich von Variablen beschäftigen.

Sie können eine `for`-Schleife auch in die andere Richtung arbeiten lassen – sozusagen von oben nach unten. Das folgende Beispiel zählt von 100 so lange abwärts, bis die Schleifenvariable den Wert 0 hat:

```
for (int i = 100; i != 0; i--)
```

Bitte denken Sie daran, dass Schleifen so lange ausgeführt werden, **wie** die Bedingung **wahr** ist. Daher muss die Bedingung in dem Beispiel auch `i != 0` lauten. Wenn Sie hier nämlich die Bedingung `i == 0` verwenden, ist der Ausdruck direkt falsch, da ja `i` im Vorlauf auf den Wert 100 gesetzt wird. Die Anweisungen in der Schleife würden dann also gar nicht ausgeführt.

Bevor wir uns jetzt mit den Anweisungen `break` und `continue` in Schleifen beschäftigen, noch einige Hinweise.

Sie können die Bedingungen von Schleifen auch komplett mit Variablen oder mit einer Mischung aus Variablen und konstanten Werte formulieren – zum Beispiel:

```
while (a != i)
```

oder

```
while (a != i * 2)
```

Achten Sie aber in diesem Fall besonders sorgfältig darauf, dass die Bedingungen auch gültig sind. Andernfalls droht eine Endlosschleife.

Bei einer `for`-Schleife gehen viele Programmierer davon aus, dass die gesamte Steuerung der Schleife ausschließlich über die drei Ausdrücke hinter dem `for` erfolgt. Sie sollten daher die Schleifenvariable in einer `for`-Schleife nicht noch einmal innerhalb der Schleife verändern. Damit sorgen Sie nämlich unter Umständen bei anderen Programmierern, die Ihren Quelltext lesen, für sehr viel Verwirrung. Das folgende Beispiel ist zwar syntaktisch korrekt, gilt aber als schlechter Programmierstil:

```
for (int i = 0; i <= 10; i++)
{
    Console.WriteLine("Die Variable hat jetzt den Wert {0}", i);
    //hier wird i jetzt noch einmal in der Schleife verändert
    i = i + 2;
}
```

Beim Lesen der Zeile

```
for (int i = 0; i <= 10; i++)
```

werden die meisten Programmierer davon ausgehen, dass die Schleife exakt 11 Mal durchlaufen wird. Tatsächlich erfolgen aber nur 4 Durchläufe, da die Schleifenvariable `i` ja zusätzlich noch innerhalb der Schleife verändert wird. Sie sollten daher auf solche Konstruktionen bei `for`-Schleifen verzichten oder zumindest durch einen Kommentar im Quelltext auf das Verändern der Schleifenvariablen in der Schleife selbst hinweisen.

3.4 Die Anweisungen `break` und `continue` bei Schleifen

Neben der Steuerung von Schleifen über die Bedingung können Sie den Schleifenablauf auch noch durch die Anweisungen `break` und `continue` steuern.

Schauen wir uns zuerst die Anweisung `break` an.

3.4.1 break

Sie kennen diese Anweisung ja bereits von der `switch ... case`-Konstruktion. Dort markiert sie das Ende eines `case`-beziehungsweise `default`-Zweiges. Bei Schleifen wird `break` eingesetzt, um die Ausführung der Schleife komplett zu beenden. Nach einem `break` in einer Schleife werden also keine weiteren Durchläufe mehr durchgeführt und die Ausführung des Programms wird mit der nächsten Anweisung hinter dem Schleifenkörper fortgesetzt.

Sinnvoll ist die Anweisung `break` in einer Schleife allerdings nur, wenn Sie die Ausführung mit einer Bedingung koppeln. Sonst würde die Schleife ja sofort beim ersten Durchlauf abgebrochen und wäre damit überflüssig. Denkbar sind aber zum Beispiel Konstruktionen wie die folgende:

```
while (Ausdruck1)
{
    ...
    if (Ausdruck2)
        break;
    ...
}
```

Sobald `Ausdruck2` bei einem Schleifendurchlauf wahr wird, wird die Schleife über das `break` abgebrochen.

Sehen wir uns den Einsatz von `break` jetzt an einem praktischen Beispiel an. Im folgenden Code werden zehn Zahlen in einer Schleife eingelesen und zusammengezählt. Bei der Eingabe von 0 wird die Verarbeitung vorzeitig abgebrochen.

```
/* #####
break in einer Schleife
#####
*/
using System;

namespace Cshp03d_03_08
{
    class Program
    {
        static void Main(string[] args)
        {
            int summe, schleifenVariable, eingabe;
            //Initialisierung
            summe = 0;
            schleifenVariable = 1;

            while (schleifenVariable <= 10)
            {
                //bitte in einer Zeile eingeben
                Console.Write("Geben Sie die {0}. Zahl ein: ",
                schleifenVariable);
                eingabe = Convert.ToInt32(Console.ReadLine());
            }
        }
    }
}
```

```
//wenn 0 eingegeben wurde, wird die Schleife abgebrochen
if (eingabe == 0)
    break;
//die Summe durch Zusammenrechnen bilden
summe = eingabe + summe;
schleifenVariable++;
}

Console.WriteLine("Das Einlesen ist beendet.");
Console.WriteLine("Die Summe der Zahlen ist {0}.",summe);
}
}
```

Code 3.8: break in einer Schleife

Da dieser Code wieder etwas länger ist, schauen wir uns die wichtigsten Anweisungen der Reihe nach an.

Zuerst vereinbaren wir drei Variablen vom Typ `int` und initialisieren die Variablen `summe` und `schleifenVariable`. Die Schleifenvariable erhält dabei den Wert 1, damit wir später in der Schleife die Anzeige der einzugebenden Zahl synchron zum Schleifendurchlauf gestalten können. Die Variable `eingabe` dagegen müssen wir nicht initialisieren, da sie ja beim Einlesen einen Wert erhält.

Danach folgt dann die eigentliche Schleife. Hier geben wir zunächst einen Text aus und lesen dann eine Zahl ein. Das übernehmen die beiden Anweisungen

```
Console.Write("Geben Sie die {0}. Zahl ein: ",
schleifenVariable);
eingabe = Convert.ToInt32(Console.ReadLine());
```

Anschließend überprüfen wir, ob `eingabe` den Wert 0 hat, und brechen gegebenenfalls die Schleife ab. Das erledigt die `if`-Anweisung

```
if (eingabe == 0)
    break;
```

Wenn `eingabe` nicht den Wert 0 hat, wird die `break`-Anweisung übersprungen und die Summe errechnet. Dazu addieren wir den Wert von `eingabe` und den aktuellen Wert von `summe`. Die entsprechende Anweisung

```
summe = eingabe + summe;
```

mag vielleicht ein wenig seltsam wirken, erfüllt aber beim genauen Hinsehen ihren Zweck. Denn es wird ja erst addiert und dann zugewiesen. Der Wert von `summe` rechts im Ausdruck entspricht damit immer der Summe aus dem letzten Durchlauf der Schleife. Dieser Wert wird mit der eingegebenen Zahl addiert und dann wieder in der Variablen `summe` abgelegt.

Tipp:

Falls Ihnen das Verfahren merkwürdig vorkommt, probieren Sie es einfach mit einem Bleistift und einem Blatt Papier aus. Führen Sie mehrere Schleifendurchläufe aus, und rechnen Sie dabei immer erst den Ausdruck rechts vom Zuweisungsoperator aus. Das Ergebnis weisen Sie dann wieder der Variablen summe zu.

Mit der letzten Anweisung in der Schleife schließlich wird die Schleifenvariable um den Wert 1 erhöht.

Am Ende des Codes werden dann eine Meldung und der Wert von `summe` ausgegeben.

Allerdings hat die `break`-Anweisung in Schleifen auch ihre Tücken. Denn Sie programmieren ja eine Art „Querausstieg“ aus der Schleife, der von einem sehr eiligen Leser möglicherweise übersehen wird. Etwas übersichtlicher ist es daher in solchen Fällen, die Ausführung der Schleife von einer weiteren Variablen abhängig zu machen, die nur markiert, ob die Schleife weiter ausgeführt werden soll oder nicht.



Variablen, die einen Zustand markieren, werden auch **Flag-Variablen** genannt.
Wenn Sie nur zwischen zwei Zuständen unterscheiden müssen, können Sie für solche Variablen zum Beispiel den Typ `bool` benutzen.

Das Beispiel von oben könnte mit einer Flag-Variablen⁵ statt einer `break`-Anweisung folgendermaßen aussehen. Damit Sie die Änderungen schneller wiederfinden, haben wir sie fett markiert.

```
/* ##### Schleifenabbruch über eine Flag-Variable #####
using System;
namespace Cshp03d_03_09
{
    class Program
    {
        static void Main(string[] args)
        {
            int summe, schleifenVariable, eingabe;
            //die Flag-Variablen vom Typ bool
            bool flagVariable = true;

            //Initialisierung
            summe = 0;
            schleifenVariable = 1;

            while ((schleifenVariable <= 10) && (flagVariable == true))
            {

```

5. Wörtlich übersetzt bedeutet *flag* so viel wie „Flagge“. Bei der Programmierung ist damit aber ein eindeutiges Kennzeichen für die Unterscheidung von Zuständen gemeint.

```
//bitte in einer Zeile eingeben  
Console.Write("Geben Sie die {0}. Zahl ein: ",  
schleifenVariable);  
eingabe = Convert.ToInt32(Console.ReadLine());  
//wenn 0 eingegeben wurde, wird flagVariable auf false  
//gesetzt  
//andernfalls wird gerechnet  
if (eingabe == 0)  
    flagVariable = false;  
else  
{  
    summe = eingabe + summe;  
    schleifenVariable++;  
}  
}  
Console.WriteLine("Das Einlesen ist beendet.");  
Console.WriteLine("Die Summe der Zahlen ist {0}.", summe);  
}  
}
```

Code 3.9: Schleifenabbruch über eine Flag-Variable

Die Schleifensteuerung erfolgt jetzt durch eine logische UND-Verknüpfung der beiden Ausdrücke `schleifenVariable <= 10` und `flagVariable == true`. Die Schleife wird also nur noch dann ausgeführt, wenn beide Ausdrücke wahr sind. Damit wir auch für die Variable `flagVariable` einen eindeutigen Zustand haben, setzen wir sie bei der Vereinbarung auf den Wert `true`.

Hinweise:

Die Klammern um die beiden Ausdrücke dienen nur zur besseren Lesbarkeit. Sie können sie auch weglassen.

Den zweiten Ausdruck können Sie auch kompakter darstellen – nämlich einfach als `flagVariable`. Hier wird dann direkt der Wert von `flagVariable` verwendet. Die Variante ohne den ausdrücklichen Vergleich ist zwar kompakter, aber auch nicht mehr unbedingt beim flüchtigen Hinsehen zu verstehen.

Innerhalb der Schleife überprüfen wir dann wieder, ob `eingabe` den Wert 0 hat. Wenn das zutrifft, setzen wir `flagVariable` auf `false`. Damit wird die Schleife bei der nächsten Überprüfung des Ausdrucks im Schleifenkopf abgebrochen. Hat `eingabe` dagegen einen Wert ungleich 0, werden die beiden Anweisungen im `else`-Zweig ausgeführt. Dadurch wird sichergestellt, dass die Summe nur dann weitergeschrieben wird, wenn die Schleife nicht abgebrochen werden soll.

Probieren Sie den vorigen Code jetzt einmal aus. Sie werden sehen, er verhält sich genauso wie die Variante mit der `break`-Anweisung. Durch die logische Verknüpfung der beiden Ausdrücke in der Schleife wird jetzt aber sofort klar, dass die Ausführung der Schleife an **zwei Bedingungen** geknüpft ist. Das Programm ist zwar ein wenig länger, dafür aber übersichtlicher. Und das macht sich vor allem bei umfangreichen und komplexen Aufgaben sehr schnell bezahlt.

So viel zur Anweisung `break`. Kommen wir jetzt zu `continue`.

3.4.2 continue

Die Anweisung `continue`⁶ arbeitet ähnlich wie `break`, allerdings wird die Schleife nicht komplett abgebrochen, sondern es wird lediglich der aktuelle Schleifendurchlauf beendet.

Der Befehl `continue` sorgt also dafür, dass das Programm sofort zum Schleifenkopf bei einer `while`-Schleife beziehungsweise zum Schleifenfuß bei einer `do ... while`-Schleife springt. Dort wird dann die nächste Auswertung und danach eventuell der nächste Schleifendurchlauf ausgeführt.

Die allgemeine Syntax der `continue`-Anweisung sieht genauso aus wie bei `break`:

```
while (Ausdruck1)
{
    ...
    if (Ausdruck2)
        continue;
    ...
}
```

Sehen wir uns den Einsatz von `continue` an einem einfachen Beispiel an. Im folgenden Code wird das Quadrat einer eingegebenen Zahl in einer Schleife berechnet. Wenn die Zahl 0 eingegeben wird, springt das Programm direkt zum nächsten Schleifendurchlauf.

```
/* ##### continue in einer Schleife #####
using System;

namespace Cshp03d_03_10
{
    class Program
    {
        static void Main(string[] args)
        {
            int quadrat, schleifenVariable, eingabe;

            //Initialisierung
            quadrat = 0;
            schleifenVariable = 1;

            while (schleifenVariable <= 10)
            {
                //bitte in einer Zeile eingeben
                Console.Write("Geben Sie die {0}. Zahl ein: ",
                schleifenVariable);
                eingabe = Convert.ToInt32(Console.ReadLine());
            }
        }
    }
}
```

6. `Continue` bedeutet übersetzt so viel wie „setze fort“.

```
//die Schleifenvariable muss jetzt vor
//der Abfrage verändert werden
schleifenVariable++;
if (eingabe == 0)
    continue;
//die folgenden Anweisungen werden nur ausgeführt,
//wenn eingabe nicht 0 ist
quadrat = eingabe * eingabe;
//bitte in einer Zeile eingeben
Console.WriteLine("Das Quadrat der Zahl ist
{0}.",quadrat);
}
Console.WriteLine("Das Einlesen ist beendet.");
}
}
```

Code 3.10: continue in einer Schleife

Die `continue`-Anweisung finden Sie in dem Code in der `if`-Abfrage in der Schleife wieder.

```
if (eingabe == 0)  
    continue;
```

Wenn `eingabe` den Wert 0 hat, wird der aktuelle Durchlauf abgebrochen und der nächste Durchlauf gestartet. Das heißt, die Anweisungen nach der `if`-Abfrage werden dann nicht ausgeführt.

Bitte achten Sie bei `continue` sorgfältig auf das Verändern der Schleifenvariablen. In unserem Beispiel verändern wir die Schleifenvariable **vor** dem Aufruf der Anweisung `continue`. Damit ist sichergestellt, dass die Schleife insgesamt zehnmal durchlaufen wird. Ändern Sie die Schleifenvariable dagegen nach dem Aufruf von `continue`, können auch sehr viel mehr Durchläufe erfolgen. Denn dann wird die Schleifenvariable ja nur erhöht, wenn ein Wert ungleich 0 eingegeben wurde.

Da auch `continue`-Anweisungen zu recht unübersichtlichem Quelltext führen können, sollten Sie die Anweisung ebenfalls nur dann einsetzen, wenn Sie unbedingt müssen. In der Regel lassen sich nämlich mit einer `if`-Konstruktion die gleichen Ergebnisse erzielen. So könnten Sie im vorigen Code zum Beispiel die Berechnung nur dann ausführen lassen, wenn `eingabe` einen Wert ungleich 0 hat. Damit wäre das `continue` überflüssig. Der geänderte Code würde dann so aussehen:

```
/* ##### if-Abfrage statt continue ##### */
if-Abfrage statt continue
#####
using System;

namespace Cshp03d_03_11
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

int quadrat, schleifenVariable, eingabe;
//Initialisierung
quadrat = 0;
schleifenVariable = 1;
while (schleifenVariable <= 10)
{
    //bitte in einer Zeile eingeben
    Console.Write("Geben Sie die {0}. Zahl ein: ",
    schleifenVariable);
    eingabe = Convert.ToInt32(Console.ReadLine());

    //die Berechnungen werden nur ausgeführt, wenn ein
    //Wert ungleich 0 eingegeben wurde
    if (eingabe != 0)
    {
        quadrat = eingabe * eingabe;
        //bitte in einer Zeile eingeben
        Console.WriteLine("Das Quadrat der Zahl ist
        {0}.",quadrat);
    }
    schleifenVariable++;
}
Console.WriteLine("Das Einlesen ist beendet.");
}
}
}

```

Code 3.11: if-Konstruktion statt continue

Die Anweisungen `break` und `continue` können Sie grundsätzlich auch in `for`-Schleifen benutzen. Allerdings lässt sich dann der Schleifenablauf ebenfalls nicht mehr nur über die drei Anweisungen zu Beginn der Schleife ablesen. Deshalb sollten Sie die beiden Anweisungen in `for`-Schleifen nur sehr vorsichtig verwenden und ausführlich kommentieren.

Zum Abschluss dieses Kapitels wollen wir uns noch – wie versprochen – ansehen, wie Sie die Eingabe von ungültigen Daten abfangen können.

3.5 Exkurs: Abfangen von ungültigen Eingaben

Bisher mussten wir uns bei allen Programmen damit zufriedengeben, dass die Eingaben recht „wackelig“ sind. Wenn ein Anwender zum Beispiel statt – wie gefordert – einer Zahl ein Zeichen eingibt, stürzen die Programme ab, da die Konvertierung nicht möglich ist. Diese Abstürze lassen sich durch eine Schleife und die **Ausnahmebehandlung** – das Exception Handling – verhindern.

Das folgende Fragment liest zum Beispiel so lange Daten über die Tastatur ein, bis die Umwandlung über die Methode `Convert.ToInt32()` gelingt.

```

//gelungen ist vom Typ bool und muss mit false
//initialisiert werden
while (gelungen == false)
{

```

```
try
{
    eingabe = Convert.ToInt32(Console.ReadLine());
    gelungen = true;
}
catch (FormatException)
{
    Console.Write("Ihre Eingabe war nicht gültig. Bitte
    wiederholen... ");
}
```

Code 3.12: Das Abfangen von Eingabefehlern

Entscheidend sind hier die Blöcke für `try` und `catch`. Im `try`-Block wird zunächst nur **versucht**, die Eingabe zu konvertieren. Wenn diese Konvertierung scheitert, werden die Verarbeitungen im `try`-Block abgebrochen und die Anweisungen aus dem `catch`-Block ausgeführt. Das heißt für das Beispiel: Die Schleifenvariable `gelungen` wird nur dann auf `true` gesetzt, wenn auch die Konvertierung im `try`-Block erfolgreich war. Scheitert die Konvertierung dagegen, behält die Schleifenvariable ihren Initialwert `false` und die Schleife wird wiederholt.

Damit wollen wir den Ausflug in die Ausnahmebehandlung an dieser Stelle auch schon wieder beenden. Mehr zu dem Thema erfahren Sie später.

Probieren Sie die Anweisungen aus dem vorigen Code aber jetzt schon einmal selbst aus. Ein vollständiges Beispiel finden Sie auch im Projekt **Cshp03d_03_12**.

Im nächsten Kapitel werden wir uns mit den Methoden beschäftigen.

Zusammenfassung

Mit Schleifen können Sie die Wiederholung von Anweisungen steuern.

C# kennt drei verschiedene Schleifentypen:

- die kopfgesteuerte Schleife mit `while`,
- die fußgesteuerte Schleife mit `do ... while` und
- die Zählschleife mit `for`.

Mit der Anweisung `break` können Sie die Ausführung einer Schleife komplett abbrechen. Die Anweisung `continue` bricht den aktuellen Durchlauf einer Schleife ab.

Aufgaben zur Selbstüberprüfung

- 3.1 Welcher wesentliche Unterschied besteht zwischen einer `while`- und einer `do ... while`-Schleife?

- 3.2 Die folgende Schleife soll von 10 bis 20 zählen. Welche Wirkung hat die Schleife tatsächlich? Warum?

```
int schleifenVariable = 10;
while (schleifenVariable <= 20)
{
    Console.WriteLine("{0}", schleifenVariable);
}
```

- 3.3 Ersetzen Sie im folgenden Quelltext die `break`-Anweisung in der `while`-Schleife durch eine andere Konstruktion. Die wesentliche Funktionalität soll dabei erhalten bleiben. Ein kleiner Tipp: Benutzen Sie eine Flag-Variable.

```
/* ##### break in einer Schleife #####
using System;

namespace LoesIII3
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, k;
            i = 0;
            k = 0;
```

```
while (i <= 5)
{
    Console.Write("Geben Sie eine 1 zum Abbruch ein.");
    k = Convert.ToInt32(Console.ReadLine());
    if (k == 1)
        break;
    i++;
}

Console.WriteLine("Schleife beendet.");
}
```

4 Methoden

In diesem Kapitel erfahren Sie, wie Sie Ihre Programme mit **Methoden** übersichtlicher und kompakter gestalten können. Sie lernen, wie Methoden in C# aufgebaut sind und wie Sie Werte aus Methoden zurückliefern beziehungsweise Werte an Methoden übergeben.

4.1 Methoden in C#

Jetzt fragen Sie sich vielleicht: „Warum brauche ich überhaupt Methoden in einem Programm? Bisher lief doch alles wunderbar. Warum also die Sache unnötig komplizierter machen?“

Die Antworten auf diese Fragen sind einfach, wenn Sie sich die folgende Situation vorstellen:

Sie sollen ein Programm schreiben, in dem Sie an fünf verschiedenen Stellen zehn verschiedene Zahlen darauf überprüfen müssen, ob sie mit den Werten 5, 7, 12, 16 oder 21 übereinstimmen.

Kein Problem, sagen Sie? Sie erstellen einfach entsprechende `switch ... case`-Verzweigungen, die Sie dann in der benötigten Anzahl kopieren. Damit blähen Sie aber Ihren Quelltext unnötig auf, da Sie ja die gleiche Funktionalität mindestens fünfmal neu programmieren. Sehr viel effektiver und einfacher wäre es doch, den Quelltext für die Anweisungen nur einmal zu programmieren, aber beliebig oft nutzen zu können. Und genau das erreichen Sie mit Methoden.



Eine Methode wird einmal erstellt und kann dann im Quelltext beliebig oft aufgerufen werden. Methoden gehören immer zu einer Klasse.

In unserem Beispiel würden Sie also eine Methode schreiben, der Sie eine Zahl übergeben können. Die Methode gibt dann zurück, ob die Zahl mit den gewünschten Werten übereinstimmt oder nicht.

Damit sparen Sie sich nicht nur jede Menge Schreibarbeit, sondern halten auch den Quelltext sehr viel übersichtlicher. Denn der Code für die Überprüfung steht so nur einmal im Programm – und zwar in der Methode. An den Stellen, an denen Sie den Test durchführen wollen, rufen Sie diese Methode lediglich auf und übergeben dabei die Zahl, die getestet werden soll.

Der syntaktische Aufbau einer Methode in C# sieht so aus:

```
static Rückgabewert Name (Parametertypen)
{
    Anweisung1;
    Anweisung2;
    Anweisung...;
}
```

Die erste Zeile

```
static Rückgabetyp Name (Parametertypen)
```

bildet den **Kopf**. Er besteht aus dem Schlüsselwort `static`, dem **Rückgabetyp**, dem **Namen** der Methode und der Angabe der **Parameter**, die an die Methode übergeben werden. Die Parameter werden durch runde Klammern umfasst.

Hinweise:

Was es mit dem Rückgabetyp und den Parametern genau auf sich hat, erfahren Sie im weiteren Verlauf dieses Kapitels.

Bei Methoden, die mit dem Schlüsselwort `static`⁷ vereinbart werden, handelt es sich um eine spezielle Art von Methoden – die **Klassenmethoden**. Es gibt auch Methoden in Klassen, die ohne das Schlüsselwort `static` vereinbart werden. Damit werden wir uns im weiteren Verlauf des Lehrgangs bei der Objektorientierung beschäftigen. Hier gehen wir immer davon aus, dass die Methoden zu der Klasse gehören, die unser eigentliches Programm darstellt.

Für die Namensvergabe an Methoden gelten dieselben Regeln wie für die Namensvergabe an andere Bezeichner. Da Sie diese Regeln bereits kennen, wollen wir hier nur die wichtigsten Sachen noch einmal im Schnelldurchgang vorstellen:

- Methodennamen müssen mit einem Buchstaben oder einem `_` beginnen und dürfen ansonsten nur Buchstaben, Ziffern oder den Unterstrich `_` enthalten. `Test1` wäre zum Beispiel ein möglicher Name für eine Methode, während `1test` nicht zulässig ist.
- Groß- und Kleinschreibung werden unterschieden. `Test`, `TEST`, `test` und `TeST` bezeichnen also vier unterschiedliche Methoden.
- Methodennamen dürfen nicht identisch mit Schlüsselwörtern sein. Schlüsselwörter als Teile des Namens sind dagegen erlaubt. `usingTest` wäre zum Beispiel ein möglicher Methodenname, während `using` oder `int` als Namen nicht zulässig sind.

Auf den Kopf folgt der **Methodenkörper**. Er besteht aus einem Anweisungsblock – also aus einer Folge von Anweisungen in geschweiften Klammern. Im Methodenkörper steht der Programmcode, der beim Aufrufen der Methode ausgeführt werden soll. Bei sehr kurzen Methoden kann der Methodenkörper auch lediglich aus einer einzigen Anweisung bestehen. Syntaktisch korrekt, aber unsinnig wäre auch eine Methode mit einem leeren Körper – also ganz ohne Anweisung und damit auch ohne jede Wirkung.

Eine sehr einfache Methode in C# könnte damit so aussehen:

```
static void Methode()  
{  
}
```

Der Rückgabetyp dieser Methode wird durch das Schlüsselwort `void` gekennzeichnet. Es bedeutet so viel wie „leer“. Damit ist aber nicht gemeint, dass die Methode keinen Inhalt hat, sondern dass sie keinen Wert zurückliefert.

7. `Static` bedeutet wörtlich übersetzt statisch – also unveränderlich.



Methoden, die keinen Wert zurückgeben, haben den Rückgabetyp void.

Werte werden von der Methode `Methode()` ebenfalls nicht verarbeitet. Deshalb ist die Liste in den Klammern hinter dem Namen der Methode leer.

Eine ganz bestimmte C#-Methode haben Sie übrigens schon die ganze Zeit in Ihren Programmen eingesetzt – nämlich die Methode `Main()`. Diese Methode bildet den Startpunkt des Programms. Aus dem Methodenkörper von `Main()` können Sie dann auch andere, selbst geschriebene Methoden aufrufen. Wie das funktioniert, zeigen wir Ihnen gleich.



Der Start einer C#-Anwendung erfolgt mit dem automatischen Aufruf der `Main()`-Methode. Jede C#-Anwendung benötigt daher eine `Main()`-Methode.

Sehen wir uns nun an einem Beispiel an, wie Sie Aufgaben in mehrere Methoden zerlegen und die Methoden dann aufrufen. Der folgende Code gibt lediglich zwei Texte auf dem Bildschirm aus. Wir werden ihn gleich so umbauen, dass die Ausgabe über zwei Methoden erfolgt.

```
/* #####
Einfache Textausgabe
##### */

using System;

namespace Cshp03d_04_01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("C# macht Spaß.");
            Console.WriteLine("Aber nicht immer.");
        }
    }
}
```

Code 4.1: Einfache Textausgabe

Die beiden Ausgaben lassen wir nun jeweils durch eine Methode ausführen. Diese Methoden nennen wir `Ausgabe1()` und `Ausgabe2()`. Sie benötigen keinen Rückgabewert und auch keine Parameter. Also setzen wir den Rückgabewert auf `void` und lassen die Liste der Parameter hinter dem Namen der Methode leer.

Hinweis:

Wir benutzen die beiden Methoden lediglich als Beispiel, das Sie sehr einfach nachvollziehen können. In der Praxis ergibt es wenig Sinn, feste Texte über eine Methode ausgeben zu lassen, da keine Änderungen möglich sind und die Methode daher häufig nur an einer bestimmten Stelle eingesetzt werden kann.

Der Code mit den beiden Methoden sieht dann so aus:

```
/* ##### Einfache Textausgabe mit Methoden ##### */
using System;
namespace Cshp03d_04_02
{
    class Program
    {
        //die erste Methode Ausgabe1()
        static void Ausgabe1()
        {
            Console.WriteLine("C# macht Spaß.");
        }

        //die zweite Methode Ausgabe2()
        static void Ausgabe2()
        {
            Console.WriteLine("Aber nicht immer.");
        }

        static void Main(string[] args)
        {
            //der Aufruf der ersten Methode
            Ausgabe1();

            //der Aufruf der zweiten Methode
            Ausgabe2();
            Console.WriteLine("Das war es.");
        }
    }
}
```

Code 4.2: Einfache Textausgabe mit Methoden

Zu Beginn des Codes werden die beiden Methoden `Ausgabe1()` und `Ausgabe2()` vereinbart. Sie geben jeweils lediglich eine Zeile Text aus.

Bitte beachten Sie:

Eine Klassenmethode kann nicht innerhalb einer anderen Methode vereinbart werden – auch nicht innerhalb von `Main()`. Die Vereinbarung einer Klassenmethode muss daher außerhalb jeder anderen Methode erfolgen, aber innerhalb der Klasse.

Die Reihenfolge der Vereinbarung im Quelltext ist beliebig. Sie können also eine Methode auch erst nach der ersten Aufrufstelle vereinbaren. Wichtig ist lediglich, dass sie in der Klasse steht und so vom Compiler „gefunden“ werden kann.



Der Aufruf der beiden Methoden `Ausgabe1()` und `Ausgabe2()` erfolgt dann in der Methode `Main()` durch den Namen der Methode, gefolgt von der Liste der **Argumente**.

Hinweis:

Argumente sind die konkreten Werte, die an eine Funktion beim Aufruf übergeben werden. Mehr zum Unterschied zwischen Parameter und Argument erfahren Sie, wenn wir uns mit der Übergabe von Werten an Funktionen beschäftigen.

In unserem Beispiel verarbeiten allerdings beide Methoden keine Argumente. Deshalb sind die Listen immer leer – sowohl bei der Vereinbarung als auch beim Aufruf.

**Bitte beachten Sie:**

Auch wenn keine Argumente beziehungsweise Parameter verarbeitet werden, ist die Angabe der Klammern immer zwingend erforderlich – sowohl bei der Vereinbarung als auch beim Aufruf einer Methode. Sie dürfen die Klammern nicht weglassen!

Durch den Aufruf werden die Anweisungen in der entsprechenden Methode abgearbeitet. Nach dem Durchlaufen der Methode kehrt das Programm automatisch zurück an die Stelle, von der die Methode aufgerufen wurde, und setzt die Verarbeitung mit der folgenden Anweisung fort.

In unserem Beispiel wird also zuerst die Methode `Ausgabe1()` ausgeführt und danach die Methode `Ausgabe2()`. Nach dem Ausführen der Methode `Ausgabe2()` wird dann die letzte Zeile im Code abgearbeitet.

Grafisch lässt sich der Ablauf so darstellen:

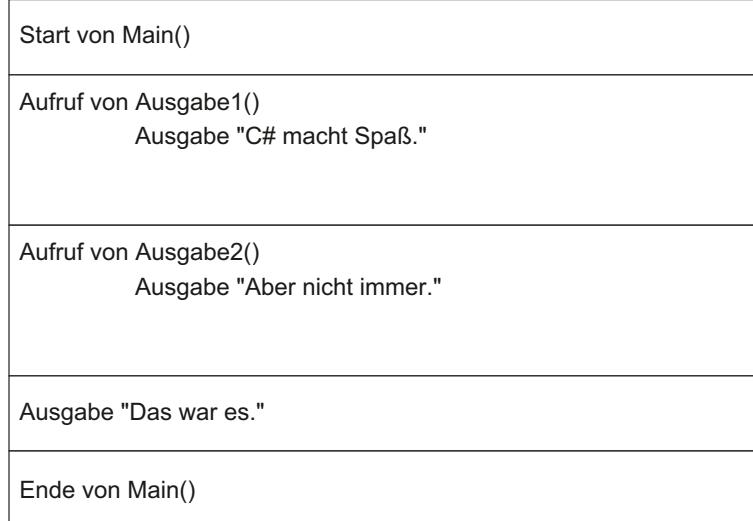


Abb. 4.1: Der Programmablauf für Code 4.2

So viel zum Aufbau und zum Aufruf von Methoden. Schauen wir uns jetzt an, wie Sie Werte aus einer Methode zurückgeben lassen.

4.2 Methoden mit Rückgabewert

Durch den Rückgabewert können Sie das Ergebnis einer Methode weiterverarbeiten – zum Beispiel in der Methode `Main()`.

Die Rückgabe eines Wertes aus einer Methode erfolgt mit dem Schlüsselwort `return`. Hinter `return` wird der Wert angegeben, den die Methode liefern soll.



Zusätzlich müssen Sie den Typ der Rückgabe im Methodenkopf angeben. Dabei können Sie alle Typen verwenden, die Sie bereits von den Variablen kennen.

Die Angabe des Rückgabetyps ist zwingend erforderlich – auch dann, wenn die Methode keinen Wert liefert. In diesem Fall müssen Sie `void` verwenden.



Die folgende Methode `Rueckgabe()` liefert zum Beispiel den Wert 100 als `int`-Typ zurück.

```
static int Rueckgabe()
{
    return 100;
}
```

Bitte beachten Sie, dass eine Methode nach einem `return` sofort beendet wird. Alle weiteren Anweisungen, die eventuell noch folgen, werden nie ausgeführt. Im folgenden Beispiel würde also die Ausgabe nie auf dem Bildschirm erscheinen.

```
static int Rueckgabe()
{
    return 100;
    Console.WriteLine("Der Wert 100 wurde zurückgegeben.");
}
```

Daher meldet sich der Compiler bei solchen Konstruktionen auch mit einer Warnung, dass unerreichbarer Code entdeckt wurde.

Schauen wir uns die Methoden mit Rückgabewert jetzt an einem C#-Programm an. Das folgende Beispiel arbeitet mit zwei Methoden `Rueckgabe100()` und `Rueckgabe10()`, die den Wert 100 beziehungsweise 10 als `int`-Typ zurückgeben.

```
/* #####
Zwei Methoden mit Rückgabe
#####
*/
using System;

namespace Cshp03d_04_03
{
```

```

class Program
{
    //die Methode Rueckgabe100() vom Typ int
    static int Rueckgabe100()
    {
        return 100;
    }

    //die Methode Rueckgabe10() vom Typ int
    static int Rueckgabe10()
    {
        return 10;
    }

    static void Main(string[] args)
    {
        int ergebnis;

        //Aufruf der beiden Methoden in einer Ausgabe
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Die Methode Rueckgabe100() liefert den Wert {0}.".Rueckgabe100());
        Console.WriteLine("Die Methode Rueckgabe10() liefert den Wert {0}.".Rueckgabe10());

        //Berechnungen gehen auch
        ergebnis = Rueckgabe100() + Rueckgabe10();
        Console.WriteLine("Das Ergebnis ist {0}.".ergebnis);

        //in Bedingungen kann eine Methode auch eingesetzt werden
        Console.Write("Geben Sie eine Zahl ein: ");
        ergebnis = Convert.ToInt32(Console.ReadLine());
        if (ergebnis < Rueckgabe100())
            Console.WriteLine("Ihre Eingabe war kleiner als 100.");
    }
}

```

Code 4.3: Zwei Methoden mit Rückgabewert

In der Methode `Main()` lassen wir zunächst einfach den Rückgabewert der beiden Methoden ausgeben. Dazu setzen Sie den Namen der Methode gefolgt von den leeren Klammern an die Stelle, an der sonst immer der Variablenname stand.

In der Anweisung

```
ergebnis = Rueckgabe100() + Rueckgabe10();
```

addieren wir dann die beiden Rückgabewerte und weisen das Ergebnis der Variablen `ergebnis` zu. Gerechnet wird hier also $100 + 10$.

In der `if`-Abfrage

```
if (ergebnis < Rueckgabe100())
```

schließlich verwenden wir den Rückgabewert der Methode `Rueckgabe100()` für einen Vergleich. Auch hier geben Sie einfach den Namen der Methode gefolgt von den Klammern an.

Sie sehen, Sie können die beiden Methoden an nahezu allen Stellen einsetzen, an denen Sie auch mit Zahlen oder Variablen vom Typ `int` arbeiten können – also bei Zuweisungen, bei Vergleichen, in Ausdrücken mit logischen Operatoren, in Verzweigungen und so weiter.

Noch einmal, weil es zu Beginn gerne vergessen wird:

Sie müssen in allen Fällen die Klammern hinter dem Namen der Methode angeben.



Immerhin meldet der Compiler in vielen Fällen einen Fehler, wenn Sie zum Beispiel eine Methode in einem Ausdruck verwenden wollen und dabei die Klammern vergessen.

Wenn Sie mit Methoden arbeiten, die Werte zurückliefern, müssen Sie darauf achten, dass die Methode auch wirklich **in jedem Fall** mit einem `return` beendet wird. Das heißt, die Anweisung `return` muss auch tatsächlich ausgeführt werden. Es reicht nicht, dass die Anweisung einfach im Quelltext steht.

Eine Methode, die einen Wert liefern soll, aber nicht mit `return` beendet wird, wird vom Compiler nicht akzeptiert.



Schauen wir uns dazu ein Beispiel an. Der folgende Code soll in der Methode `Eingabe()` einen Wert über die Tastatur einlesen und ihn an `Main()` zurückliefern. Die Rückgabe erfolgt allerdings nur dann, wenn der eingelesene Wert kleiner oder gleich 20 ist.

```
/* ##### Undefinierte Rückgabe aus einer Methode
Das Beispiel lässt sich nicht übersetzen
##### */  
  
using System;  
  
namespace Cshp03d_04_04  
{  
    class Program  
    {  
        static int Eingabe()  
        {  
            int einVariable;  
  
            Console.Write("Geben Sie eine Zahl ein: ");  
            einVariable = Convert.ToInt32(Console.ReadLine());  
        }  
    }  
}
```

```

        if (einVariable <= 20)
            return einVariable;
    }

    static void Main(string[] args)
    {
        //bitte in einer Zeile eingeben
        Console.WriteLine("Die Methode Eingabe() liefert den Wert
        {0}.",Eingabe());
    }
}

```

Code 4.4: Undefinierte Rückgabe aus einer Methode (der Code lässt sich nicht übersetzen)

Hier beschwert sich der Compiler zu Recht, dass in der Methode `Eingabe()` nicht alle Codepfade einen Wert zurückliefern – obwohl eine `return`-Anweisung vorhanden ist. Die wird allerdings nicht in jedem Fall auch tatsächlich ausgeführt.

Sie müssen außerdem darauf achten, dass der Rückgabetyp der Methode zu dem Typ des Wertes passt, der mit `return` verarbeitet wird. Schauen wir uns auch dazu ein Beispiel an. Wir überarbeiten die Methode `Eingabe()` aus dem vorigen Code einmal so, dass sie einen `char`-Typ zurückliefert. Den Typ der Variablen `einVariable` in der Methode lassen wir unverändert.

```

/* ##### Unterschiedliche Typen in einer Methode
Das Beispiel lässt sich ebenfalls nicht übersetzen
##### */

using System;

namespace Cshp03d_04_05
{
    class Program
    {
        static char Eingabe()
        {
            int einVariable;

            Console.Write("Geben Sie eine Zahl ein: ");
            einVariable = Convert.ToInt32(Console.ReadLine());
            return einVariable;
        }

        static void Main(string[] args)
        {
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Methode Eingabe() liefert den Wert
            {0}.", Eingabe());
        }
    }
}

```

Code 4.5: Unterschiedliche Typen (der Code lässt sich ebenfalls nicht übersetzen)

Hier beschwert sich der Compiler jetzt bei der `return`-Anweisung in der Methode, dass er den Typ `int` nicht in einen `char` konvertieren kann.

Hinweis:

Ob Sie den Wert hinter `return` in Klammern setzen oder nicht, ist bei der Rückgabe eines einzelnen Wertes übrigens beliebig. Häufig werden die Klammern zur besseren Lesbarkeit gesetzt.

Mit Standardtechniken kann eine Methode in C# immer nur einen einzigen Wert zurückgeben. Sie können also zum Beispiel in einer Methode nicht zwei Werte einlesen und dann ohne Weiteres beide Werte zurückgeben. Der folgende Code wird daher ebenfalls nicht übersetzt.

```
/* ##### Das klappt nicht! #####
using System;
namespace Cshp03d_04_06
{
    class Program
    {
        static int Eingabe()
        {
            int einVariable1, einVariable2;

            Console.Write("Geben Sie Zahl 1 ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie Zahl 2 ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());
            return (einVariable1, einVariable2);
        }

        static void Main(string[] args)
        {
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Methode Eingabe() liefert den Wert
{0}.",Eingabe());
        }
    }
}
```

Code 4.6: Das klappt nicht!

Hier beschwert sich der Compiler ebenfalls wieder über die Rückgabe aus der Methode.

Damit die Rückgabe mehrerer Werte möglich ist, müssen Sie ein Tupel verwenden. Der geänderte Code sieht dann so aus:

```
/*
#####
Rückgabe mehrerer Werte über ein Tupel
#####
*/

using System;

namespace Cshp03d_04_07
{
    class Program
    {
        //die Methode gibt ein Tupel zurück
        static (int, int) Eingabe()
        {
            int einVariable1, einVariable2;

            Console.Write("Geben Sie Zahl 1 ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie Zahl 2 ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());

            return (einVariable1, einVariable2);
        }

        static void Main(string[] args)
        {
            //die Werte aus der Methode zuweisen
            //die Zuweisung erfolgt auf ein Tupel
            (int wert1, int wert2) = Eingabe();
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Methode Eingabe() liefert
            die Werte {0} und {1}.", wert1, wert2);
        }
    }
}
```

Code 4.7: Rückgabe mehrerer Werte über ein Tupel

Im Kopf der Methode wird als Rückgabewert jetzt ein Tupel vereinbart.



Ein Tupel ist eine geordnete Menge von Werten.

Dazu geben Sie einfach die Typen für die Werte an, die Sie zurückliefern wollen, und trennen Sie durch ein Komma. Die Liste setzen Sie dabei in runde Klammern.

Der Kopf

```
static (int, int) Eingabe()
```

legt also fest, dass unsere Methode `Eingabe()` zwei `int`-Werte zurückliefert.

Bei der Rückgabe geben Sie dann die beiden Werte hintereinander in runden Klammern an und trennen Sie ebenfalls durch Kommas.

Bei der Rückgabe eines Tupels sind die runden Klammern bei `return` zwingend erforderlich.



Interessant ist dann noch einmal die Zuweisung der Ergebnisse in der Methode `Main()`. Sie erfolgt durch die Anweisung

```
(int wert1, int wert2) = Eingabe();
```

In den runden Klammern stehen dabei die Typen und Bezeichner für die Variablen, die die Werte speichern sollen. Die Angaben trennen Sie dabei wieder durch ein Komma. Dann erfolgt die Zuweisung durch den Operator `=` und den Aufruf der Methode.

Bei der Ausgabe gibt es keine Besonderheiten. Hier greifen wir auf die beiden Variablen zu, denen wir die Rückgabe der Methode zugewiesen haben und geben die Werte aus.

4.3 Methoden mit Argumenten

Sonderlich nützlich waren unsere Methoden bisher noch nicht. Denn alles, was wir in den vorigen Beispielen mit den Methoden umgesetzt haben, lässt sich genauso gut auch ohne Methode erledigen – sogar mit weniger Schreibaufwand.

Richtig spannend werden Methoden nämlich erst, wenn Sie sie mit Werten füttern. Dazu übergeben Sie beim Aufruf der Methode **Argumente**.

Die Begriffe **Argument** und Parameter werden oft nicht eindeutig unterschieden. Streng genommen sind **Parameter** die Angaben in den runden Klammern im Methodenkopf. **Argumente** dagegen sind die Werte, die beim Aufruf der Methode in den runden Klammern angegeben werden. Häufig wird der Begriff Parameter aber für beides verwendet – also sowohl für die eigentlichen Parameter als auch für die Argumente.



Damit Sie Argumente an eine Methode übergeben können, sind zwei Erweiterungen erforderlich:

- 1) Sie müssen im **Methodenkopf** in den Klammern die Typen und die Namen der Parameter angeben.
- 2) Sie müssen die Werte – die Argumente – beim **Aufruf** der Methode in den Klammern angeben.

Schauen wir uns zuerst die Änderungen im Methodenkopf an einem Beispiel genauer an:

```
static int Quadrat(int zahl)
```

Vereinbart wird hier eine Methode `Quadrat()`, die einen Wert vom Typ `int` zurückliefert. Die Methode verarbeitet einen Parameter vom Typ `int` mit dem Namen `zahl`. Durch die Angabe in der Parameterliste kann die Variable `zahl` innerhalb der Methode

genauso benutzt werden wie jede andere innerhalb der Methode vereinbarte Variable auch. Der einzige Unterschied besteht darin, dass die Variable bereits mit einem Wert initialisiert ist, der beim Aufruf der Methode übergeben wird.

Außerhalb der Methode ist die Variable allerdings nicht bekannt. Das heißt, sie gilt ausschließlich für die Methode.



Eine Variable, die Sie als Parameter einer Methode verwenden, wird automatisch vereinbart. Sie müssen keine eigene Anweisung für die Vereinbarung benutzen.

Der Aufruf der Methode `Quadrat()` kann dann zum Beispiel so erfolgen:

```
Console.WriteLine("Das Quadrat der Zahl {0} ist {1}", 10,
    Quadrat(10));
```

Hier wird die Zahl 10 als Argument an die Methode `Quadrat()` übergeben. Möglich wäre aber auch ein Aufruf wie

```
Console.WriteLine("Das Quadrat der Zahl {0} ist {1}", a,
    Quadrat(a));
```

Hier wird der Wert der Variablen `a` als Argument an die Methode übergeben.

Grundsätzlich können Sie an eine Methode alles als Argument übergeben, was Sie auch dem Parameter im Methodenkopf zuweisen können. Für unser Beispiel wären also auch komplexe Ausdrücke wie `a * b / c` möglich. An die Methode wird dann der Wert des Ausdrucks als Argument übergeben.



Lassen Sie sich durch die verschiedenen Bezeichner des Parameters im Methodenkopf und des Arguments beim Aufruf nicht verwirren.

Der Bezeichner des Parameters im Methodenkopf ist eine Art Platzhalter für das Argument. Sie müssen also beim Aufruf nicht den Bezeichner des Parameters verwenden, sondern können für den Platzhalter beliebige Werte und Ausdrücke einsetzen. Sie müssen lediglich darauf achten, dass der Typ des Arguments zum Typ des Parameters passt.

Sehen wir uns jetzt die Arbeit mit Argumenten an einem praktischen Beispiel an. Im folgenden Code wird das Quadrat einer Zahl über eine Methode berechnet und das Ergebnis dann in der Methode `Main()` ausgegeben.

```
/* #####
Eine Methode mit Argument
#####
*/
using System;
namespace Cshp03d_04_08
{
    class Program
    {
```

```
static int Quadrat(int zahl)
{
    int ergebnis;

    //zahl ist aus der Parameterliste bekannt
    ergebnis = zahl * zahl;
    return ergebnis;
}

static void Main(string[] args)
{
    int einVariable;

    Console.Write("Geben Sie eine Zahl ein: ");
    einVariable = Convert.ToInt32(Console.ReadLine());

    //einVariable wird als Argument an die Methode übergeben
    //bitte in einer Zeile eingeben
    Console.WriteLine("Das Quadrat der Zahl {0} ist {1}.",
        einVariable, Quadrat(einVariable));
}
```

Code 4.8: Eine Methode mit Argument

Das Programm liest zunächst einen Wert in die Variable `einVariable` ein. Dieser Wert wird dann beim Aufruf an die Methode in Klammern als Argument übergeben.

In der Methode selbst findet sich der Wert von `einVariable` in der Variablen `zahl` wieder. Diese Variable wurde im Methodenkopf in der Parameterliste vereinbart.

Nach der Berechnung liefert die Methode das Ergebnis über die `return`-Anweisung zurück. Abschließend erfolgt die Ausgabe des zurückgegebenen Wertes über `Console.WriteLine()`.

Bitte beachten Sie:

Bei der Übergabe der Argumente müssen Sie darauf achten, dass der Typ des Arguments zum Typ des Parameters passt. Wenn Sie in dem Code oben zum Beispiel den Typ der Variablen `einVariable` in `float` ändern, beschwert sich der Compiler, dass er das Argument für die Methode `Quadrat()` nicht konvertieren kann.



Schauen wir uns nun noch an, wie Sie mehrere Argumente an eine Methode übergeben.

Dazu geben Sie einfach die Argumente hintereinander in der gewünschten Reihenfolge in den Klammern beim Aufruf an und trennen sie durch Kommas. Außerdem müssen Sie auch noch die Parameterliste im Methodenkopf so erweitern, dass mehrere Werte verarbeitet werden können.

Im folgenden Beispiel sehen Sie sowohl den Kopf als auch den Aufruf einer Methode mit zwei Parametern beziehungsweise Argumenten:

```
static int Summe(int x, int y) //Kopf
Summe(a,b)                  //Aufruf der Methode
```


Bitte beachten Sie:

Sie müssen in der Parameterliste für jeden Parameter den Typ einzeln angeben. Die verkürzte Form

```
static int Summe(int x, y)
```

wird vom Compiler nicht akzeptiert.

Die Anzahl der Argumente beim Aufruf und der Parameter bei der Vereinbarung muss übereinstimmen. Wenn Sie zum Beispiel an die Methode `Summe()` nur ein Argument übergeben, meldet der Compiler, dass die Methode für diese Anzahl Argumente nicht verfügbar ist.

Eine ähnliche Meldung erscheint auch dann, wenn Sie mehr Argumente angeben, als die Methode verarbeiten kann.

Im praktischen Einsatz finden Sie die Methode `Summe()` im folgenden Code. Das Programm addiert zwei Zahlen, die Sie eingeben, und gibt das Ergebnis auf dem Bildschirm aus.

```
/* ##### Eine Methode mit mehreren Argumenten #####
using System;
namespace Cshp03d_04_09
{
    class Program
    {
        static int Summe(int x, int y)
        {
            return (x + y);
        }

        static void Main(string[] args)
        {
            int einVariable1, einVariable2;

            Console.Write("Geben Sie die erste Zahl ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie die zweite Zahl ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Summe der beiden Zahlen ist
{0}", Summe(einVariable1, einVariable2));
        }
    }
}
```

Code 4.9: Eine Methode mit zwei Argumenten

Die Schreibweise

```
return (x + y);
```

für die Rückgabe des Wertes aus der Methode ist die verkürzte Version von

```
int ergebnis;
ergebnis = x + y;
return ergebnis;
```

Da `x + y` ein berechenbarer Ausdruck ist und einen `int`-Wert liefert, können wir ihn direkt hinter `return` angeben und uns eine eigene Variable für die Rückgabe sparen.

Experimentieren Sie mit dem vorigen Code ein wenig. Versuchen Sie zum Beispiel einmal, die Summe aus vier oder fünf Werten in der Methode zu berechnen.

4.4 Tipps zum Arbeiten mit Methoden

Der Editor von Visual Studio stellt Ihnen einige Funktionen zur Verfügung, die die Arbeit mit Methoden erleichtern können – vor allem dann, wenn Sie später einmal ein Programm mit sehr vielen Methoden erstellen.

So finden Sie zum Beispiel links neben jedem Methodenkopf ein Symbol mit einem Minus-Zeichen und einer Linie. Durch einen Mausklick auf dieses Symbol können Sie den gesamten Körper der jeweiligen Methode ausblenden und so die Anzeige des Quelltextes sehr kompakt halten.

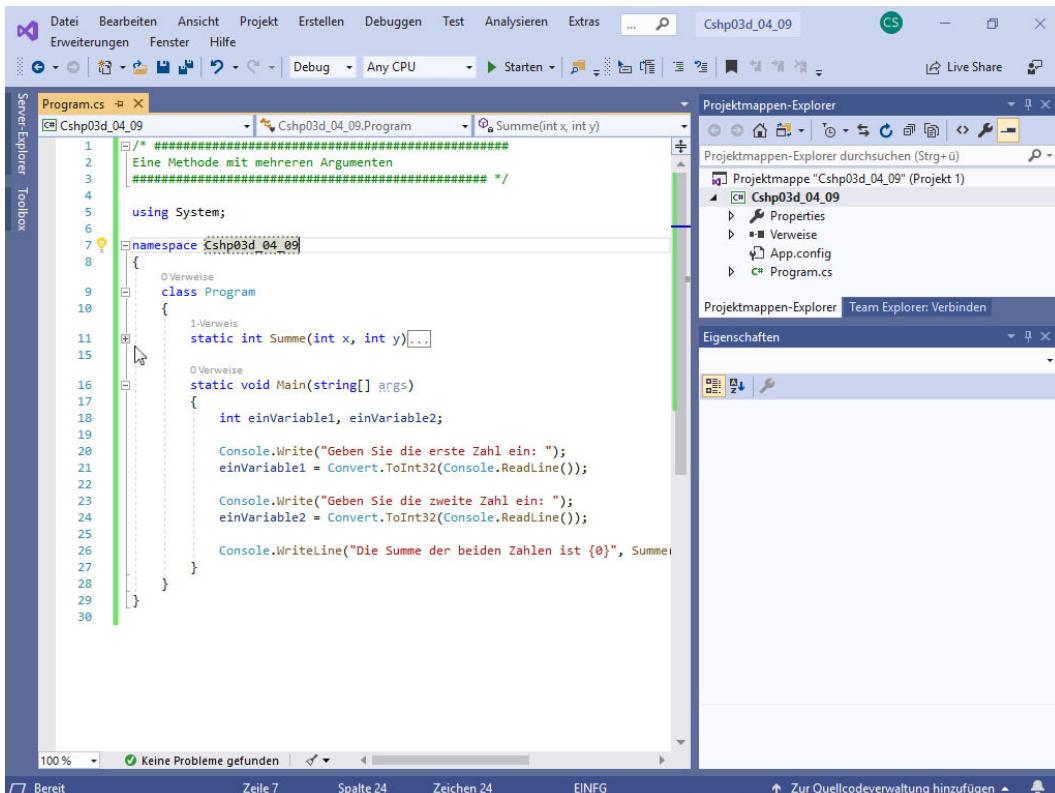


Abb. 4.2: Ein ausgeblendeter Methodenkörper (links in der Mitte der Abbildung am Mauszeiger)

Wenn Sie den Quelltext der Methode wieder anzeigen möchten, klicken Sie noch einmal auf das Symbol. Alternativ können Sie den Mauszeiger auch auf den Kasten hinter dem Methodenkopf stellen. Dann erscheint der Quelltext der Methode als Quickinfo.

Über das Kombinationsfeld rechts oben im Editor können Sie sehr schnell zwischen verschiedenen Methoden im Quelltext hin und her springen. Dazu öffnen Sie das Kombinationsfeld und klicken dann in der Liste auf den Eintrag der gewünschten Methode. Außerdem können Sie in der Liste auch ablesen, welche Argumente eine Methode erwartet.

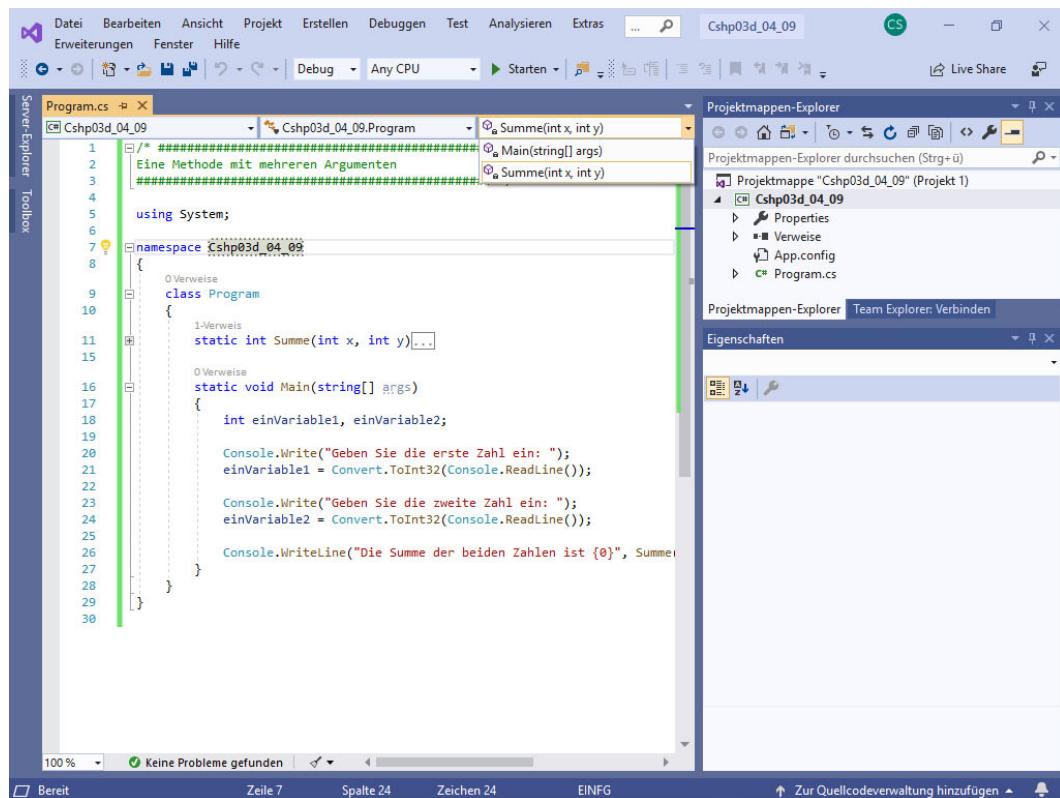


Abb. 4.3: Die Liste der Methoden (oben in der Mitte der Abbildung)

Informationen zu den Argumenten werden übrigens auch dann angezeigt, wenn Sie den Namen der Methode und die Klammer (im Quelltext eingeben. Dann erscheinen die Parameterliste und auch der Rückgabetyp in einer Quickinfo.



In diesem Video stellen wir Ihnen die Hilfen zum Arbeiten mit Methoden im praktischen Einsatz vor.

www.dfz.media/3cuu12

Video 4.1: Hilfen zum Arbeiten mit Methoden

Abschließend noch ein Hinweis:

Lassen Sie sich von den Begriffen Parameter und Argument nicht verwirren. Wichtig ist vor allem, dass Sie an eine Methode Werte übergeben können. Die Anzahl und der Typ dieser Werte werden bei der Vereinbarung der Methode angegeben und dienen dort quasi als Platzhalter. Beim Aufruf geben Sie dann die konkreten Werte an, die an die Methode übergeben werden sollen. Dabei müssen Typ und Anzahl mit der Liste im Kopf der Methode übereinstimmen. Der Bezeichner dagegen ist beliebig.

Zusammenfassung

Eine Methode wird einmal erstellt und kann dann beliebig oft im Quelltext aufgerufen werden.

Eine Methode besteht aus dem Kopf und dem Körper. Der Kopf enthält den Typ des Rückgabewertes, den Namen der Methode und eine Liste der Parameter. Der Körper enthält die Anweisungen, die die Methode ausführen soll.

Der Aufruf einer Methode erfolgt durch den Namen der Methode gefolgt von einer Liste der Argumente.

Auch bei Methoden ohne Parameter müssen Sie die Klammern () angeben – sowohl bei der Vereinbarung als auch beim Aufruf.

Mit der Anweisung `return` geben Sie einen Wert aus einer Methode zurück. Der Rückgabewert steht hinter dem `return`.

Wenn Sie mehrere Parameter beziehungsweise Argumente verwenden, müssen Sie die einzelnen Werte durch Kommas trennen.

Aufgaben zur Selbstüberprüfung

- 4.1 Welchen Wert liefert die folgende Methode zurück? Sehen Sie sich bitte vor der Antwort den Kopf der Methode genau an.

```
Summe(int a, int b)
{
    return (a + b);
}
```

- 4.2 Wie viele Werte können Sie mit Standardtechniken aus einer Methode zurückliefern lassen? Wie können Sie mehrere Werte zurückgeben lassen?

- 4.3 Sehen Sie sich bitte den folgenden Methodenkopf an. Was stimmt nicht? Wie muss der Methodenkopf korrekt lauten?

```
char Buchstabe(char a, b)
```

- 4.4 Schreiben Sie ein kurzes Programm, das eine Methode `Produkt()` enthält, die zwei Zahlen `x` und `y` miteinander multipliziert und das Ergebnis zurückgibt. Überprüfen Sie die Funktionsweise der Methode `Produkt()` durch einen Aufruf aus der Methode `Main()`.

- 4.5 Schreiben Sie eine Methode `Quadrat()`, die das Quadrat einer Zahl x berechnet. Setzen Sie zum Berechnen aber nicht den Operator `*` ein, sondern rufen Sie in der Methode `Quadrat()` die Methode `Produkt()` aus der Aufgabe 4.4 auf. Schreiben Sie eine passende Methode `Main()`, um die korrekte Funktion zu überprüfen.

Schlussbetrachtung

Sie dürfen jetzt ein wenig stolz auf sich sein. Sie haben in diesem Studienheft einige schon recht komplexe Anweisungen kennengelernt und auch ausprobiert.

Sie können jetzt Programme erstellen, die flexibel auf Bedingungen reagieren, Anweisungen wiederholen und auch Anweisungen in Methoden mehrfach einsetzen, ohne sie erneut eingeben zu müssen.

Vielleicht kommt Ihnen das eine oder andere in diesem Studienheft noch etwas seltsam und fremd vor. Nehmen Sie sich dann die Zeit und arbeiten Sie die entsprechenden Kapitel noch einmal in aller Ruhe durch. Probieren Sie dabei auch die Beispielcodes aus und haben Sie keine Scheu, Änderungen an den Programmen vorzunehmen. Viele scheinbar schwierige Konstruktionen werden durch wiederholtes Ausprobieren sehr viel leichter verständlich.

Denken Sie daran: Nur Übung macht den Meister!

Und vor allem: Lassen Sie sich Zeit. Beim Programmieren geht es nicht um Geschwindigkeit, sondern vor allem um ein stabiles und korrekt funktionierendes Endergebnis. Denken Sie deshalb auch erst einmal in Ruhe nach, wie Sie überhaupt ein Problem mit einem Programm lösen wollen, und fangen Sie nicht an, „wild“ drauflos zu programmieren.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Wenn Sie zwei Zahlen auf Gleichheit testen wollen, benutzen Sie den Operator `==`. Mit dem Operator `>` können Sie prüfen, ob die Zahl links vom Operator größer ist als die Zahl rechts vom Operator. Alternativ können Sie auch den Operator `<` verwenden. Dann müssen allerdings die Operanden getauscht werden.
- 1.2
- a) wahr
 - b) falsch
 - c) wahr
 - d) falsch
 - e) wahr
 - f) wahr
 - g) falsch
- 1.3
- | A | B | $!(A B)$ |
|--------|--------|-----------|
| falsch | falsch | wahr |
| falsch | wahr | falsch |
| wahr | falsch | falsch |
| wahr | wahr | falsch |

Kapitel 2

- 2.1 Die Lösung könnte so aussehen:

```
/* ##### Lösung II.1 #####
using System;
namespace LoesIII
{
    class Program
    {
        static void Main(string[] args)
        {
            int x, y;

            Console.Write("Geben Sie den Wert für x ein: ");
            x = Convert.ToInt32(Console.ReadLine());
```

```

Console.WriteLine("Geben Sie den Wert für y ein: ");
y = Convert.ToInt32(Console.ReadLine());

if (x>y)
    Console.WriteLine("x ist größer als y.");
else
    Console.WriteLine("y ist größer oder gleich x.");
}
}
}

```

- 2.2 Die fehlenden Teile sind im folgenden Quelltext fett markiert.

```

/* ##### LÖSUNG II.2 #####
using System;

namespace LoesII2
{
    class Program
    {
        static void Main(string[] args)
        {
            int essenWahl;

            Console.WriteLine("Wählen Sie bitte ein Essen aus:
\n");
            Console.WriteLine("1 Jägerschnitzel mit Pommes");
            Console.WriteLine("2 Currywurst mit Pommes");
            Console.WriteLine("3 Bratwurst mit Brötchen\n");
            Console.Write("Was möchten Sie essen? ");

            essenWahl = Convert.ToInt32(Console.ReadLine());

            switch (essenWahl)
            {
                case 1:
                    Console.WriteLine("Sie haben ein Jägerschnitzel
mit Pommes gewählt.");
                    break;
                case 2:
                    Console.WriteLine("Sie haben eine Currywurst mit
Pommes gewählt.");
                    break;
                case 3:
                    Console.WriteLine("Sie haben eine Bratwurst mit
Brötchen gewählt.");
                    break;
            }
        }
    }
}

```

Kapitel 3

- 3.1 Bei der `while`-Schleife wird die Bedingung vor dem Schleifendurchlauf geprüft, bei der `do ... while`-Schleife erst nach dem Schleifendurchlauf. Daher wird die `do ... while`-Schleife auch in jedem Fall mindestens einmal ausgeführt.
- 3.2 Die Schleife gibt endlos die Zahl 10 aus. Es fehlt die Veränderung der Schleifenvariablen.
- 3.3 Die Lösung könnte so aussehen:

```
/*
#####
Lösung III.3
#####
*/

using System;

namespace LoesIII3
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, k;
            bool flagVariable = true;
            i = 0;
            k = 0;

            while ((i <= 5) && (flagVariable == true))
            {
                Console.Write("Geben Sie eine 1 zum Abbruch ein.");
                k = Convert.ToInt32(Console.ReadLine());
                if (k == 1)
                    flagVariable = false;
                else
                    i++;
            }

            Console.WriteLine("Schleife beendet.");
        }
    }
}
```

Kapitel 4

- 4.1 Die Methode liefert keinen Wert zurück, da sich das Programm gar nicht übersetzen lässt. Es fehlt der Rückgabetyp für die Methode.
- 4.2 Eine Methode kann mit Standardtechniken nur einen Wert zurückliefern. Um mehrere Werte zurückzuliefern, müssen Sie ein Tupel verwenden.
- 4.3 In der Liste der Parameter fehlt die Typangabe für den zweiten Parameter. Korrekt wäre

```
char Buchstabe(char a, char b)
```

4.4 Die Lösung könnte so aussehen:

```
/* ##### Lösung IV.4 #####
Lösung IV.4
##### */

using System;

namespace LoesIV4
{
    class Program
    {
        static int Produkt(int x, int y)
        {
            return (x * y);
        }

        static void Main(string[] args)
        {
            int einVariable1, einVariable2;
            Console.Write("Geben Sie den Wert für Zahl 1 ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Geben Sie den Wert für Zahl 2 ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("Das Produkt der Zahlen ist {0}",
                Produkt(einVariable1, einVariable2));
        }
    }
}
```

4.5 Die Lösung könnte so aussehen:

```
/* ##### Lösung IV.5 #####
Lösung IV.5
##### */

using System;

namespace LoesIV5
{
    class Program
    {
        static int Produkt(int x, int y)
        {
            return (x * y);
        }

        static int Quadrat(int x)
        {
            return (Produkt(x, x));
        }
    }
}
```

```
static void Main(string[] args)
{
    int einVariable;
    Console.Write("Geben Sie eine Zahl ein: ");
    einVariable = Convert.ToInt32(Console.ReadLine());

    //bitte in einer Zeile eingeben
    Console.WriteLine("Das Quadrat der Zahl ist {0}",
        Quadrat(einVariable));
}
```

B. Glossar

Abbruchbedingung	Bei einer Schleife ist die Abbruchbedingung die Bedingung, die die Schleife steuert. Solange die Abbruchbedingung logisch „wahr“ ist, wird die Schleife ausgeführt.
Abfrage	Über eine Abfrage werden Daten aus einer Datenbank anhand bestimmter Kriterien selektiert.
Alternative Verzweigung	Bei einer alternativen Verzweigung werden sowohl Anweisungen vorgegeben, die ausgeführt werden sollen, wenn die Bedingung zutrifft, als auch Anweisungen, die ausgeführt werden sollen, wenn die Bedingung nicht zutrifft.
Anweisungsteil	Der Anweisungsteil ist der Teil einer Methode, der die Anweisungen enthält. Der Anweisungsteil wird immer durch geschweifte Klammern umfasst. Die Anweisungen im Anweisungsteil müssen durch ein Semikolon getrennt werden.
Argument	Ein Argument wird beim Aufruf einer Methode in den runden Klammern an die Methode übergeben. Statt des Begriffs Argument wird häufig auch der Begriff Parameter benutzt. Streng genommen bezeichnet der Begriff Parameter aber die Angaben im Kopf der Methode.
Arithmetik	Die Arithmetik ist ein Teilgebiet der Mathematik, das sich mit Zahlen und den für sie geltenden Rechenregeln befasst.
Ausnahme	Ausnahme ist die deutsche Übersetzung für <i>Exception</i> .
Ausnahmebehandlung	Durch die Ausnahmebehandlung können Sie in einem Programm auf Laufzeitfehler reagieren. Die Ausnahmebehandlung wird auch <i>Exception Handling</i> genannt.
Bedingung	Eine Bedingung steuert den weiteren Ablauf eines Programms – zum Beispiel bei einer Verzweigung oder bei einer Schleife.
Bezeichner	Ein Bezeichner ist der Name eines Datenobjekts. Für die Vergabe von Bezeichnern gibt es feste Regeln. So dürfen Sie zum Beispiel keine Schlüsselwörter verwenden und müssen einen Bezeichner mit einem Buchstaben oder dem Unterstrich _ beginnen.
Einfache Verzweigung	Bei einer einfachen Verzweigung werden Anweisungen nur dann ausgeführt, wenn eine Bedingung zutrifft.

Eingabeaufforderung	Die Eingabeaufforderung ist ein Teil von Windows, über den Konsolenprogramme gestartet werden können.
Endlosschleife	Eine Endlosschleife ist eine Schleife, die nie beendet wird. Endlosschleifen entstehen in der Regel durch Programmierfehler.
Exception	Eine <i>Exception</i> – eine Ausnahme – wird durch nicht definierte Zustände bei der Ausführung eines Programms ausgelöst. Dazu gehören zum Beispiel Divisionen durch Null bei ganzen Zahlen oder fehlgeschlagene Speicher-reservierungen.
Exception Handler	Der <i>Exception Handler</i> besteht aus den Anweisungen, die beim Auftreten einer Ausnahme ausgeführt werden sollen.
Exception Handling	<i>Exception Handling</i> ist die englische Bezeichnung für Ausnahmebehandlung.
Flag	Als <i>Flag</i> wird bei der Programmierung eine Variable bezeichnet, die einen Zustand eindeutig anzeigt. Dafür werden häufig die Werte 0 und 1 beziehungsweise true und false benutzt. <i>Flag</i> bedeutet übersetzt „Flagge“.
Fußgesteuerte Schleife	Bei einer fußgesteuerten Schleife wird die Abbruchbedingung nach jedem Schleifendurchlauf geprüft. Eine fußgesteuerte Schleife wird daher mindestens einmal durchlaufen.
Geschachtelte Verzweigung	Bei einer geschachtelten Verzweigung werden mehrere Verzweigungen ineinander geschachtelt. Der if-Zweig einer Verzweigung enthält dann zum Beispiel selbst wieder eine Verzweigung.
Kontrollstruktur	Über eine Kontrollstruktur können Sie den Ablauf eines Programms steuern – zum Beispiel in Abhängigkeit von einer Bedingung bestimmte Anweisungen ausführen oder Anweisungen wiederholen.
Kopfgesteuerte Schleife	Bei einer kopfgesteuerten Schleife wird die Abbruchbedingung vor jedem Schleifendurchlauf geprüft.
Logische Verneinung	Siehe Logisches NICHT.
Logisches NICHT	Beim logischen NICHT wird ein Wert in sein Gegenteil verkehrt.
Logisches ODER	Beim logischen ODER werden zwei Ausdrücke miteinander verknüpft. Das Ergebnis ist wahr, wenn mindestens einer der beiden Ausdrücke wahr ist.

Logisches UND	Beim logischen UND werden zwei Ausdrücke miteinander verknüpft. Das Ergebnis ist nur dann wahr, wenn beide Ausdrücke wahr sind.
Main()	Main() ist die Hauptmethode eines C#-Programms.
Mehrfachauswahl	Bei der Mehrfachauswahl können zahlreiche Alternativen berücksichtigt werden. Die Mehrfachauswahl wird in C# durch die switch ... case-Konstruktion umgesetzt.
Methoden	Methoden beschreiben das Verhalten eines Objekts.
Modulo	Modulo liefert den Rest einer Division.
Nachlauf	Der Nachlauf ist Teil einer for-Schleife. Er wird nach jedem Schleifendurchlauf ausgeführt. Im Nachlauf wird in der Regel die Schleifenvariable verändert.
Objektorientierte Programmierung	Bei der objektorientierten Programmierung werden Daten- und Verhaltensaspekte gemeinsam betrachtet. Das wesentliche Element der objektorientierten Programmierung ist das Objekt.
Operand	Ein Operand ist der Teil eines Ausdrucks, auf den ein Operator wirkt.
Operator	Ein Operator ist ein Zeichen, das eine bestimmte Operation auslöst – zum Beispiel eine Rechenoperation. C# kennt verschiedene Arten von Operatoren – zum Beispiel <ul style="list-style-type: none"> • arithmetische Operatoren, • logische Operatoren und • Vergleichsoperatoren.
Parameter	Parameter geben an, welche Werte von einer Methode verarbeitet werden können. Sie werden im Kopf der Methode in runden Klammern angegeben. Der Begriff Parameter wird häufig auch für Argumente benutzt.
Quickinfos	Quickinfos sind kurze Hinwestexte. Sie werden normalerweise angezeigt, wenn der Mauszeiger einen kurzen Moment auf einem bestimmten Element stehen bleibt.
Reservierte Wörter	Reservierte Wörter sind Zeichenketten, die intern vom Compiler verwendet werden. Solche Zeichenketten dürfen Sie nicht selbst benutzen. Reservierte Wörter sind zum Beispiel using oder while.

Schleife	Über eine Schleife können Sie Anweisungen wiederholen lassen.
Schleifenfuß	Der Schleifenfuß ist das Ende einer Schleife.
Schleifenkörper	Der Schleifenkörper enthält die Anweisungen, die von einer Schleife ausgeführt werden sollen.
Schleifenkopf	Der Schleifenkopf ist der Anfang einer Schleife.
Syntaxfehler	Syntaxfehler sind Verstöße gegen die Regeln der Programmiersprache – zum Beispiel ein fehlendes Semikolon oder eine falsche Klammer. Syntaxfehler werden vom Compiler bei der Übersetzung des Programms gemeldet.
Testausdruck	Der Testausdruck ist Teil einer <code>for</code> -Schleife. Er steuert die Wiederholung der Schleife.
Tupel	Ein Tupel ist eine geordnete Menge von Werten.
Vorlauf	Der Vorlauf ist Teil einer <code>for</code> -Schleife. Er wird einmal vor der Ausführung der Schleife ausgeführt. Im Vorlauf werden in der Regel Variablen initialisiert.
Zählschleife	Zählschleife ist ein anderer Begriff für eine <code>for</code> -Schleife.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch.*
Spracheinführung, Objektorientierung, Programmiertechniken.
8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019.* Ideal für Programmieranfänger.
6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 2.1	Eine einfache if-Verzweigung	16
Abb. 2.2	Eine if ... else-Verzweigung	20
Abb. 2.3	Vergleich von if ... else (links) und if (rechts)	21
Abb. 2.4	Eine geschachtelte Verzweigung	23
Abb. 2.5	Geschachtelte Verzweigungen mit if ... else	24
Abb. 2.6	Die Entscheidungskette aus Code 2.4	26
Abb. 2.7	Die switch ... case-Konstruktion	27
Abb. 2.8	Darstellung der möglichen Fälle aus Code 2.6	32
Abb. 3.1	Die while-Schleife	37
Abb. 3.2	Die do ... while-Schleife	40
Abb. 3.3	Die for-Schleife	44
Abb. 4.1	Der Programmablauf für Code 4.2	62
Abb. 4.2	Ein ausgeblendeter Methodenkörper (links in der Mitte der Abbildung am Mauszeiger)	73
Abb. 4.3	Die Liste der Methoden (oben in der Mitte der Abbildung)	74

E. Tabellenverzeichnis

Tab. 1.1	Vergleichsoperatoren von C#	3
Tab. 1.2	Logische Operatoren von C#	5
Tab. 1.3	Wahrheitstabelle für das logische ODER	7
Tab. 1.4	Wahrheitstabelle für das logische UND	8
Tab. 2.1	Syntax der if-Verzweigung	19
Tab. 2.2	Syntax der if ... else-Verzweigung	22

F. Codeverzeichnis

Code 1.1	Beispiele für Vergleichsoperatoren	4
Code 1.2	Logisches NICHT	6
Code 1.3	Logisches ODER	8
Code 1.4	Logisches UND	9
Code 1.5	Kombination von logischen Operatoren mit Vergleichsoperatoren	10
Code 2.1	Einfache if-Verzweigung	16
Code 2.2	if-Verzweigung mit Anweisungsblock	18
Code 2.3	if ... else-Verzweigung	22
Code 2.4	Entscheidungsketten mit if ... else	25
Code 2.5	switch ... case-Konstruktion	29
Code 2.6	Verschachteltes switch ... case	31
Code 3.1	while-Schleife	38
Code 3.2	do ... while-Schleife	41
Code 3.3	Die umgebaute Schleife	42
Code 3.4	Noch eine umgebaute Schleife	43
Code 3.5	for-Schleife	45
Code 3.6	Von 0 bis 10 mit while	45
Code 3.7	Eine sehr kompakte for-Schleife	46
Code 3.8	break in einer Schleife	49
Code 3.9	Schleifenabbruch über eine Flag-Variable	51
Code 3.10	continue in einer Schleife	53
Code 3.11	if-Konstruktion statt continue	54
Code 3.12	Das Abfangen von Eingabefehlern	55
Code 4.1	Einfache Textausgabe	60
Code 4.2	Einfache Textausgabe mit Methoden	61
Code 4.3	Zwei Methoden mit Rückgabewert	64
Code 4.4	Undefinierte Rückgabe aus einer Methode (der Code lässt sich nicht übersetzen)	66
Code 4.5	Unterschiedliche Typen (der Code lässt sich ebenfalls nicht übersetzen)	66
Code 4.6	Das klappt nicht!	67
Code 4.7	Rückgabe mehrerer Werte über ein Tupel	68
Code 4.8	Eine Methode mit Argument	71
Code 4.9	Eine Methode mit zwei Argumenten	72

G. Medienverzeichnis

Video 4.1 Hilfen zum Arbeiten mit Methoden	74
--	----

H. Sachwortverzeichnis

A	
Abbruchbedingung	36
Abfangen	
von ungültigen Eingaben	54
Anweisung	
break	47
continue	52
Anweisungsblock	18
Argument	61
Ausnahmebehandlung	54
Auswertungsblock	27
B	
Bedingung	17
C	
case-Marke	27
case-Zweig	27
E	
Endlosschleife	36
F	
Flag-Variable	50
K	
Klassenmethode	59
Kontrollstruktur	15
M	
Mehrfachauswahl	
mit switch ... case	26
Methode	58
Main()	60
mit Argumenten	69
mit Rückgabewert	63
Methodenkörper	59
N	
Nachlauf	44
O	
ODER	
logisches	7
Operator	
logischer	5
P	
Parameter	59
R	
Rückgabetyp	59
void	60
S	
Schleife	36
fußgesteuerte mit do ... while	39
kopfgesteuerte mit while	37
Schleifenfuß	36
Schleifenkopf	36
Schleifenkörper	36
Schleifenvariable	38
Schlüsselwort	
return	63
T	
Testausdruck	43
Tipp	
zum Arbeiten mit Methoden	73
Tupel	68
U	
UND	
logisches	7, 8
V	
Vergleichsoperator	3
==	3
Verneinung	
logische	5, 6

Verzweigung	
geschachtelte	22
if-	15
if ... else-	20
Vorlauf	43

Z

Zählschleife	
mit for	43
Zuweisungsoperator	
=	3

I. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP03D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

1. Schreiben Sie ein Programm, das eine Jahreszahl über die Tastatur abfragt und dann ausgibt, ob das Jahr ein Schaltjahr ist.

Die Überprüfung können Sie mit folgenden Regeln durchführen:

Ein Jahr ist kein Schaltjahr, wenn die Jahreszahl nicht durch 4 teilbar ist.

Ein Jahr ist ein Schaltjahr, wenn die Jahreszahl durch 4, aber nicht durch 100 teilbar ist.

Es ist ebenfalls ein Schaltjahr, wenn die Jahreszahl gleichzeitig durch 4, durch 100 und durch 400 teilbar ist.

Ein Beispiel:

Das Jahr 1964 war ein Schaltjahr. Die Jahreszahl lässt sich durch 4, aber nicht durch 100 teilen.

Das Jahr 1900 war kein Schaltjahr. Die Jahreszahl lässt sich zwar durch 4 und auch durch 100 teilen, aber nicht durch 400.

Sie können für die Überprüfung der Teilbarkeit den Modulo-Operator `%` und `if ... else`-Verzweigungen benutzen. Zur Erinnerung: Wenn eine Zahl `x` nicht glatt durch `y` teilbar ist, dann liefert der Ausdruck `(x % y)` einen Wert größer als 0.

Setzen Sie bei der Überprüfung der Teilbarkeit eine weitere Variable ein, die markiert, ob das Jahr ein Schaltjahr ist oder nicht. Werten Sie diese Variable am Ende des Programms aus und lassen Sie dann auf dem Bildschirm ausgeben, ob es sich um ein Schaltjahr handelt oder nicht.

Verwenden Sie bitte folgenden Programmkopf:

```
/* #####  
Einsendeaufgabe 3.1  
##### */
```

30 Pkt.

2. Schreiben Sie ein Programm, das von 1 bis 5 zählt und die Zahlen nebeneinander durch Kommas getrennt ausgibt. Vor der ersten Zahl und nach der letzten Zahl darf kein Komma stehen. Die Ausgabe soll durch eine `for`-Schleife erfolgen und so aussehen:

`1, 2, 3, 4, 5`

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.2
##### */
```

10 Pkt.

3. Erstellen Sie eine `while`-Schleife, die für die Zahlen 1 bis 25 jeweils das Doppelte des Wertes ausgibt. Für die Zahl 2 soll also der Wert 4 ausgegeben werden, für die Zahl 3 der Wert 6 und so weiter.

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.3
##### */
```

10 Pkt.

4. Programmieren Sie einen einfachen Taschenrechner. Er soll zwei Werte von der Tastatur einlesen und das Ergebnis einer Rechenoperation auf dem Bildschirm ausgeben. Dabei sollen auch Kommazahlen verarbeitet werden können. Als Rechenoperationen sollen Addition, Subtraktion, Division und Multiplikation möglich sein. Die Rechenoperationen sollen als eigene Methoden erstellt werden.

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.4
##### */
```

30 Pkt.

5. Erweitern Sie den Taschenrechner aus der Aufgabe 4 so, dass er Potenzen berechnen kann. Der erste eingelesene Wert soll dabei die Basis bilden und der zweite eingelesene Wert den Exponenten. Wenn Sie die Zahlen 2 und 3 eingelesen haben, soll der Taschenrechner also 2^3 rechnen.

Erstellen Sie für das Berechnen der Potenz eine eigene Methode. Die Potenz soll durch eine Schleife errechnet werden.

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.5
##### */
```

20 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Bezugsrahmen von Variablen – Arrays und Strings
Strukturen, Tupel und Aufzählungstypen

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP04D

Objektorientierte Software-Entwicklung mit C#

Bezugsrahmen von Variablen – Arrays und Strings Strukturen, Tupel und Aufzählungstypen

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Bezugsrahmen von Variablen – Arrays und Strings Strukturen, Tupel und Aufzählungstypen

Inhaltsverzeichnis

Einleitung	1
1 Bezugsrahmen von Variablen	3
1.1 Lokale Variablen	3
1.2 Klassenvariablen	10
1.3 Regeln für das Arbeiten mit lokalen Variablen und Klassenvariablen	13
Zusammenfassung	15
2 Arrays	18
2.1 Zugriff auf die Elemente eines Arrays	21
2.2 Mehrdimensionale Arrays	27
2.3 Übergabe von Arrays an Methoden	30
2.4 Kopieren von Arrays	32
2.5 Sonderfunktionen für Arrays	35
Zusammenfassung	36
3 Zeichenketten	40
3.1 Zeichenketten vergleichen	41
3.2 Zeichenketten verändern	43
3.3 Zeichenketten durchsuchen	45
3.4 Die Klasse StringBuilder	46
Zusammenfassung	48
4 Strukturen, Tupel und Aufzählungstypen	50
4.1 Strukturen	50
4.1.1 Vereinbarung einer Struktur	50
4.1.2 Zugriff auf Mitglieder einer Struktur	52
4.1.3 Arbeiten mit Strukturen	54
4.2 Tupel	57
4.3 Aufzählungstypen	59
Zusammenfassung	61
Schlussbetrachtung	63

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	64
B.	Glossar	66
C.	Literaturverzeichnis	68
D.	Abbildungsverzeichnis	69
E.	Tabellenverzeichnis	70
F.	Codeverzeichnis	71
G.	Sachwortverzeichnis	72
H.	Einsendeaufgabe	73

Einleitung

In diesem Studienheft beschäftigen wir uns zuerst mit dem Gültigkeitsbereich von Variablen. Sie werden lernen, welche unterschiedlichen Gültigkeitsbereiche es gibt und wann Sie welchen Gültigkeitsbereich verwenden.

Danach erfahren Sie etwas über Arrays. Diese Arrays bieten Ihnen die Möglichkeit, mehrere Variablen eines Datentyps in einer Art Reihe zu verarbeiten.

Anschließend beschäftigen wir uns intensiver mit den Zeichenketten.

Danach werfen wir noch einen Blick auf Strukturen, Tupel und Aufzählungstypen. Mit diesen Konstruktionen können Sie bestimmte Arten von Daten sehr viel einfacher verarbeiten als bisher gewohnt.

Im Einzelnen lernen Sie in diesem Studienheft,

- was lokale Variablen und Klassenvariablen sind,
- wie Sie lokale Variablen und Klassenvariablen vereinbaren,
- wann Sie lokale Variablen und Klassenvariablen verwenden sollten,
- was Arrays sind und wie Sie Arrays vereinbaren,
- wie Sie mit Arrays arbeiten,
- wie Sie mehrdimensionale Arrays anlegen,
- wie Sie Arrays an eine Methode übergeben,
- welche Besonderheiten Sie beim Kopieren von Arrays beachten müssen,
- wie Sie Arrays sortieren und durchsuchen,
- was sich genau hinter dem Datentyp `string` verbirgt,
- wie Sie Zeichenketten verändern und durchsuchen,
- welche Vorteile die Klasse `StringBuilder` bietet,
- was Strukturen sind und wie Sie Strukturen vereinbaren,
- wie Sie auf die Mitglieder einer Struktur zugreifen,
- was Tupel sind und wie Sie mit Tupeln arbeiten und
- was sich hinter Aufzählungstypen verbirgt.

Christoph Siebeck

1 Bezugrahmen von Variablen

In diesem Kapitel werden wir uns mit den Gültigkeitsbereichen von Variablen beschäftigen – dem **Bezugrahmen** oder **Kontext**.

Der Bezugrahmen ist der Bereich eines Programms, in dem ein bestimmtes Datenobjekt angesprochen werden kann.



Grundsätzlich werden zwei Bezugrahmen unterschieden: **lokal** und für die gesamte **Klasse**. Der Bezugrahmen hängt dabei von der Stelle im Quelltext ab, an der die Variable vereinbart wird.

1.1 Lokale Variablen

Schauen wir uns zuerst die lokalen Variablen an.

Eine **lokale Variable** wird innerhalb eines Anweisungsblocks vereinbart. Sie ist nur in dem Block bekannt, in dem sie vereinbart wird.



Sehen wir uns dazu ein einfaches Beispiel an:

```
/*
#####
Beispiel für eine lokale Variable
#####
*/
using System;

namespace Cshp04d_01_01
{
    class Program
    {
        static void GibVarAus()
        {
            int varA = 4;
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }

        static void Main(string[] args)
        {
            GibVarAus();
        }
    }
}
```

Code 1.1: Beispiel für eine lokale Variable

In diesem Beispiel wird innerhalb des Anweisungsblocks der Methode `GibVarAus()` die Variable `varA` mit dem Wert 4 initialisiert und anschließend ausgegeben. Diese Technik haben Sie bereits eingesetzt. Es handelt sich also um nichts Neues.

Wenn Sie versuchen, auf die Variable innerhalb der Methode `Main()` zuzugreifen, wird sich der Compiler beschweren, dass der Name `varA` im aktuellen Kontext nicht vorhanden ist. Ändern wir dazu das Beispiel aus dem vorigen Code ein wenig ab und versuchen, den Wert der Variablen in der Methode `Main()` auszugeben.

```
/*
#####
Beispiel für einen nicht erlaubten Zugriff
Der Code kann nicht übersetzt werden!
#####
*/

using System;

namespace Cshp04d_01_02
{
    class Program
    {
        static void GibVarAus()
        {
            int varA = 4;
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }

        static void Main(string[] args)
        {
            GibVarAus();
            //DAS GEHT NICHT!!!
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }
    }
}
```

Code 1.2: Beispiel für einen nicht erlaubten Zugriff auf eine lokale Variable

Wir haben lediglich die Anweisung

```
Console.WriteLine("Der Wert der Variablen ist {0}.", varA);
```

in der Methode `Main()` hinzugefügt.

Und prompt beschwert sich der Compiler, der Name `varA` sei im aktuellen Kontext nicht vorhanden. Er kennt die Variable schlicht und einfach nicht. Jetzt fragen Sie sich, warum diese Fehlermeldung erscheint? Die Variable `varA` ist doch völlig korrekt im Programm mit der Zeile

```
int varA = 4;
```

in der Methode `GibVarAus()` vereinbart worden.

Die Fehlermeldung lässt sich leicht nachvollziehen, wenn Sie sich noch einmal die Definition einer lokalen Variablen ansehen: Eine lokale Variable ist nur in dem Anweisungsblock bekannt, in dem sie vereinbart wurde. Bei den Methoden `Main()` und `GibVarAus()` handelt es sich aber um **zwei** getrennte Anweisungsblöcke. Das erkennen Sie daran, dass sowohl die Anweisungen von `Main()` als auch die Anweisungen von `GibVarAus()` von geschweiften Klammern umfasst werden.

Für unser Beispiel bedeutet das: Die Variable `varA` gilt nur innerhalb des Anweisungsblocks der Methode `GibVarAus()`, oder etwas anders ausgedrückt: `varA` ist eine lokale Variable der Methode `GibVarAus()`. Sie ist damit nur innerhalb dieser einen Methode bekannt. Die Methode `Main()` kennt die Variable `varA` **nicht**.

Wir können natürlich die Variable `varA` auch in der `Main()`-Methode vereinbaren. Das ist ohne Weiteres möglich, da die Vereinbarungen von lokalen Variablen ja nur innerhalb eines Anweisungsblocks gelten. In einem anderen Anweisungsblock können Sie also ohne Probleme eine andere Variable mit demselben Namen vereinbaren.

Das geänderte Beispiel sieht dann so aus:

```
/*
#####
Beispiel für zwei lokale Variablen
Auch dieser Code wird nicht übersetzt!
#####
*/

using System;

namespace Cshp04d_01_03
{
    class Program
    {
        static void GibVarAus()
        {
            int varA = 4;
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }

        static void Main(string[] args)
        {
            int varA;

            GibVarAus();
            //DAS GEHT NICHT!!!
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }
    }
}
```

Code 1.3: Code 1.2 – jetzt mit einer anderen Fehlermeldung

Zwar verschwindet jetzt die Meldung, dass der Name `varA` im aktuellen Kontext nicht vorhanden ist, dafür beschwert sich der Compiler, dass die Variable `varA` ohne Zuweisung verwendet wird. Diese Meldung erscheint, obwohl doch scheinbar mit der Zeile

```
int varA = 4;
```

die Zuweisung eines Wertes erfolgt ist.



Zur Auffrischung:

Sie können nur dann lesend auf eine Variable zugreifen, wenn sie vor dem ersten lesenden Zugriff einen definierten Wert erhalten hat. Die Zuweisung dieses ersten Wertes nennt man **Initialisierung**.

Bei lokalen Variablen erfolgt die Initialisierung **nicht** automatisch.

Auch für dieses Verhalten gibt es eine Erklärung: Die Variablen haben zwar denselben Namen, sind aber völlig unabhängig voneinander, da sie ja lokal in unterschiedlichen Anweisungsblöcken vereinbart wurden.



Beim Verlassen eines Anweisungsblocks werden alle lokalen Variablen, die in ihm vereinbart wurden, gelöscht.

Das bedeutet für unser Beispiel: Die Variable `varA` der Methode `GibVarAus()` gilt nur innerhalb dieser Methode. Sobald die Methode verlassen wird, wird die Variable `varA` aus der Methode und damit auch ihr Wert gelöscht. In der Methode `Main()` wird dann die Variable `varA` benutzt, die innerhalb der `Main()`-Methode vereinbart wurde. Dieser Variablen haben wir aber **keinen** Wert zugewiesen. Daher erscheint auch die Fehlermeldung vom Compiler.

Damit sich das Beispiel ausführen lässt, müssen Sie also der Variablen `varA` in der Methode `Main()` ebenfalls noch einen Wert zuweisen. Der Code könnte dann so aussehen:

```
/*
#####
Beispiel für zwei lokale Variablen
Jetzt läuft das Programm
#####
*/
using System;

namespace Cshp04d_01_04
{
    class Program
    {
        static void GibVarAus()
        {
            int varA = 4;
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }

        static void Main(string[] args)
        {
            int varA = 10;
            GibVarAus();
        }
    }
}
```

```
        Console.WriteLine("Der Wert der Variablen ist {0}.",  
    varA);  
    }  
}  
}
```

Code 1.4: Code 1.3 – jetzt ohne Fehlermeldung

An den unterschiedlichen Ausgaben wird jetzt auch sofort klar, dass es sich bei den beiden Variablen um zwei verschiedene Datenobjekte handelt, die nur den Namen gemeinsam haben. In der Methode `GibVarAus()` wird der Wert 4 für `varA` ausgegeben, in der Methode `Main()` dagegen der Wert 10.

Wenn Sie mehrere Variablen mit demselben Namen in einem Programm vereinbaren, greifen Sie immer auf die aktuell gültige – die lokalste – Variable zu.



Da der Bezugsrahmen für eine Variable durch einen Anweisungsblock festgelegt wird, können Sie auch noch engere Bezugsrahmen als eine Methode definieren – zum Beispiel Anweisungsblöcke in einer Verzweigung. Sehen wir uns dazu das folgende Beispiel an. Es soll zwei Zahlen einlesen und das Produkt der beiden Zahlen berechnen, wenn die erste Zahl kleiner ist als die zweite Zahl.

```
/* ##### Beispiel für einen sehr engen Bezugsrahmen
Das Programm lässt sich so nicht übersetzen!
##### */  
  
using System;  
  
namespace Cshp04d_01_05  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int zahl1,zahl2;  
  
            Console.Write("Geben Sie bitte die erste Zahl ein: ");  
            zahl1 = Convert.ToInt32(Console.ReadLine());  
            Console.Write("Geben Sie bitte die zweite Zahl ein: ");  
            zahl2 = Convert.ToInt32(Console.ReadLine());  
  
            if (zahl1<zahl2)  
            {  
                //produkt wird mit sehr engem Bezugsrahmen vereinbart  
                int produkt;  
                produkt = zahl1 * zahl2;  
                //bitte in einer Zeile eingeben  
                Console.WriteLine("Das Produkt der beiden Zahlen  
ist {0}", produkt);  
            }  
        }  
    }  
}
```

```
//Hier ist produkt unbekannt
//bitte in einer Zeile eingeben
Console.WriteLine("Das Produkt der beiden Zahlen
ist {0}", produkt);
    }
}
```

Code 1.5: Ein Beispiel für einen sehr engen Bezugsrahmen

Auch hier wird sich der Compiler beschweren. Die Variable `produkt` in der letzten Ausgabeanweisung ist im aktuellen Kontext nicht vorhanden.

Um den Fehler zu finden, müssen Sie jetzt schon sehr genau hinsehen. `produkt` wurde innerhalb des `if`-Anweisungsblocks vereinbart und ist deshalb auch nur innerhalb dieses Blocks gültig.

Erst wenn Sie die letzte Ausgabeanweisung löschen oder auskommentieren, läuft das Programm. Die beiden Variablen `zahl1` und `zahl2` sind auch innerhalb des `if`-Anweisungsblocks gültig, weil die `if`-Anweisung innerhalb des Blockes liegt, in dem `zahl1` und `zahl2` vereinbart wurden – nämlich im Anweisungsblock der Methode `Main()`.

**Bitte beachten Sie:**

Die Variable `produkt` wäre auch in einem `else`-Zweig für die `if`-Anweisung unbekannt. Denn der `else`-Zweig gehört zwar zu der `if`-Anweisung, kennt aber die Variable aus dem Anweisungsblock von `if` nicht.

Einen ähnlich engen Bezugsrahmen haben wir auch schon einmal verwendet – und zwar bei einer `for`-Schleife, in der die Schleifenvariable direkt im Vorlauf vereinbart wurde. Diese Schleife sah so aus:

```
for(int i = 0; i <= 10; i++)
    Console.WriteLine("Die Variable hat jetzt den Wert {0}", i);
```

Hier gilt die Variable `i` nur innerhalb der Schleife. Außerhalb der Schleife ist die Variable unbekannt.

In den bisherigen Beispielen hat der Compiler sich mehr oder weniger deutlich über fehlerhafte Zugriffe beschwert. Es gibt aber auch Situationen, in denen mögliche Probleme durch lokale Variablen nicht sofort klar werden. Schauen Sie sich dazu einmal den folgenden Code an:

```
/* #####
Das sieht gut aus, klappt aber nicht
#####
using System;
namespace Cshp04d_01_06
```

```
{
    class Program
    {
        static void ErhoeheA(int a)
        {
            a = a + 1;
        }

        static void Main(string[] args)
        {
            int a = 10;
            Console.WriteLine("Die Variable hat den Wert {0}", a);
            ErhoeheA(a);
            Console.WriteLine("Die Variable hat den Wert {0}", a);
        }
    }
}
```

Code 1.6: Ein Problem mit lokalen Variablen**Bevor Sie weiterlesen ...**

Überlegen Sie einmal selbst, welche Werte die beiden Ausgabeanweisungen für die Variable `a` liefern.

Beim flüchtigen Hinsehen könnte man glauben, die Methode `ErhoeheA()` würde den Wert der Variablen `a` aus der Methode `Main()` um 1 erhöhen. Tatsächlich passiert aber genau das nicht. Denn an die Methode `ErhoeheA()` wird lediglich der Wert von `a` aus der Methode `Main()` übergeben und in eine eigene lokale Variable der Methode `ErhoeheA()` kopiert, die ebenfalls den Namen `a` hat.

Zur Auffrischung:

Eine Variable, die Sie in der Parameterliste einer Methode angeben, wird automatisch vereinbart. Sie müssen keine eigene Anweisung für die Vereinbarung benutzen.



Das heißt also, nur die lokale Variable `a` in der Methode `ErhoeheA()` hat den Wert 11. Der Wert der lokalen Variablen `a` in der Methode `Main()` dagegen ändert sich nicht. Er ist auch nach dem Aufruf der Methode `ErhoeheA()` nach wie vor 10.

Merken Sie sich:

Bei der Übergabe einer Variablen an eine Methode wird eine Kopie des Wertes angelegt und in einer lokalen Variablen der Methode gespeichert. Diese Übergabe wird auch **call by value^{a)}** genannt.



Sie können den übergebenen Wert in der Methode zwar ändern, diese Änderungen gelten aber ausschließlich für die lokale Kopie. Der eigentliche Wert – das Original – bleibt unverändert. Dabei spielt es auch überhaupt keine Rolle, ob Sie für den übergebenen Wert denselben Namen wie für das Original benutzen.

a) *Call by value* bedeutet übersetzt so viel wie „Aufruf mit Wert“.

So viel erst einmal zu den lokalen Variablen. Kommen wir nun zu den Klassenvariablen.

1.2 Klassenvariablen

Klassenvariablen sind das Gegenteil der lokalen Variablen. Sie werden außerhalb von Methoden in einer Klasse vereinbart.



Bitte beachten Sie:

Neben den Klassenvariablen gibt es in Klassen auch **Felder** oder **Instanzvariablen**. Sie gelten ebenfalls für die gesamte Klasse. Mit den Feldern werden wir uns bei der Objektorientierung noch sehr intensiv beschäftigen. In diesem Studienheft wollen wir uns ausschließlich den Umgang mit Klassenvariablen ansehen.

Die Vereinbarung von Klassenvariablen muss zwingend außerhalb **aller** Methoden erfolgen – also auch außerhalb der Methode `Main()`. Eine Variable, die Sie innerhalb der Methode `Main()` vereinbaren, ist **keine** Klassenvariable, sondern eine **lokale Variable** der Methode `Main()`.

Schauen wir uns eine Klassenvariable an einem Beispiel an. Dazu ändern wir Code 1.2 und ersetzen die lokalen Vereinbarungen der Variablen `varA` durch eine Klassenvariable. Dieser Klassenvariablen weisen wir dann in `Main()` und in der Methode `GibVarAus()` einen Wert zu, den wir anschließend an verschiedenen Stellen ausgeben lassen. Der geänderte Code sieht so aus:

```
/*
#####
Beispiel für eine Klassenvariable
#####
*/
using System;

namespace Cshp04d_01_07
{
    class Program
    {
        //die Vereinbarung von varA als Klassenvariable
        //die Vereinbarung muss außerhalb jeder Methode,
        //aber innerhalb der Klasse erfolgen
        //die Angabe static ist zwingend erforderlich
        static int varA;

        static void GibVarAus()
        {
            //die Klassenvariable varA erhält den Wert 4
            varA = 4;
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }

        static void Main(string[] args)
        {
            //die Klassenvariable varA erhält den Wert 5
            varA = 5;
```

```

        Console.WriteLine("Der Wert der Variablen ist {0}.",  

                           varA);  

        GibVarAus();  

        Console.WriteLine("Der Wert der Variablen ist {0}.",  

                           varA);  

    }  

}  

}

```

Code 1.7: Beispiel für eine Klassenvariable**Hinweise:**

Klassenvariablen müssen mit dem Schlüsselwort `static` vereinbart werden. Andernfalls ist in unseren Beispielen der Zugriff nicht möglich.

Anders als lokale Variablen werden Klassenvariablen automatisch mit dem Nullwert des jeweiligen Datentyps initialisiert – bei einem `int` also zum Beispiel mit 0.

Das Programm erzeugt folgende Ausgabe:

```

Der Wert der Variablen ist 5.  

Der Wert der Variablen ist 4.  

Der Wert der Variablen ist 4.

```

Sehen wir uns an, wie diese Ausgabe zustande kommt:

Der Wert 5 wird von der ersten Ausgabeanweisung in der Methode `Main()` ausgegeben. Da wir `varA` vorher den Wert 5 zugewiesen haben, ist die Ausgabe nicht weiter überraschend.

Die erste Ausgabe des Wertes 4 wird dann von der Ausgabeanweisung in der Methode `GibVarAus()` erzeugt. Auch hier ist die Ausgabe nicht überraschend, da wir ja in der ersten Anweisung von `GibVarAus()` der Variablen `varA` den Wert 4 zuweisen.

Interessant wird dann die letzte Ausgabe. Hier erscheint wieder der Wert 4, obwohl die Ausgabe aus `Main()` erfolgt. Das liegt daran, dass wir `varA` als Klassenvariable vereinbart haben. Damit ist der Wert, den wir `varA` in der Methode `GibVarAus()` zugewiesen haben, auch außerhalb der Methode bekannt. Anders als bei lokalen Variablen wird der Wert einer Klassenvariablen also beim Verlassen einer Methode nicht gelöscht.

Sie können lokale Variablen und Klassenvariablen auch gemeinsam verwenden. Dabei dürfen Sie auch denselben Namen verwenden – vorausgesetzt, Sie versuchen nicht, den Namen in ein und demselben Bezugsrahmen mehrfach zu verwenden.

Der folgende Code 1.8 zeigt Ihnen ein recht extremes Beispiel für den unterschiedlichen Einsatz der Variablen `varA`.

```

/* #####  

Gemeinsame Verwendung von lokalen Variablen  

und Klassenvariablen  

##### */  

using System;

```

```

namespace Cshp04d_01_08
{
    class Program
    {
        //die Vereinbarung von varA als Klassenvariable
        static int varA;

        static void AendereA()
        {
            //lokale Vereinbarung von varA
            int varA;
            varA = 0;
            if (varA == 0)
            {
                //hier wird eine noch lokalere Vereinbarung vom
                //Compiler verhindert
                //int varA = 42;
                //bitte in einer Zeile eingeben
                Console.WriteLine("Der Wert der Variablen ist {0}.",
                    varA);
            }
        }

        static void Main(string[] args)
        {
            varA = 5;
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
            AendereA();
            Console.WriteLine("Der Wert der Variablen ist {0}.",
                varA);
        }
    }
}

```

Code 1.8: Gemeinsame Verwendung von lokalen Variablen und Klassenvariablen

Die Variable `varA` wird im vorigen Code zweimal vereinbart. Damit Sie die entsprechenden Stellen im Code leichter wiederfinden, geben wir hier immer die komplette Zeile mit Kommentar an.

- 1) als Klassenvariable in der Zeile

```
static int varA;          //Vereinbarung von varA als
                        //Klassenvariable
```

- 2) als lokale Variable für die Methode `AendereA()`. Diese Vereinbarung erfolgt in der Zeile

```
int varA;                //lokale Vereinbarung von varA
```

Wenn Sie das Programm ausführen lassen, erscheinen folgende Ausgaben:

```
Der Wert der Variablen ist 5.
Der Wert der Variablen ist 0.
Der Wert der Variablen ist 5.
```

Schauen wir uns wieder an, wie diese Werte zustande kommen.

In der Methode `Main()` erhält `varA` in der ersten Anweisung den Wert 5 zugewiesen. Zu diesem Zeitpunkt ist nur die Klassenvariable `varA` bekannt. Die Zuweisung und auch die folgende Ausgabe beziehen sich daher auf diese Klassenvariable.

Danach wird die Methode `AendereA()` aufgerufen. Dort wird erneut eine Variable `varA` vereinbart. Diese Variable gilt aber nur lokal für die Methode `AendereA()`, da sie innerhalb des Anweisungsblocks der Methode vereinbart wird. Hier überdeckt die lokale Variable die Klassenvariable. Das heißt, die Zuweisung des Wertes 0 erfolgt an die lokale Variable `varA` der Methode `AendereA()`.

Lokale Variablen **überdecken** Klassenvariablen mit demselben Namen. Daher wird immer mit der lokaleren Variablen gearbeitet.



In der Methode `Main()` wird nach dem Verlassen der Methode `AendereA()` für `varA` wieder der Wert 5 ausgegeben. Das liegt daran, dass hier jetzt keine lokalen Variablen mehr mit dem Namen `varA` existieren. Es wird also wieder die Klassenvariable `varA` benutzt.

Wie Sie sehen, können Sie mit Klassenvariablen und lokalen Variablen sehr schnell Verwirrung stiften – vor allem, wenn Sie denselben Namen verwenden. Sie müssen den vorigen Code schon sehr konzentriert lesen, um die unterschiedlichen Ausgaben auf Anhieb nachvollziehen zu können.

Endlos können Sie ein und denselben Namen für immer lokale Variablen allerdings nicht benutzen. Wenn Sie im vorigen Code zum Beispiel die Kommentarzeichen vor der Anweisung

```
//int varA = 42;
```

in der `if`-Anweisung in der Methode `AendereA()` löschen, beschwert sich der Compiler, dass es bereits eine lokale Variable mit dem Namen `varA` gibt – obwohl diese Variable eigentlich nur lokal für den Anweisungsblock der `if`-Anweisung gelten würde. Sie können also innerhalb eines Anweisungsblocks den Namen einer lokalen Variablen aus der Methode nicht noch einmal für eine andere, lokale Variable benutzen.

1.3 Regeln für das Arbeiten mit lokalen Variablen und Klassenvariablen

Nachdem Sie nun lokale Variablen und Klassenvariablen kennengelernt haben, werden Sie sich sicher fragen, wann Sie welchen Typ einsetzen sollen.

Klassenvariablen haben scheinbar einen großen Vorteil: Sie können auf die Übergabe von Werten an Methoden verzichten und die erforderlichen Variablen einfach als Klassenvariablen vereinbaren. Diese Variablen können Sie ja auch in der jeweiligen Methode ansprechen und verändern.

Allerdings haben Klassenvariablen auch einen gravierenden Nachteil:

Ihre Programme werden sehr schnell unübersichtlich, wenn Sie den Wert einer Variablen an vielen Stellen im Quelltext verändern. Denken Sie nur an den vorigen Code. Hier wird erst beim sehr genauen Hinsehen klar, warum sich die Werte der Variablen ändern.

Besonders bei komplexen Programmen können Veränderungen an Klassenvariablen schnell zu sehr unangenehmen Fehlern führen, die nur mit viel Mühe zu finden sind. Sie verlieren mit Sicherheit irgendwann den Überblick, welche Klassenvariable Sie an welcher Stelle verändern, und wundern sich möglicherweise über seltsame Werte, für die Sie keine Erklärung haben.

Daher sollten Sie sich beim Arbeiten mit Variablen an die folgende Regel halten:



Vereinbaren Sie Variablen so lokal wie möglich.

Oder etwas drastischer:

Benutzen Sie Klassenvariablen nur dann, wenn es zwingend erforderlich ist.

Lassen Sie sich auch nicht in Versuchung führen, Klassenvariablen einzusetzen, um sich ein wenig Programmierarbeit zu sparen. Das folgende Beispiel rechnet über eine Methode das Quadrat einer Zahl aus. Es funktioniert zwar, ist aber kein guter Programmierstil.

```
/* ##### SO BITTE NICHT!
#####
SO BITTE NICHT!
##### */

using System;

namespace Cshp04d_01_09
{
    class Program
    {
        //zwei Klassenvariablen
        static int zahl, quadrat;

        static void Berechnung()
        {
            quadrat = zahl * zahl;
        }

        static void Main(string[] args)
        {
            Console.Write("Bitte geben Sie eine Zahl ein: ");
            //bitte jeweils in einer Zeile eingeben
            zahl = Convert.ToInt32(Console.ReadLine());
            Berechnung();
            Console.WriteLine("Das Quadrat der Zahl ist {0} ", quadrat);
        }
    }
}
```

Code 1.9: SO BITTE NICHT!

Die beiden Klassenvariablen `quadrat` und `zahl` lassen sich ohne jede Einschränkung der Funktionalität durch lokale Variablen ersetzen.

Bevor Sie weiterlesen ...

Versuchen Sie erst einmal selbst, das Programm so umzuschreiben, dass nur noch lokale Variablen benutzt werden.

Der geänderte Code könnte dann zum Beispiel so aussehen:

```
/* ##### Die Klassenvariablen werden durch lokale Variablen ersetzt #####
using System;
namespace Cshp04d_01_10
{
    class Program
    {
        //die Methode liefert einen Wert zurück und erhält
        //einen Wert
        static int Berechnung(int zahl)
        {
            //quadrat ist jetzt lokal
            int quadrat;
            quadrat = zahl * zahl;
            return quadrat;
        }

        static void Main(string[] args)
        {
            //zahl ist ebenfalls lokal
            int zahl;
            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());
            //die Methode wird mit zahl als Argument
            //aufgerufen und das Ergebnis direkt ausgegeben
            //bitte in einer Zeile eingeben
            Console.WriteLine("Das Quadrat der Zahl ist {0} ",
                Berechnung(zahl));
        }
    }
}
```

Code 1.10: Lokale Variablen statt Klassenvariablen

So viel zu lokalen Variablen und Klassenvariablen. Im nächsten Kapitel werden wir uns mit den Arrays beschäftigen.

Zusammenfassung

Variablen haben unterschiedliche Gültigkeitsbereiche – den Bezugsrahmen.

Lokale Variablen gelten nur in dem Anweisungsblock, in dem sie vereinbart wurden.
Klassenvariablen gelten in der Klasse, in der sie vereinbart wurden.

Wenn mehrere Variablen mit gleichen Namen und unterschiedlichen Gültigkeitsbereichen existieren, wird immer auf die lokale Variable zugegriffen. Lokale Variablen überdecken also Klassenvariablen mit demselben Namen.

Verzichten Sie nach Möglichkeit auf Klassenvariablen und verwenden Sie lokale Variablen – auch wenn dazu etwas mehr Aufwand erforderlich ist.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Wo muss eine Klassenvariable vereinbart werden?

- 1.2 Geben Sie für den folgenden Quelltextausschnitt an, bei welchen Variablen es sich um lokale Variablen handelt und welche Variablen Klassenvariablen sind. Geben Sie außerdem an, innerhalb welcher Methoden die Variablen gültig sind.

```
class Aufgabe {  
    static int e = 2;  
    static int Ausgabe(int zahl){  
        int ergebnis;  
        ergebnis = zahl * e;  
        return ergebnis;  
    }  
    static void Main(string[] args)  
    {  
        int eingabe;  
        ...  
    }  
}
```

e ist _____

gültig in _____

ergebnis ist _____

gültig in _____

zahl ist _____

gültig in _____

eingabe ist _____

gültig in _____

- 1.3 Sehen Sie sich den folgenden Quelltextausschnitt an. Welchen Wert hat die Variable `ausgabe` in der Methode `Main()` nach dem Aufruf der Methode `EinenAbziehen()`? Begründen Sie bitte Ihre Antwort.

```
...
static void EinenAbziehen(int ausgabe)
{
    ausgabe = ausgabe - 1;
}
static void Main(string[] args)
{
    int ausgabe = 100;
    EinenAbziehen(ausgabe);
...

```

2 Arrays

In diesem Kapitel lernen Sie, wie Sie mehrere Variablen desselben Typs in einem Array zu einer Einheit zusammenfassen.



Arrays^{a)} werden auch **Felder** genannt. Der Begriff Feld wird – wie Sie ja bereits aus dem letzten Kapitel wissen – in C# allerdings auch für spezielle Variablen innerhalb einer Klasse verwendet. Wir benutzen daher hier immer den Begriff Array.

- a) Wörtlich übersetzt bedeutet *array* so viel wie „Reihe“ oder „Aufstellung“.

Wenn wir noch einmal den Schrank mit den Variablen nehmen, entspricht ein Array in etwa einer Schublade, in der sich nicht mehr einzelne Kleidungsstücke befinden, sondern mehrere identische Schachteln. In jede dieser Schachteln können Sie dann genau eine Art von Kleidungsstück legen – zum Beispiel nur Hemden oder nur Hosen. Dabei dürfen zwei unterschiedliche Kleidungsstücke weder gemeinsam in einer Schachtel abgelegt werden, noch dürfen Sie die unterschiedlichen Kleidungsstücke in verschiedene Schachteln in einer Schublade legen.

Damit Sie die Schachteln eindeutig identifizieren können, erhält jede Schachtel eine eindeutige Nummer. Der Zugriff auf die Schachteln erfolgt dann durch eine Kombination aus dem Namen der Schublade und der Nummer der Schachtel – zum Beispiel Schublade „Hemden“ Schachtel 2.

Übertragen auf C# bedeutet das:

1. Ein Array fasst **mehrere Variablen desselben Typs** zusammen. Sie können in einem Array nicht verschiedene Typen mischen.
2. Die Identifizierung der einzelnen Variablen in einem Array erfolgt über eine laufende Nummer – den **Index**. Über den Index greifen Sie auf die Variablen in dem Array zu.



Die Variablen in einem Array werden **Arrayelement** oder kurz **Element** genannt. Jedes Element hat einen eindeutigen Index.

Die Vereinbarung eines Arrays erfolgt zunächst einmal fast genauso wie die Vereinbarung einer normalen Variablen. Sie müssen lediglich hinter dem Typ eckige Klammern angeben. Allgemein sieht die Vereinbarung eines Arrays so aus:

```
typ [] bezeichner;
```

Hier wird ein Array mit dem Namen `bezeichner` vereinbart, das Elemente des Typs `typ` verwalten kann. Für den Namen und den Typ gelten dabei die Regeln, die Sie bereits von normalen Variablen kennen.

Schauen wir uns die Vereinbarung jetzt in der Praxis an.

Um ein Array für den Typ `int` anzulegen, benutzen Sie die folgende Anweisung:

```
int [] zahlen;
```

Bitte beachten Sie:

Die eckigen Klammern bleiben bei der Vereinbarung eines Arrays grundsätzlich leer. Sie teilen dem Compiler lediglich mit, dass es sich bei der Vereinbarung um ein Array handelt und nicht um eine „normale“ Variable.



Die Vereinbarung gilt auch für nachfolgende Bezeichner, die in derselben Zeile stehen. Die Vereinbarung

```
int [] zahlen, nummern;
```

legt also zwei Arrays für den Typ `int` an und nicht – wie beim flüchtigen Hinsehen zu vermuten – ein Array und eine „normale“ `int`-Variable.

Um solche Verwirrungen zu vermeiden, sollten Sie sich angewöhnen, jedes Array in einer eigenen Zeile zu vereinbaren – also zum Beispiel so:

```
int [] zahlen;
int [] nummern;
```

Neben der eigentlichen Vereinbarung müssen Sie dem Compiler aber auch noch mitteilen, wie groß das Array sein soll – also wie viele Elemente das Array enthalten soll. Dazu gibt es zwei Möglichkeiten:

1. Sie zählen die einzelnen Elemente direkt bei der Vereinbarung auf.
2. Sie benutzen den Operator `new`¹.

Die Aufzählung könnte zum Beispiel so aussehen:

```
int [] zahlen = {1, 2, 3, 4, 5};
```

Hier wird ein Array mit fünf Elementen erzeugt. Die einzelnen Elemente erhalten dabei die Werte von 1 bis 5.

**Bitte beachten Sie:**

Sie müssen die Aufzählung direkt bei der Vereinbarung angeben. Die folgende Konstruktion wird vom Compiler nicht akzeptiert:

```
int [] zahlen;
zahlen = {1, 2, 3, 4, 5};
```

Bei größeren Arrays ist diese direkte Zuweisung allerdings recht umständlich, da Sie den Wert jedes einzelnen Elements angeben müssen. Hier können Sie den Operator `new` verwenden. Dieser Operator legt – ganz allgemein – ein neues Objekt im Speicher an und liefert einen Verweis auf das Objekt – eine **Referenz** – zurück. Um ein neues Array mit zehn Elementen zu erzeugen, könnten Sie zum Beispiel die folgende Anweisung verwenden:

```
int [] nummern = new int [10];
```

1. `New` bedeutet übersetzt „neu“.

Im linken Teil des Ausdrucks wird zunächst einmal ein Array mit dem Namen `nummern` für `int`-Typen vereinbart. Im rechten Teil wird dann über den Operator `new` ein neues Array vom Typ `int` erzeugt. Die Anzahl der Elemente muss dabei in eckigen Klammern hinter dem Typ angegeben werden.

Etwas übersichtlicher wird diese Vereinbarung, wenn Sie sie auf zwei Zeilen verteilen.

```
int [] nummern;
nummern = new int [10];
```

In der ersten Zeile wird das Array vereinbart. In der zweiten Zeile wird dann ein neues Array vom Typ `int` mit 10 Elementen erzeugt und die Referenz dem Array `nummern` zugewiesen.

Hinweis:

Die Vereinbarung in zwei Zeilen ist zwar gerade zu Beginn etwas übersichtlicher, hat aber durchaus ihre Tücken. Denn das eigentliche Erzeugen des Arrays über den Operator `new` wird schnell vergessen. Und dann gibt es nur den Bezeichner für das Array, aber nicht das Array selbst. Allerdings fängt der Compiler solche Fehler ab und weist Sie darauf hin, dass eine nicht zugewiesene Variable verwendet wird; beziehungsweise beim Zugriff wird eine Ausnahme ausgelöst. Dahinter verbirgt sich in der Regel aber das fehlende Erzeugen des Arrays über den Operator `new`.

Neben einer Konstanten können Sie bei der Angabe der Größe auch eine Variable oder einen Ausdruck benutzen. Möglich wären also auch die folgenden Vereinbarungen für ein Array `nummern` vom Typ `int`:

```
int a = 10;
//Angabe der Größe über eine Variable
nummern = new int [a];
```

oder

```
//Angabe der Größe über einen Ausdruck
nummern = new int [a + 20];
```

Hinweise:

Wenn Sie die Vereinbarung und das Anlegen auf zwei Zeilen verteilen, geben Sie bei der Zuweisung auf der linken Seite nur den Namen des Arrays an. Die eckigen Klammern dürfen Sie dann auf der linken Seite **nicht** verwenden.

Beim Erzeugen eines Arrays über den Operator `new` erhalten die Elemente den Nullwert des jeweiligen Datentyps – bei einem `int`-Array also den Wert 0.

Mit dem Operator `new` werden wir uns im weiteren Verlauf bei der Objektorientierung noch ausführlicher beschäftigen.

2.1 Zugriff auf die Elemente eines Arrays

Um auf ein Element in einem Array zuzugreifen, geben Sie erst den Bezeichner des Arrays und direkt dahinter den Index in eckigen Klammern an.

Bitte beachten Sie unbedingt:

Das erste Element in einem Array hat immer den Index 0. Damit hat das letzte Element immer den Index Anzahl – 1. Anzahl steht dabei für den Wert, den Sie bei der Vereinbarung angegeben haben.



Bei der Vereinbarung wird damit also quasi „normal“ von 1 an gezählt. Beim Zugriff dagegen beginnt die Zählung bei 0. Das kann gerade zu Beginn für viel Verwirrung sorgen.

Merken Sie sich deshalb:

Bei der Vereinbarung geben Sie die echte Anzahl der Elemente an. Beim Zugriff dagegen hat das erste Element den Index 0 und das letzte Element den Index Anzahl – 1.

Mit der Anweisung

```
zahlen[0] = 2;
```

weisen Sie zum Beispiel dem **ersten** Element im Array `zahlen` den Wert 2 zu.

Wenn Sie dem **letzten** Element in einem Array `zahlen` mit zehn Elementen den Wert 5 zuweisen wollen, benutzen Sie folgende Anweisung:

```
zahlen[9] = 5;
```

Noch einmal, weil es so wichtig ist:



Für ein Array `zahlen` mit zehn Elementen gibt es **kein** Element mit dem Index 10. Der Ausdruck `zahlen[10]` ist damit nicht gültig. Das letzte gültige Element in dem Array `zahlen` hat den Index 9. Auf dieses Element greifen Sie mit dem Ausdruck `zahlen[9]` zu.

Vor allem bei der Verarbeitung von Arrays in Schleifen ist diese Besonderheit sehr wichtig.

Schauen wir uns nun den Einsatz von Arrays an einem vollständigen Programm an. Nehmen wir einmal an, Sie wollen mit einem Programm hintereinander fünf Zahlen einlesen. Nach dem Einlesen soll für die Zahlen das Quadrat berechnet und ausgegeben werden.

Ohne Arrays müssten Sie für diese Aufgabe fünf Variablen vereinbaren und getrennt verarbeiten. Mit einem Array dagegen lässt sich diese Aufgabe sehr elegant durch eine Schleife lösen.

```

/*
#####
Beispiel für ein Array
#####
*/

using System;

namespace Cshp04d_02_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Vereinbarung des Arrays
            int[] zahlen = new int[5];

            //das Einlesen in das Array
            for (int index = 0; index < 5; index++)
            {
                //bitte in einer Zeile eingeben
                Console.Write("Bitte geben Sie die {0}. Zahl ein: ",
                index + 1);
                zahlen[index] = Convert.ToInt32(Console.ReadLine());
            }

            //die Berechnung und die Ausgabe
            for (int index = 0; index < 5; index++)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Das Quadrat von {0} ist: {1}",
                zahlen[index], zahlen[index] * zahlen[index]);
        }
    }
}

```

Code 2.1: Beispiel für ein Array

Schauen wir uns die einzelnen Zeilen des Codes der Reihe nach an.

Zuerst vereinbaren wir ein Array `zahlen` mit fünf `int`-Variablen.

Danach werden in der `for`-Schleife die fünf Werte eingelesen. Beachten Sie hier unbedingt, dass die Schleife von 0 bis 4 läuft und nicht von 1 bis 5!



Denken Sie daran:

Der Index für ein Array beginnt immer bei 0 und endet bei Anzahl – 1; in unserem Beispiel also bei 4. Ein Element `zahlen[5]` gibt es in dem Programm nicht!

Beim Einlesen in der ersten `for`-Schleife verwenden wir im Text für die Ausgabe den Ausdruck `index + 1`. Andernfalls würde der Text für die erste Eingabe ja lauten

Geben Sie die 0. Zahl ein

und auch bei allen weiteren Ausgaben würde die Angabe der Zahl eine Stelle hinterherlaufen.

Mit der zweiten `for`-Schleife werden dann die Zahlen und das jeweilige Quadrat ausgegeben. Auch diese Schleife muss von 0 bis 4 laufen!

Bis jetzt haben wir Elemente in einem Array entweder eingelesen oder ausgegeben. Die Elemente lassen sich aber im Prinzip genauso verwenden wie eine „normale“ Variable auch. Sie können also

- ein Element in Vergleichen verwenden,
- Rechenoperationen mit Elementen durchführen oder
- Elemente an eine Methode übergeben.

So sind zum Beispiel alle folgenden Operationen für ein `int`-Array `testArray` mit zehn Elementen erlaubt:

```
//Einsatz in einem Vergleich
if (testArray[i] == 5)

//Zuweisung an das Element mit Index 1
testArray[1] = 11;

//Dem Element mit dem Index 2 wird der Wert des Elements mit
//dem Index 4 zugewiesen
testArray[2] = testArray[4];

//Dem Element mit dem Index 3 wird das Ergebnis einer
//Rechenoperation zugewiesen
testArray[3] = 10 * 20;

//Das Element mit dem Index 5 wird als Argument an eine
//Methode übergeben
Methode(testArray[5]);
```

Möglich sind auch etwas abenteuerliche Konstruktionen wie zum Beispiel die Angabe des Index über ein anderes Element des Arrays:

```
//Der Index stammt aus dem Element mit dem Index 3
testArray[testArray[3]] = 4;
```

Egal wie sinnvoll die obigen Beispiele sind, erlaubt sind sie alle.

Auf eine Sache müssen Sie aber beim Arbeiten mit Arrays immer sehr sorgfältig achten:

Verlassen Sie beim Zugriff auf ein Element nie die Grenzen des Arrays – weder nach oben noch nach unten! Solche ungültigen Zugriffe führen zum sofortigen Ende des Programms.



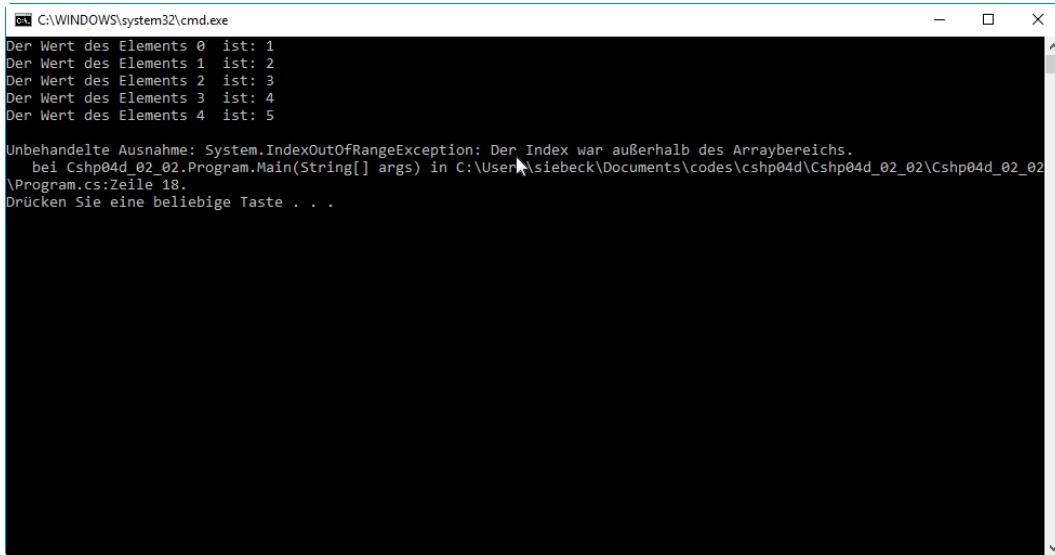
Sehen wir uns dazu das folgende Beispielprogramm an:

```
/* ##### VORSICHT! Die Arraygrenzen werden überschritten ##### */  
using System;  
  
namespace Cshp04d_02_02  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            //die Vereinbarung des Arrays mit Initialisierung  
            //der Werte  
            int[] zahlen = {1,2,3,4,5};  
  
            //die Ausgabe mit einer Überschreitung der  
            //Arraygrenzen  
            for(int index = 0; index < 7; index++)  
                //bitte in einer Zeile eingeben  
                Console.WriteLine("Der Wert des Elements {0} ist:  
                {1}", index, zahlen[index]);  
        }  
    }  
}
```

Code 2.2: Beispiel für die Überschreitung der Arraygrenzen

Vereinbart wird ein Array `zahlen` mit fünf Elementen. In der `for`-Schleife wird dann aber auf sieben Elemente zugegriffen – und zwar auf die Elemente mit den Indizes von 0 bis 6. Es werden also Elemente angesprochen, die überhaupt nicht vorhanden sind.

Wenn Sie das Programm jetzt eingeben, werden Sie eine Überraschung erleben. Es wird ohne Murren vom Compiler akzeptiert und auch ausgeführt. Beim Zugriff auf das erste ungültige Element wird allerdings eine Ausnahme ausgelöst und das Programm abgebrochen.



The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'C:\WINDOWS\system32\cmd.exe'. The window displays the following text:
Der Wert des Elements 0 ist: 1
Der Wert des Elements 1 ist: 2
Der Wert des Elements 2 ist: 3
Der Wert des Elements 3 ist: 4
Der Wert des Elements 4 ist: 5

Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war außerhalb des Arraybereichs.
bei Cshp04d_02.Program.Main(String[] args) in C:\User\siebeck\Documents\codes\cshp04d\Cshp04d_02_02\Cshp04d_02_02\Program.cs:Zeile 18.
Drücken Sie eine beliebige Taste . . .

Abb. 2.1: Der Programmabbruch durch einen Zugriff außerhalb des gültigen Bereichs

Merken Sie sich daher unbedingt:

Der Index des letzten Elements in einem Array ist immer um 1 kleiner als die Anzahl der Elemente, die Sie bei der Vereinbarung angegeben haben. Achten Sie bei der Verarbeitung von Arrays in Schleifen sehr sorgfältig darauf, dass Sie mit dem Element mit dem Index 0 beginnen und die Verarbeitung beim Element mit dem Index Anzahl – 1 beenden.



Um solche Probleme beim Zugriff auf Elemente eines Arrays zu vermeiden, können Sie die Länge eines Arrays über die Eigenschaft `Length`² ermitteln.

Die folgende Schleife durchläuft zum Beispiel alle Elemente in einem Array `zahlen`.

```
for (int index = 0; index < zahlen.Length; index++)
```

Bitte beachten Sie:



Die Eigenschaft `Length` liefert Ihnen die Anzahl der Elemente in einem Array. Da der Index aber immer um 1 kleiner ist als die Anzahl der Elemente, müssen Sie in der Schleife auf kleiner als `Length` überprüfen.

Denken Sie außerdem daran, dass die Schleife bei 0 beginnen muss und nicht bei 1. Andernfalls „vergessen“ Sie das erste Element im Array.

Zusätzlich kennt C# aber auch noch eine spezielle Schleife für die Verarbeitung von Arrays – nämlich `foreach`³. Mit dieser Schleife können Sie der Reihe nach auf jedes Element in einem Array zugreifen – ohne sich weiter um den Index kümmern zu müssen.

2. `Length` bedeutet übersetzt „Länge“.

3. Übersetzt bedeutet *for each* so viel wie „für jedes“.

Eingeleitet wird eine foreach-Schleife mit dem Schlüsselwort `foreach`. Bitte beachten Sie, dass es sich um ein Schlüsselwort handelt und nicht um zwei. Sie dürfen die Wörter also nicht auseinanderschreiben. Danach folgen in runden Klammern die Vereinbarung einer Variablen, das Schlüsselwort `in` und anschließend der Bezeichner des Arrays, das verarbeitet werden soll. Für ein Array `zahlen` könnte eine solche Schleife zum Beispiel so aussehen:

```
foreach (int element in zahlen)
    Console.WriteLine("Das Element hat den Wert {0}", element);
```

Die Variable `element` erhält bei jedem Schleifendurchlauf den Wert des aktuellen Elements im Array. Entsprechend werden über die Ausgabeanweisung dann die Werte aller Elemente hintereinander auf dem Bildschirm ausgegeben.

Bitte beachten Sie, dass ein direkter schreibender Zugriff auf die Werte in einem Array mit der `foreach`-Schleife nicht möglich ist. Das folgende Fragment sieht zwar auf den ersten Blick gut aus, wird vom Compiler aber nicht akzeptiert.

```
...
int[] nummern = new int[10];
foreach(int element in nummern)
    element = 1;
...
```



Für die sichere Ausgabe von Arrays verwenden Sie die `foreach`-Schleife.

Für den sicheren schreibenden Zugriff erstellen Sie eine Schleife mit der Eigenschaft `Length` des Arrays.

Die beiden Techniken für den sicheren Zugriff auf die Elemente eines Arrays finden Sie auch noch einmal zusammengefasst im folgenden Code.

```
/* #####
Sicherer Zugriff auf die Elemente
eines Arrays
#####*/
using System;

namespace Cshp04d_02_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] zahlen = new int[10];
            //Zuweisung über eine Schleife und die
            //Eigenschaft Length
            for (int element = 0; element < zahlen.Length; element++)
                zahlen[element] = element * element;
```

```
//Ausgabe über eine foreach-Schleife
foreach (int element in zahlen)
    Console.WriteLine("Der Wert ist {0}.", element);
}
```

Code 2.3: Sicherer Zugriff auf die Elemente eines Arrays

2.2 Mehrdimensionale Arrays

Bisher hatten unsere Arrays immer nur eine Dimension. Sie können aber auch Arrays mit mehreren Dimensionen anlegen – zum Beispiel eine Art Tabelle mit Spalten und Zeilen. Dazu geben Sie bei der Vereinbarung für jede weitere gewünschte Dimension ein Komma an und legen dann auch für die weiteren Dimensionen die Anzahl der Elemente getrennt durch Kommas fest.

Ein Array für eine Tabelle mit vier Spalten und acht Zeilen vereinbaren Sie zum Beispiel so:

```
int[,] tabellenArray = new int[4,8];
```

Das Array `tabellenArray` umfasst vier Elemente, die jeweils selbst wieder aus acht Elementen bestehen. Grafisch lässt sich dieses Array so darstellen:

Abb. 2.2: Ein mehrdimensionales Array als Tabelle

Hinweis:

Wie Sie die Werte in einem mehrdimensionalen Array interpretieren, ist relativ beliebig. Sie können die erste Dimension auch als Zeile betrachten und die zweite als Spalte. Die interne Organisation ändert sich dadurch nicht.

Der Zugriff auf die Elemente in einem mehrdimensionalen Array erfolgt fast genauso wie bei einem eindimensionalen Array. Sie müssen lediglich alle Dimensionen in der richtigen Reihenfolge angeben.

Die Anweisung

```
tabellenArray[2,3] = 5;
```

setzt zum Beispiel den Wert in der dritten Spalte der vierten Zeile von `tabellenArray` auf 5.


Denken Sie bitte daran:

Der Index für die Elemente beginnt bei 0. Für die dritte Spalte müssen Sie daher den Wert 2 angeben und für die vierte Zeile den Wert 3. Um den Wert in der zweiten Spalte und der dritten Zeile zu setzen, müssen Sie den Ausdruck `tabellenArray[1, 2]` verwenden.

Nach dieser Zuweisung würde unsere Beispieltabelle so aussehen:

		5	

Abb. 2.3: Die Tabelle nach der Wertzuweisung auf das Element `tabellenArray[2, 3]`


Bitte beachten Sie:

Bei mehrdimensionalen Arrays müssen Sie beim Zugriff auf die Elemente immer alle Dimensionen angeben. Eine direkte Zuweisung einer Zahl auf das Element `tabellenArray[1]` ist für unser Beispiel nicht möglich. Hier fehlt die Angabe der zweiten Dimension.

Genau wie eindimensionale Arrays können Sie auch mehrdimensionale Arrays direkt bei der Vereinbarung mit Werten belegen. Dabei geben Sie die Werte in der Reihenfolge der Dimensionen an. Die einzelnen Dimensionen werden dabei ebenfalls durch geschweifte Klammern umfasst und durch Kommas getrennt.

Die Initialisierung eines zweidimensionalen Arrays `tabellenArray` mit zwei Spalten und vier Zeilen sieht dann so aus:

```
int[,] tabellenArray = { {3, 5, 7, 6}, {2, 1, 0, 9} };
```

Zuerst werden die Werte der ersten Spalte angegeben, dann die Werte der zweiten Spalte.

Mehrdimensionale Arrays lassen sich auch sehr komfortabel mit einer `foreach`-Schleife verarbeiten. Dabei werden die einzelnen Dimensionen der Reihe nach verarbeitet – wie zum Beispiel im folgenden Code.

```
/*
#####
# Mehrdimensionale Arrays
#####
*/
using System;
namespace Cshp04d_02_04
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            //ein Array mit zwei Dimensionen und direkter Zuweisung
            //bitte in einer Zeile eingeben
            int[,] tabellenArray1 = { { 3, 5, 7, 6 },
                                    { 2, 1, 0, 9 } };

            //noch ein Array mit 2 Dimensionen
            int[,] tabellenArray2;
            tabellenArray2 = new int[3, 3];
            //die Daten im ersten Array ausgeben
            foreach (int element in tabellenArray1)
                Console.WriteLine("Der Wert ist {0}", element);

            Console.WriteLine();

            //einem Element im zweiten Array einen Wert zuweisen
            tabellenArray2[1, 1] = 10;
            //die Daten im zweiten Array ausgeben
            foreach (int element in tabellenArray2)
                Console.WriteLine("Der Wert ist {0}", element);
        }
    }
}

```

Code 2.4: Ein mehrdimensionales Array

Arrays können nahezu beliebig viele Dimensionen haben. Die folgende Anweisung verleiht zum Beispiel ein dreidimensionales Array `raumArray` mit je 30 Elementen in jeder Dimension:

```
int[, ,] raumArray = new int[30, 30, 30];
```

Beachten Sie aber, dass große mehrdimensionale Arrays unter Umständen sehr viel Speicherplatz benötigen. Das Array `raumArray` kann zum Beispiel 27 000 einzelne Werte aufnehmen und beansprucht damit bereits etwas mehr als 105 Kilobyte.

Hinweis:

Den Speicherbedarf eines Arrays können Sie recht einfach ermitteln: Multiplizieren Sie alle Dimensionen miteinander und multiplizieren Sie das Ergebnis mit dem Speicherbedarf des Typs. Für unser Beispiel lautet die Rechnung dann:
 $30 * 30 * 30 * 4 = 108\,000$. Die 4 steht für den Speicherbedarf des Typs `int` in Bytes. 108 000 geteilt durch 1 024 (1 024 Bytes ergeben ein Kilobyte) ergibt dann ganz genau 105,46875.

Die Initialisierung eines dreidimensionalen Arrays könnte zum Beispiel so aussehen:

```
int[, ,] raumArray = { {{1,4,6,7},{2,4,0,9},{-4,3,6,7}},
                      {{5,7,8,-2},{-3,22,6,4},{0,-3,2,4}} };
```

Wenn Sie das dreidimensionale Array jetzt ein wenig verwirrend finden: In der Praxis werden vor allem ein- und zweidimensionale Arrays eingesetzt. Arrays mit mehr Dimensionen finden Sie eher selten.

Hinweis:

Neben mehrdimensionalen Arrays, die wie eine Tabelle aufgebaut sind, können Sie mit C# auch mehrdimensionale Arrays erstellen, bei denen die Elemente in den weiteren Dimensionen ebenfalls wieder aus Arrays bestehen – also Arrays von Arrays. Damit wollen wir uns hier aber nicht weiter beschäftigen.

Schauen wir uns nun an, wie Sie Arrays an eine Methode übergeben.

2.3 Übergabe von Arrays an Methoden

Wie Sie ja bereits wissen, wird beim Erzeugen eines Arrays über den Operator `new` eine Referenz für das Array zurückgeliefert. Und mit dieser Referenz können Sie das gesamte Array an eine Methode übergeben. Dazu geben Sie einfach nur den Bezeichner des Arrays ohne die eckigen Klammern als Argument beim Aufruf der Methode an.



Der Bezeichner eines Arrays ohne eckige Klammern liefert die Referenz des Arrays.

Nehmen wir als Beispiel noch einmal den Code mit der Eingabe von Zahlen und der Berechnung des Quadrats. Wir werden es jetzt so ändern, dass die Berechnung und die Ausgabe der Rechenergebnisse in eigenen Methoden erfolgen. Die Methode `Quadrat()` soll die Berechnung übernehmen und die Methode `Eingabe()` soll die Daten einlesen.

```
/* ##### Ein Array als Argument für eine Methode #####
using System;

namespace Cshp04d_02_05
{
    class Program
    {
        static void Eingabe(int[] argArray)
        {
            //das Einlesen in das Array
            for (int index = 0; index < argArray.Length; index++)
            {
                //bitte jeweils in einer Zeile eingeben
                Console.Write("Bitte geben Sie die {0}. Zahl ein: ",
                index + 1);
                argArray[index] = Convert.ToInt32 (Console.
                ReadLine());
            }
        }

        static void Quadrat(int[] argArray)
```

```
{  
    foreach (int element in argArray)  
        //bitte in einer Zeile eingeben  
        Console.WriteLine("Das Quadrat von {0} ist: {1}",  
            element, element * element);  
}  
  
static void Main(string[] args)  
{  
    //die Vereinbarung des Arrays  
    int[] zahlenArray = new int[5];  
  
    //das Einlesen in einer Methode  
    Eingabe(zahlenArray);  
  
    //das Berechnen und die Ausgabe über eine Methode  
    Quadrat(zahlenArray);  
}  
}  
}
```

Code 2.5: Ein Array als Argument für eine Methode

In dem Code wird zunächst in der Methode `Main()` ein Array `zahlenArray` mit fünf Elementen vereinbart. Dieses Array wird komplett an die Methoden `Eingabe()` und `Quadrat()` übergeben und dort weiterverarbeitet. Die Übergabe erfolgt durch die Angabe des Arraybezeichners als Argument.

Bitte beachten Sie:

In den Parameterlisten der Methoden müssen Sie wieder ein Array angeben – also mit den eckigen Klammern. Die folgende Parameterliste führt zu einem Fehler, da ja nur ein einfacher `int`-Typ als Parameter vereinbart wird.

```
static void Eingabe(int argArray)
```



Wenn Sie den vorigen Code sorgfältig gelesen haben, ist Ihnen sicherlich eine Besonderheit aufgefallen: Die Methode `Eingabe()` liefert keinen Wert zurück. Trotzdem werden die eingelesenen Werte korrekt in der Methode `Main()` beziehungsweise in der Methode `Quadrat()` weiterverarbeitet. Eigentlich ist das nicht möglich, da ja die lokalen Werte einer Methode beim Verlassen der Methode verloren gehen. Eine Klassenvariable gibt es in dem Code auch nicht. Wie kann es also sein, dass `zahlenArray` die Werte enthält, die lokal in der Methode `Eingabe()` in die Elemente des Arrays `argArray` eingelesen werden?

Für die Erklärung müssen wir einen Blick hinter die Kulissen werfen:

Sie übergeben nicht die Werte des Arrays `zahlenArray` an die Methode, sondern die Referenz des Arrays `zahlenArray`. Diese Referenz wird dann auch vom Array `argArray` in der Methode benutzt. Damit haben `zahlenArray` und `argArray` also **dieselbe Referenz** und verweisen auf **dasselbe Array**. Jede Änderung an `argArray` führt so automatisch zu einer Änderung von `zahlenArray`.

Warum das so ist, werden wir uns gleich beim Kopieren von Arrays noch etwas genauer ansehen. An dieser Stelle sollten Sie sich lediglich merken:

Wenn Sie ein Array über die Referenz an eine Methode übergeben und in der Methode Änderungen an dem Array vornehmen, ändern Sie immer das Original. Es werden also keine Kopien erstellt, wie Sie das von der Übergabe normaler Variablen an Methoden kennen.



Bei der Übergabe einer Referenz übergeben Sie einen Verweis auf einen Speicherbereich. Dieser Übertragungsmechanismus wird daher auch **call by reference**^{a)} genannt. Die Übergabe eines Wertes dagegen ist – wie Sie ja bereits wissen – ein **call by value**.

- a) Übersetzt bedeutet *call by reference* so viel wie „Aufruf mit Referenz“.

Kommen wir nun zum Kopieren von Arrays.

2.4 Kopieren von Arrays

In der einfachsten Form „kopieren“ Sie ein Array durch eine direkte Zuweisung. Die Anweisung

```
aArray = bArray;
```

„kopiert“ das Array `bArray` in das Array `aArray` – vorausgesetzt, die beiden Arrays wurden korrekt vereinbart und über `new` erzeugt.

Da wir in dem Ausdruck jeweils die Referenzen der beiden Arrays benutzen, werden allerdings nicht tatsächlich die Werte kopiert, sondern lediglich die Referenz des Arrays `aArray` auf die Referenz des Arrays `bArray` gesetzt.



Noch einmal zur Auffrischung:

Die Referenz eines Arrays erhalten Sie durch den Bezeichner des Arrays ohne die eckigen Klammern. Die Referenz ist lediglich ein Verweis auf das Array und enthält nicht die Werte des Arrays.

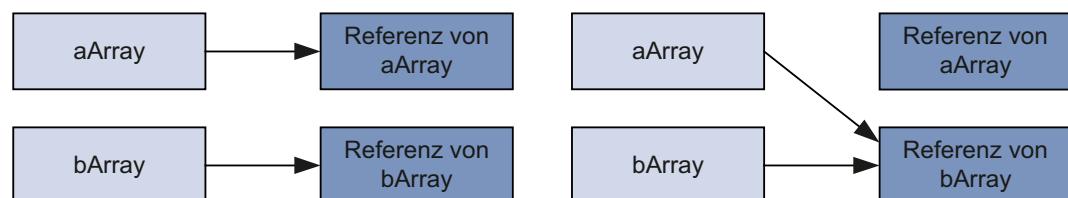


Abb. 2.4: Das Verändern der Referenz eines Arrays durch eine direkte Zuweisung
(links vor der Zuweisung, rechts nach der Zuweisung)

Da nun beide Array-Variablen mit derselben Referenz arbeiten – also auf ein und das-selbe Array verweisen –, führt jede Änderung über `aArray` auch zu einer entsprechen-den Änderung an `bArray` und umgekehrt. Das zeigt auch der folgende Code:

```
/* ##### Eine "unechte" Kopie eines Arrays ##### */

using System;

namespace Cshp04d_02_06
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] aArray;
            int[] bArray = new int[5];

            //bArray mit Werten füllen
            for (int element = 0; element < bArray.Length; element++)
                bArray[element] = 5 * element;

            //die Kopie erstellen
            aArray = bArray;
            //die Werte zur Kontrolle ausgeben
            Console.WriteLine("Die Werte in aArray sind:");
            foreach (int element in aArray)
                Console.WriteLine("{0}", element);
            Console.WriteLine("Die Werte in bArray sind:");
            foreach (int element in bArray)
                Console.WriteLine("{0}", element);

            //je einen Wert verändern
            aArray[0] = 200;
            bArray[4] = 1000;

            //die Änderung erfolgt in beiden Arrays
            Console.WriteLine("Die Werte in aArray sind:");
            foreach (int element in aArray)
                Console.WriteLine("{0}", element);

            Console.WriteLine("Die Werte in bArray sind:");
            foreach (int element in bArray)
                Console.WriteLine("{0}", element);
        }
    }
}
```

Code 2.6: Eine „unechte“ Kopie eines Arrays

Die Zeilen

```
aArray[0] = 200;
bArray[4] = 1000;
```

führen jetzt dazu, dass sich die Werte der entsprechenden Elemente sowohl in `aArray` als auch in `bArray` ändern – obwohl die Änderung eigentlich nur in jeweils einem Array erfolgt.

Ein ähnliches Verhalten haben Sie ja bereits bei der Übergabe von Arrays an Methoden kennengelernt. Auch hier erfolgt durch die Übergabe eine „unechte“ Kopie, da die Referenz identisch ist.

Hinweis:

Da durch die „unechte“ Kopie die Referenz von `aArray` verändert wird, ist im vorigen Code das eigentliche Erzeugen des Arrays `aArray` über den Operator `new` oder durch eine direkte Aufzählung nicht erforderlich. Die Referenz wird ja durch die Zuweisung gesetzt.

Wenn Sie tatsächlich den Inhalt eines Arrays in ein anderes kopieren wollen, können Sie jedes Element einzeln anfassen und es dann über eine Schleife dem passenden Element im Zielarray zuweisen. Für unser Beispiel könnte diese Schleife so aussehen:

```
for(int index = 0; index < bArray.Length; index++)
    aArray[index] = bArray[index];
```

Hier müssen Sie dann allerdings sorgfältig darauf achten, dass das Zielarray mindestens genauso groß ist wie das Array, aus dem Sie kopieren. Für das Beispiel von oben müssten Sie die Vereinbarung von `aArray` also so erweitern, dass auch tatsächlich Platz für fünf Elemente ist. Die Zeile müsste dann so aussehen:

```
int[] aArray = new int[5];
```

Da das Kopieren von größeren Arrays zu Fuß recht mühselig werden kann, bietet Ihnen C# für das Erstellen einer echten Kopie außerdem die Methode `Array.Clone()`⁴ an. Sie erzeugt eine Kopie eines Arrays mit demselben Inhalt und demselben Typ. Da die Methode `Array.Clone()` nur einen sehr allgemeinen Typ `object` liefert, müssen Sie das Ergebnis in der Regel noch in den passenden Typ umwandeln. Die Anweisung

```
bArray = (int[])cArray.Clone();
```

würde also eine Kopie des `int`-Arrays `cArray` erzeugen und die Referenz auf die Kopie im Array `bArray` ablegen.

Eindimensionale Arrays können Sie auch mit der Methode `CopyTo()`⁵ direkt in ein anderes Array kopieren. Als Argumente erwartet die Methode das Ziel der Kopie und den Index, ab dem die kopierten Werte im Ziel abgelegt werden sollen.

Mit der Methode `Copy()` können Sie auch nur einen bestimmten Bereich von Elementen aus einem Array in ein anderes kopieren.

4. *Clone* bedeutet übersetzt „klonen“. Gemeint ist damit eine absolut identische Kopie.

5. Übersetzt bedeutet *copy to* so viel wie „Kopiere nach“.

Beim Kopieren von Arrays müssen Sie darauf achten, dass das Zielarray groß genug ist. Andernfalls löst das Programm eine Ausnahme aus.



Zum Abschluss dieses Kapitels wollen wir uns – sozusagen als „Sahnehäubchen“ – noch ein paar Sonderfunktionen für Arrays ansehen.

2.5 Sonderfunktionen für Arrays

Arrays lassen sich sortieren und mit wenig Aufwand durchsuchen. Das Sortieren erfolgt dabei über die Methode `Array.Sort()`⁶ und für das Suchen können Sie zum Beispiel die Methode `Array.BinarySearch()`⁷ verwenden. Die Methode `Array.Sort()` erwartet als Argument das Array, das sortiert werden soll. Bei der Methode `Array.BinarySearch()` übergeben Sie zuerst das Array, das durchsucht werden soll, und dann den Wert, nach dem gesucht werden soll. Als Ergebnis liefert die Methode entweder einen Wert kleiner 0, wenn die Suche keinen Treffer ergab, oder die Position des Elements im Array.

Hinweis:

Die Methode `BinarySearch()` führt eine binäre Suche durch. Diese Suchvariante ist sehr schnell, setzt aber voraus, dass die Elemente sortiert sind. Vor dem Einsatz von `BinarySearch()` sollten Sie daher die Methode `Sort()` verwenden.

Im praktischen Einsatz finden Sie die verschiedenen Sonderfunktionen für Arrays im folgenden Code. In dem Programm werden zunächst einige Zahlen eingelesen und in einem Array abgelegt. Dieses Array wird dann in ein zweites Array kopiert. Danach wird das zweite Array sortiert und durchsucht.

```
/* ##### Sonderfunktionen für Arrays ##### */
using System;
namespace Cshp04d_02_07
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Vereinbarung des ersten Arrays mit fünf
            //Elementen
            int[] zahlenArray1 = new int[5];
            //ein zweites hat nur ein Element
            int[] zahlenArray2 = new int[1];

            //zwei int-Variablen für die Suche
            int suche, treffer;
```

6. *Sort* bedeutet übersetzt „Sortieren“.

7. *Binary search* bedeutet übersetzt „binäre Suche“.

```

//Werte in das erste Array einlesen
//bitte in einer Zeile eingeben
for (int element = 0; element < zahlenArray1.Length;
element++) {
{
    Console.Write("Bitte geben Sie einen Wert ein: ");
    //bitte in einer Zeile eingeben
    zahlenArray1[element] = Convert.ToInt32(Console.
    ReadLine());
}

//das erste Array wird komplett in das zweite kopiert
zahlenArray2 = (int[])zahlenArray1.Clone();

//jetzt wird das zweite Array sortiert und dann ausgegeben
Array.Sort(zahlenArray2);

Console.WriteLine("Die sortierte Ausgabe");
foreach (int element in zahlenArray2)
    Console.Write("{0} ",element);

Console.WriteLine();

//bitte in einer Zeile eingeben
Console.Write("Nach welcher Zahl soll gesucht
werden? ");
suche = Convert.ToInt32(Console.ReadLine());
//die Suche
treffer = Array.BinarySearch(zahlenArray2,suche);
//wurde etwas gefunden?
if (treffer < 0)
    Console.WriteLine("Die Suche ergab keinen Treffer.");
else
    //bitte in einer Zeile eingeben
    Console.WriteLine("Das Element {0} befindet sich an der
Position {1}.",suche, treffer+1);
//an zahlenArray1 hat sich nichts geändert
foreach (Int32 element in zahlenArray1)
    Console.Write("{0} ",element);

Console.WriteLine();
}
}
}

```

Code 2.7: Sonderfunktionen für Arrays

Mehr zu diesen Methoden und auch noch einige weitere Methoden für Arrays finden Sie in der Hilfe unter dem Stichwort **Array-Klasse**.

Im nächsten Kapitel werden wir uns mit dem Datentyp `string` für Zeichenketten beschäftigen.

Zusammenfassung

Ein Array kann für die Ablage von Werten mit demselben Datentyp benutzt werden.

Der Zugriff auf die Elemente in einem Array erfolgt über einen Index. Außerdem stellt Ihnen C# die `foreach`-Schleife zur Verfügung, die den Zugriff auf alle Elemente nacheinander ermöglicht.

Die Angaben zur Größe erfolgen entweder durch eine Aufzählung direkt bei der Vereinbarung oder über den Operator `new`.

Über die Eigenschaft `Length` können Sie die Anzahl der Elemente in einem Array ermitteln.

Der Zugriff auf ein ungültiges Element in einem Array führt zum Absturz des Programms.

Arrays können auch mehrere Dimensionen haben.

Um ein komplettes Array an eine Methode zu übergeben, geben Sie nur den Bezeichner des Arrays als Argument an. Dabei wird allerdings keine Kopie des Arrays in der Methode erstellt. Sie arbeiten auch in der Methode immer mit dem Original.

C# stellt Ihnen zahlreiche Sonderfunktionen für Arrays zur Verfügung. So können Sie zum Beispiel Arrays direkt in ein anderes Array kopieren, sortieren und auch durchsuchen.

Aufgaben zur Selbstüberprüfung

- 2.1 Sie haben ein Array `int [] testArray = new int [6]` vereinbart. Welchen Index hat das erste gültige Element? Welchen Index hat das letzte gültige Element?

- 2.2 Das folgende Quelltextfragment soll eine Methode `Eingabe()` aufrufen und dabei das komplette Array `testArray` vom Typ `float` als Argument übergeben. Die Anweisungen enthalten zwei grobe Fehler. In welchen Zeilen befinden sich die Fehler? Wie müssen die falschen Anweisungen richtig lauten?

```
static void Eingabe(int argArray[4])
{
    for(int i = 0; i < argArray.Length; i++)
    {
        ...
    }
}

static void Main(string[] args)
{
    float[] testArray = new float[5];
    Eingabe();
```

} ...

- 2.3 Sie haben ein dreidimensionales Array `r`, das wie folgt initialisiert ist:

```
int [, , ] r = { {{1,4,6,7},{2,4,0,9},{-4,3,6,7}},  
                  {{5,7,8,-2},{-3,22,6,4},{0,-3,2,4}} };
```

Geben Sie bitte für die folgenden Elemente jeweils den Wert an. Wenn der Wert nicht definiert ist, geben Sie „ungültig“ an.

a) `r[0,0,0]` _____

b) `r[0,2,3]` _____

c) `r[0,1,1]` _____

d) `r[1,0,3]` _____

e) `r[1,2,1]` _____

f) `r[1,3,4]` _____

- 2.4 Sie haben ein Array `test` mit zehn Elementen vom Typ `float` vereinbart. Erstellen Sie bitte Schleifen, die den Wert jedes einzelnen Elements der Reihe nach ausgeben. Geben Sie dabei die drei folgenden Varianten an:

a) über den Index mit einer festen Größe,

- b) über eine `foreach`-Schleife,

- c) über eine Schleife, bei der die Anzahl der Elemente über eine Eigenschaft des Arrays ermittelt wird.

3 Zeichenketten

Mit dem Datentyp `string` für Zeichenketten haben Sie bereits einige Male Kontakt gehabt. In diesem Kapitel werden wir uns den Datentyp einmal genauer ansehen und Ihnen auch einige Sonderfunktionen für die Arbeit mit Zeichenketten vorstellen.

Zuerst wollen wir aber noch einmal kurz zusammenfassen, was Sie bisher zum Datentyp `string` wissen:

- Die Zuweisung einer Zeichenkette erfolgt durch den Zuweisungsoperator `=`.
- Sie können mehrere Zeichenketten über den Operator `+` verbinden.
- Zeichenketten müssen immer mit den Anführungszeichen `" "` umfasst werden.

Schauen wir uns jetzt den Datentyp einmal etwas genauer an.

Beim Datentyp `Code` handelt es sich um eine Folge von Zeichen, die weitgehend automatisch verwaltet wird. Sie müssen eigentlich nichts weiter machen, als eine entsprechende Variable zu vereinbaren, und können dieser Variablen dann beliebige Zeichenketten zuweisen. Dabei ist über den Index auch der Zugriff auf einzelne Zeichen möglich – wie zum Beispiel im folgenden Code.

```
/*
#####
Zeichenweise Ausgabe von Zeichenketten
#####
*/
using System;

namespace Cshp04d_03_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung
            string zKette1;
            string zKette2;
            string zKette3;

            //die Zeichenketten erhalten Werte
            zKette1 = "Ich bin";
            zKette2 = " eine Zeichenkette.";

            //hier werden zwei Zeichenketten verbunden
            zKette3 = zKette1 + zKette2;

            //die Ausgabe der Länge bitte in einer Zeile eingeben
            Console.WriteLine("Die Zeichenkette enthält
{0} Zeichen.", zKette3.Length);

            //"normale Ausgabe" mit for und der Länge
            for (int index = 0; index < zKette3.Length; index++)
                Console.Write(" {0}", zKette3[index]);
            Console.WriteLine();
        }
    }
}
```

```
//Ausgabe über foreach
foreach (char zeichen in zKette3)
    Console.WriteLine("{0}", zeichen);

    Console.WriteLine();
}
}
```

Code 3.1: Zeichenweise Ausgabe einer Zeichenkette

In dem Code vereinbaren wir drei Variablen für Zeichenketten. Den ersten beiden Variablen weisen wir über den Zuweisungsoperator Werte zu. Bei der dritten Variablen verbinden wir die beiden ersten Zeichenketten über den Operator + und weisen dann das Ergebnis zu.

Anschließend geben wir die Länge der dritten Zeichenkette aus. Dazu benutzen wir – wie bei den Arrays – die Eigenschaft `Length`.

In den beiden Schleifen wird jedes einzelne Zeichen aus der Zeichenkette separat ausgegeben. Die erste Schleife arbeitet dabei mit einer normalen `for`-Konstruktion, die zweite Schleife dagegen mit einer `foreach`-Konstruktion.

Bitte beachten Sie:

Über den Index können Sie einzelne Zeichen aus einer Zeichenkette nur lesen. Ein schreibender Zugriff wie in der folgenden Anweisung klappt nicht.

```
zKette3[2] = 'A';
```

Denn der Wert einer Zeichenkette, die Sie über den Typ `string` speichern, kann an sich nicht verändert werden. Sie können lediglich der Variablen, die die Zeichenkette speichert, einen neuen Wert zuweisen.

Wenn Sie einzelne Zeichen in einer Zeichenkette verändern müssen, sollten Sie den Typ `StringBuilder` benutzen. Mehr zu diesem Typ erfahren Sie später.

**Tipp:**

Über den Operator + können Sie nicht nur zwei Zeichenketten miteinander verbinden. Auch die Verbindung einer Zahl und einer Zeichenkette ist möglich. C# nimmt dabei die erforderliche Konvertierung der Zahl in eine Zeichenkette automatisch vor.

3.1 Zeichenketten vergleichen

Zeichenketten lassen sich – genau wie zum Beispiel Zahlen – auch vergleichen. Dazu benutzen Sie wie gewohnt die Operatoren `==` beziehungsweise `!=`. Für den Vergleich mit einer leeren Zeichenkette können Sie auch die Eigenschaft `String.Empty`⁸ benutzen.

8. *Empty* bedeutet übersetzt „leer“.


Bitte beachten Sie:

Der C#-Datentyp heißt `string` mit kleinem `s`. Die .NET Framework-Klasse dagegen schreibt sich mit großem `S` zu Beginn – also `String`.

Mit der Methode `String.Compare()`⁹ können Sie außerdem genau feststellen, wie sich zwei Zeichenketten unterscheiden. Die Methode erwartet als Argumente zwei Zeichenketten und liefert Ihnen entweder den Wert 0 – bei Gleichheit – oder aber einen Wert ungleich 0. Handelt es sich bei dem zurückgelieferten Wert um eine positive Zahl, ist das erste abweichende Zeichen in der ersten Zeichenkette größer als das entsprechende Zeichen in der zweiten Zeichenkette. Ist die zurückgelieferte Zahl dagegen negativ, ist das erste abweichende Zeichen in der ersten Zeichenkette kleiner als das entsprechende Zeichen in der zweiten Zeichenkette. Der Vergleich erfolgt dabei jeweils über den numerischen Wert des Zeichens.

Im praktischen Einsatz finden Sie den Vergleich von Zeichenketten im folgenden Code. Hier wird zunächst so lange eine Zeichenkette eingelesen, bis die Eingabe nicht leer ist. Danach wird die eingegebene Zeichenkette mit einer vorgegebenen Zeichenkette verglichen.

```
/* ##### Zeichenketten vergleichen #####
using System;
namespace Cshp04d_03_02
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung
            string zKette2;
            string zKette1 = "Rätsel";
            //die zweite Zeichenkette einlesen
            do
            {
                Console.Write("Raten Sie. Wie heißt das Wort? ");
                zKette2 = Console.ReadLine();
            } while (zKette2 == String.Empty);
            //die beiden Zeichenketten vergleichen
            if (zKette1 == zKette2)
                Console.WriteLine("Sie haben richtig geraten.");
            else
                Console.WriteLine("Leider falsch.");
        }
    }
}
```

Code 3.2: Zeichenketten vergleichen

9. `Compare` bedeutet übersetzt „Vergleiche“.

Schauen wir uns nun noch einige Funktionen für die Veränderung von Zeichenketten an.

3.2 Zeichenketten verändern

Wie Sie bei der zeichenweisen Ausgabe einer Zeichenkette gelernt haben, kann eine Zeichenkette, die Sie über den Typ `string` verarbeiten, an sich nicht verändert werden. C# stellt Ihnen aber trotzdem zahlreiche Methoden für die Veränderung von Zeichenketten vom Typ `string` zur Verfügung. Dabei wird allerdings nie die eigentliche Zeichenkette bearbeitet, sondern immer eine neue, veränderte Zeichenkette zurückgeliefert.

Schauen wir uns das an einem Beispiel an:

Mit der Methode `Trim()`¹⁰ können Sie zum Beispiel Leerzeichen am Anfang und Ende einer Zeichenkette entfernen. Das folgende Codefragment wird zwar vom Compiler akzeptiert, liefert aber nicht das gewünschte Ergebnis.

```
Console.WriteLine(zKette1);
zKette1.Trim();
Console.WriteLine(zKette1);
```

Denn die Methode `Trim()` verändert nicht direkt die Zeichenkette `zKette1`, sondern liefert eine neue Zeichenkette mit den Änderungen zurück. Da wir diese neue Zeichenkette nicht zuweisen, gehen die Änderungen verloren. Die Ausgabe vor der Veränderung ist daher identisch mit der Ausgabe nach der Veränderung.

Erst wenn Sie das Ergebnis der Veränderung wieder der Variablen `zKette1` zuweisen, sind die Leerzeichen vorne und hinten verschwunden. Korrekt müsste die Anweisung aus dem Codefragment also so aussehen:

```
zKette1 = zKette1.Trim();
```

Mit der Methode `Trim()` können Sie auch gezielt einzelne Zeichen am Anfang oder Ende einer Zeichenkette löschen. Dazu geben Sie die Zeichen als Argument an und trennen sie durch ein Komma. Die folgende Anweisung löscht zum Beispiel die Buchstaben R und l am Anfang und am Ende einer Zeichenkette.

```
zKette1 = zKette1.Trim('R', 'l');
```

Bitte beachten Sie:

Einzelne Zeichen müssen durch Apostrophe umfasst werden. Die Anweisung

```
zKette1 = zKette1.Trim("R", "l");
```

führt zu einer Fehlermeldung, da das erste Argument durch die Anführungszeichen als Zeichenkette übergeben wird.



Weitere Methoden zur Veränderung von Zeichenketten mit einer kurzen Beschreibung finden Sie in der folgenden Tabelle. Bitte denken Sie bei allen Methoden daran, dass nicht die Zeichenkette selbst verändert wird, sondern eine neue Zeichenkette mit den Änderungen zurückgeliefert wird.

10. `Trim` bedeutet übersetzt so viel wie „beschneide, stutze“.

Tab. 3.1: Methoden zur Veränderung von Zeichenketten

Methode ^{a)}	Wirkung
Insert()	Die Methode <code>Insert()</code> fügt eine Zeichenkette in eine andere Zeichenkette ein. Als Argumente müssen Sie zuerst die Indexposition für das Einfügen übergeben und dann die Zeichenkette, die eingefügt werden soll. Bitte denken Sie daran, dass der Index bei 0 beginnt und nicht bei 1.
Remove()	Die Methode <code>Remove()</code> löscht Zeichen in einer Zeichenkette. Als Argumente müssen Sie zuerst die Startposition für das Löschen übergeben und dann die Anzahl der Zeichen, die gelöscht werden sollen. Bitte denken Sie daran, dass der Index bei 0 beginnt und nicht bei 1.
Replace()	Die Methode <code>Replace()</code> ersetzt alle Vorkommen einer Zeichenkette durch eine andere. Als Argumente müssen Sie zuerst die Zeichenkette angeben, die ersetzt werden soll, und dann die Zeichenkette, die stattdessen benutzt wird.
ToLower()	Die Methode <code>ToLower()</code> wandelt alle Großbuchstaben in einer Zeichenkette in Kleinbuchstaben um.
ToUpper()	Die Methode <code>ToUpper()</code> wandelt alle Kleinbuchstaben in einer Zeichenkette in Großbuchstaben um.

- a) Die Wirkung können Sie sich auch leicht an den deutschen Übersetzungen merken. `Insert` bedeutet so viel wie „füge ein“, `remove` „entferne“ und `replace` „ersetze“. `To lower` lässt sich mit „zu kleingeschrieben“ übersetzen und `to upper` mit „zu großgeschrieben“.

Im praktischen Einsatz finden Sie einige dieser Methoden im folgenden Code.

```
/*
#####
Zeichenketten verändern
#####
*/
using System;

namespace Cshp04d_03_03
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vereinbarung
            string zKette1 = " Rätsel ";

            //Leerzeichen vorne und hinten entfernen
            Console.WriteLine(zKette1);
            zKette1=zKette1.Trim();
            Console.WriteLine(zKette1);

            //gezielt einzelne Zeichen vorne und hinten entfernen
            zKette1=zKette1.Trim('R','l');
            Console.WriteLine(zKette1);
        }
    }
}
```

```
//und wieder einfügen  
zKette1=zKette1.Insert(0,"R");  
Console.WriteLine(zKette1);  
  
zKette1=zKette1.Insert(zKette1.Length,"l");  
Console.WriteLine(zKette1);  
  
//ersetzen geht auch  
zKette1=zKette1.Replace("ä","ae");  
Console.WriteLine(zKette1);  
  
//alles in Großbuchstaben  
zKette1=zKette1.ToUpper();  
Console.WriteLine(zKette1);  
  
//alles in Kleinbuchstaben  
zKette1=zKette1.ToLower();  
Console.WriteLine(zKette1);  
}  
}
```

Code 3.3: Einige Veränderungen an Zeichenketten

3.3 Zeichenketten durchsuchen

Kommen wir nun noch zum Durchsuchen von Zeichenketten. Dazu können Sie die Methoden `IndexOf()` beziehungsweise `LastIndexOf()`¹¹ verwenden. Beide Methoden erwarten als Argument das Zeichen beziehungsweise die Zeichenkette, nach der gesucht werden soll. Die Methode `IndexOf()` liefert den Index des ersten Vorkommens zurück und die Methode `LastIndexOf()` den Index des letzten Vorkommens. Wenn das Zeichen beziehungsweise die Zeichenkette nicht gefunden wurde, liefern die Methoden den Wert `-1` zurück.

Bitte beachten Sie:



Beide Methoden liefern nicht die Position in der Zeichenkette, sondern den Index in der Zeichenkette. Für die eigentliche Position müssen Sie auf das Ergebnis daher noch 1 addieren.

Zusätzlich können Sie an beide Methoden auch den Index übergeben, ab dem mit der Suche begonnen werden soll. Das ist zum Beispiel interessant, wenn Sie alle Vorkommen eines Zeichens in einer Zeichenkette zählen wollen. Dazu lassen Sie in einer Schleife jeweils immer ab dem letzten Treffer weitersuchen. Das könnte zum Beispiel so aussehen:

```
/* ##### Zeichenketten durchsuchen #####
using System;
```

¹¹ *Index of* lässt sich mit „Index von“ übersetzen und *last index of* mit „letzter Index von“.

```

namespace Cshp04d_03_04
{
    class Program
    {
        static void Main(string[] args)
        {
            string zKette;
            char zeichen;
            int index=0;
            int zaehler=0;

            Console.Write("Bitte geben Sie die Zeichenkette ein: ");
            zKette=Console.ReadLine();
            //bitte in einer Zeile eingeben
            Console.Write("Nach welchem Zeichen soll gesucht
werden? ");
            zeichen=Convert.ToChar(Console.Read());

            //die erste Suche
            index=zKette.IndexOf(zeichen);

            //wenn das Zeichen gefunden wurde, wird weitergesucht
            while (index>=0)
            {
                index++;
                zaehler++;
                index=zKette.IndexOf(zeichen, index);
            }

            //bitte in einer Zeile eingeben
            Console.WriteLine("\nIn der Zeichenkette {0} kommt {1} {2}
Mal vor.", zKette, zeichen, zaehler);
        }
    }
}

```

Code 3.4: Alle Vorkommen eines Zeichens in einer Zeichenkette zählen

Mit der Anweisung

```
index = zKette.IndexOf(zeichen);
```

überprüfen wir zunächst einmal, ob das gesuchte Zeichen überhaupt in der Zeichenkette vorkommt. Wenn das der Fall ist, werden in der Schleife die Variablen `index` und `zaehler` erhöht. Danach wird die Suche so lange fortgesetzt, bis `index` den Wert `-1` hat. Allerdings wird die Suche in der Schleife jeweils ab dem aktuellen Wert von `index` durchgeführt.

3.4 Die Klasse `StringBuilder`

Neben der Klasse `string`, die an der eigentlichen Zeichenkette nichts verändern kann, gibt es in C# auch noch die Klasse `System.Text.StringBuilder`¹². Zeichenketten aus dieser Klasse können direkt verändert werden, müssen also nicht erst wieder zuge-

12. *String Builder* lässt sich etwas holprig mit „Zeichenkettenerbauer“ übersetzen.

wiesen werden. Dadurch sind Operationen wie das häufige Aneinanderfügen von Zeichenketten über die Klasse `StringBuilder` oft schneller als entsprechende Operationen über die Klasse `string`, da intern weniger Aufwand erforderlich ist.

Allerdings ist die Klasse `StringBuilder` auch ein wenig sperriger als die Klasse `string`. So können Sie zum Beispiel Zeichenketten nicht direkt zuweisen, sondern müssen den Operator `new` benutzen und die gewünschte Zeichenkette übergeben.

Im praktischen Einsatz finden Sie die Klasse `StringBuilder` im folgenden Code.

```
/*
#####
Die Klasse StringBuilder
#####
*/

using System;
using System.Text;

namespace Cshp04d_03_05
{
    class Program
    {
        static void Main(string[] args)
        {
            //eine Zeichenkette mit dem Inhalt Rätsel erzeugen
            StringBuilder zKette = new StringBuilder("Rätsel");
            Console.WriteLine("{0}", zKette);
            //eine Zeichenkette anhängen
            zKette.Append("haft ist manches Mal das Programmieren.");
            Console.WriteLine("{0}", zKette);

            //Zeichen lassen sich auch direkt verändern
            zKette[0] = 'A';
            Console.WriteLine("{0}", zKette);
        }
    }
}
```

Code 3.5: Die Klasse `StringBuilder`

Hinweis:

Die Klasse `StringBuilder` befindet sich im Namensraum `System.Text`. Denken Sie bitte daher daran, den entsprechenden Namensraum im vorigen Code einzubinden. Sonst ist die Klasse nicht bekannt.

Über die Zeile

```
StringBuilder zKette = new StringBuilder("Rätsel");
```

erzeugen wir eine Zeichenkette vom Typ `StringBuilder` mit dem Inhalt „Rätsel“.

Danach hängen wir mit der Methode `Append()`¹³ eine Zeichenkette an. Da diese Änderungen direkt an der ursprünglichen Zeichenkette vorgenommen werden, ist eine erneute Zuweisung wie bei der Klasse `string` nicht erforderlich.

13. `Append` bedeutet übersetzt so viel wie „füge hinzu“.

Mit der Anweisung

```
zKette[0] = 'A';
```

schließlich ändern wir das erste Zeichen in der Zeichenkette. Dieser schreibende Zugriff auf einzelne Zeichen war ja bei der Klasse `string` nicht möglich.

Das .NET Framework kennt noch zahlreiche weitere Funktionen für Zeichenketten, die wir Ihnen hier aber nicht einzeln vorstellen wollen. So können Sie über die Methode `Substring()` zum Beispiel auch Teile einer Zeichenkette „herausoperieren“. Erklärungen zu diesen Methoden finden Sie in der Hilfe unter dem Stichwort **String-Klasse** beziehungsweise **StringBuilder-Klasse**. Bitte beachten Sie aber, dass es nicht zwangsläufig alle Eigenschaften und Methoden der Klasse `string` auch für die Klasse `StringBuilder` gibt. Auch Konstruktionen wie eine `foreach`-Schleife sind für die Klasse `StringBuilder` nicht möglich.

So viel zu den Zeichenketten.

Zusammenfassung

Der Datentyp `string` bildet eine Folge von Zeichen ab, die weitgehend automatisch vom Compiler verwaltet wird.

Über die Operatoren `==` und `!=` können Sie zwei Zeichenketten vergleichen. Über die Methode `String.Compare()` können Sie außerdem genau feststellen, wie sich zwei Zeichenketten unterscheiden.

C# stellt Ihnen zahlreiche Methoden zur Veränderung von Zeichenketten zur Verfügung. Diese Methoden verändern beim Typ `string` allerdings nicht das Original, sondern liefern eine veränderte Kopie zurück.

Mit den Methoden `IndexOf()` und `LastIndexOf()` können Sie Zeichenketten auch durchsuchen.

Mit der Klasse `StringBuilder` können Sie Änderungen direkt an einer Zeichenkette vornehmen.

Aufgaben zur Selbstüberprüfung

- 3.1 Ist die folgende Anweisung für einen Datentyp `string` möglich? Begründen Sie bitte Ihre Antwort.

```
zeichenkette[10] = 'x';
```

- 3.2 Welche Eigenschaft der Klasse `String` steht für eine leere Zeichenkette?

- 3.3 Welches Ergebnis liefert die Methode `String.Compare()`, wenn die beiden Zeichenketten identisch sind?

4 Strukturen, Tupel und Aufzählungstypen

*Im Kapitel über Arrays haben Sie gelernt, wie Sie mehrere Variablen desselben Datentyps zusammenfassen. In diesem Kapitel beschäftigen wir uns mit ähnlichen Konstruktionen, die auch Datenobjekte mit unterschiedlichen Datentypen enthalten können – den **Strukturen** und **Tupeln**. Außerdem lernen Sie Aufzählungstypen kennen.*



Strukturen, Tupel und Aufzählungstypen sind **benutzerdefinierte Datentypen**. Diese benutzerdefinierten Datentypen basieren auf den Standarddatentypen, erhalten aber einen eigenen Namen.

4.1 Strukturen

Mit Strukturen können Sie Daten zusammenfassen und diese Zusammenfassung unter einem eigenen Namen als eine Art eigenen Datentyp speichern. Damit können Sie dann auch mehrere Datenobjekte mit einer identischen Struktur erzeugen.

Hinweis:

Strukturen können nicht nur Daten speichern, sondern auch Methoden. Damit kommen Strukturen den Klassen sehr nahe. Da wir uns später noch sehr intensiv mit Klassen beschäftigen werden, konzentrieren wir uns hier nur auf die Speicherung von Daten in Strukturen. Um Methoden in einer Struktur werden wir uns nicht weiter kümmern.

4.1.1 Vereinbarung einer Struktur

Eingeleitet wird die Vereinbarung einer Struktur mit dem Schlüsselwort `struct`. Dann folgen ein Bezeichner für die Struktur und in geschweiften Klammern die Liste der **Mitglieder**. Diese Mitglieder – auch **Members** genannt – sind sozusagen die Elemente der Struktur.

Eine Struktur mit drei `int`-Mitgliedern könnte zum Beispiel so aussehen:

```
struct Kiste
{
    public int Breite;
    public int Hoehe;
    public int Laenge;
}
```

Vereinbart wird hier eine Struktur mit dem Bezeichner `Kiste`. Sie enthält die drei Mitglieder `Breite`, `Hoehe` und `Laenge`. Alle Mitglieder sind vom Typ `int`.

Bitte beachten Sie:

Damit Sie auf die Mitglieder einer Struktur von außen zugreifen können, müssen Sie vor den Typ jedes Mitglieds die Angabe `public` setzen. Sie ermöglicht einen öffentlichen Zugriff. Was sich genau dahinter verbirgt, erfahren Sie, wenn wir uns mit den Sichtbarkeiten beschäftigen.

Die Vereinbarung einer Struktur muss außerhalb einer Methode erfolgen – auch außerhalb der Methode `Main()`.

Der Datentyp, die Anzahl und der Name der Mitglieder in einer Struktur sind relativ beliebig. Sie müssen lediglich die üblichen Vorgaben einhalten – zum Beispiel beim Namen.

Die Mitglieder einer Struktur können – müssen aber nicht – unterschiedliche Datentypen haben. Möglich wäre also auch folgende Vereinbarung für die Struktur `Kiste`:

```
struct Kiste
{
    public int Breite;
    public float Hoehe;
    public double Laenge;
}
```

Mit der Vereinbarung der Struktur haben Sie aber zunächst einmal nur den Datentyp festgelegt – also dem Compiler bekannt gemacht, wie der benutzerdefinierte Typ heißen soll und wie er aufgebaut ist. Um nun die Struktur nutzen zu können, müssen Sie im zweiten Schritt noch eine Variable vom Typ der Struktur vereinbaren. Die Vereinbarung dieser **Strukturvariablen** erfolgt in der gewohnten Art und Weise: Zuerst geben Sie den Typ an – in unserem Beispiel `Kiste` – und anschließend den Bezeichner der Variablen.

Die Anweisung

```
Kiste kleineKiste;
```

vereinbart also eine Variable mit dem Namen `kleineKiste` vom Typ `Kiste`.

Bitte beachten Sie:

Für den Einsatz einer Struktur sind immer zwei Schritte erforderlich:

1. Sie vereinbaren die eigentliche Struktur mit dem Schlüsselwort `struct` und der Liste der Mitglieder.
2. Sie vereinbaren eine Variable für die Struktur.

Wie von anderen Datentypen gewohnt, können Sie auch für eine Struktur Arrays anlegen. Die Anweisung

```
Kiste[] lagerraum = new Kiste[80];
```

vereinbart ein Array `lagerraum`, das 80 Elemente vom Typ `Kiste` aufnehmen kann. Bei `Kiste` handelt es sich um die Struktur, die wir weiter oben vereinbart haben.

Mit Strukturen lassen sich viele Programmierprobleme recht elegant lösen. Nehmen wir dazu ein Beispiel mit Kisten und einem Lagerraum. Sie könnten die Daten der einzelnen Kisten natürlich auch in drei getrennten Arrays speichern – eins für die Breite, eins für die Höhe und ein weiteres für die Länge. Die jeweiligen Werte legen Sie dann immer an derselben Position in den Arrays ab. Die Breite der ersten Kiste speichern Sie in dem Element `breite[0]`, die Höhe entsprechend in dem Element `hoehe[0]` und so weiter. Zwar können Sie diese Arrays dann auch in einer Schleife einlesen oder an eine Methode übergeben. Dabei müssen Sie aber sehr sorgfältig darauf achten, dass Sie nicht versehentlich die Daten von zwei unterschiedlichen Kisten durcheinanderwerfen. Außerdem müssten Sie ja immer alle drei Arrays einzeln verarbeiten.

Sehr einfacher wird die Verarbeitung mit einer Struktur, die die Daten zusammenfasst. Statt für jedes Element einzeln ein Array anzulegen, erstellen Sie einfach ein Array für die Struktur – so wie in dem Beispiel von oben. Über dieses Array können Sie dann gezielt auf die Werte jedes einzelnen Eintrags zugreifen. Im Element `lagerraum[0]` würden sich – schön sauber zusammengefasst – die Breite, die Höhe und auch die Länge der ersten Kiste befinden. Diese Daten können Sie komfortabel in einer Schleife einlesen oder auch gemeinsam in einem Rutsch an eine Methode übergeben. Wie das genau geht, sehen wir uns gleich noch detailliert an.

Eine Struktur hat im Vergleich zu normalen Variablen also zwei Vorteile:

1. Sie können logisch zusammengehörige Daten unter einem Bezeichner zusammenfassen. Damit lassen sich zusammengehörige Informationen als eine Einheit betrachten.
2. Sie können Daten über die Strukturvariable zusammenhängend verarbeiten.

Schauen wir uns nun an, wie Sie auf die Mitglieder einer Struktur zugreifen.

4.1.2 Zugriff auf Mitglieder einer Struktur

Der Zugriff erfolgt über den Namen der Strukturvariablen und den Namen des Mitglieds. Die beiden Teile müssen dabei durch einen Punkt getrennt werden.

Der Ausdruck `kleineKiste.Breite` steht zum Beispiel für das Mitglied `Breite` der Strukturvariablen `kleineKiste`. Das Mitglied `Laenge` der Strukturvariablen `kleineKiste` sprechen Sie über `kleineKiste.Laenge` an.

Beim Zugriff sind vor allem zwei Sachen wichtig:

1. Sie müssen den **Namen der Strukturvariablen** angeben und nicht den Namen der Struktur selbst. Nehmen wir noch einmal das Beispiel der Kiste von oben. Die Struktur haben wir mit dem Namen `Kiste` vereinbart, die Strukturvariable heißt `kleineKiste`. Zur Erinnerung noch einmal der Quelltext:

```
struct Kiste
{
    public int Breite;
    public int Hoehe;
    public int Laenge;
}
Kiste kleineKiste;
```

Der Bezeichner `Kiste` steht hier quasi für den Datentyp. Eine Variable `Kiste.Breite` gibt es daher nicht, sondern nur eine Variable `kleineKiste.Breite`.

2. Sie müssen beim Zugriff in jedem Fall den Bezeichner der Strukturvariablen und auch den Punkt angeben. Für die Vereinbarung von oben gibt es weder eine Variable `Breite` noch eine Variable `Hoehe`. Dem Compiler sind nur die Variablen `kleineKiste.Breite` und `kleineKiste.Hoehe` bekannt. Die Bezeichner `Breite` und `Hoehe` für sich allein betrachtet kennt der Compiler nur innerhalb der Vereinbarung der Struktur.

Sie müssen beim Zugriff auf ein Mitglied einer Struktur immer den Namen der Strukturvariablen, den Punkt und dahinter den Namen des Mitglieds angeben. Wenn Sie nur den Namen des Mitglieds angeben oder den Bezeichner der Struktur vor dem Punkt setzen, meldet der Compiler, dass er die Bezeichner nicht kennt.



Sie können aber außerhalb der Strukturvereinbarung durchaus eine Variable mit demselben Bezeichner vereinbaren. Möglich wäre also zum Beispiel auch die folgende Konstruktion:

```
struct Kiste
{
    public int Breite;
    public int Hoehe;
    public int Laenge;
}
Kiste kleineKiste;
int Breite;
```

Nach diesen Vereinbarungen kennt der Compiler jetzt sowohl die Variable `kleineKiste.Breite` als auch die Variable `Breite`. Außer dem teilweise identischen Namen haben diese Variablen aber nichts miteinander zu tun. Da diese scheinbar doppelte Verwendung eines Bezeichners gerade am Anfang für sehr viel Verwirrung sorgen kann, sollten Sie sie besser nicht benutzen.

Denn wenn Sie einen Namen nur innerhalb einer Struktur verwenden und die Angabe der Strukturvariablen beim Zugriff vergessen, erhalten Sie sofort eine Meldung vom Compiler und Sie können den Fehler korrigieren. Benutzen Sie denselben Namen dagegen noch einmal außerhalb einer Struktur, greifen Sie – möglicherweise ungewollt – auf die Variable außerhalb der Struktur zu. Eine Fehlermeldung erscheint dann nicht, da die Variable ja korrekt vereinbart ist.

Die Zuweisung von Werten und das Auslesen von Werten für die Mitglieder einer Struktur erfolgen wie gewohnt. Die Zeilen

```
kleineKiste.Breite = 2;
kleineKiste.Hoehe = 1;
kleineKiste.Laenge = 2;
```

weisen den drei Mitgliedern `Breite`, `Hoehe` und `Laenge` der Strukturvariablen `kleineKiste` die Werte 2, 1 und 2 zu.

Mit der Anweisung

```
Console.WriteLine("Die Breite der Kiste ist: {0}",  
kleineKiste.Breite);
```

geben Sie den Wert des Mitglieds Breite von kleineKiste aus.



Bitte beachten Sie:

Genau wie bei lokalen Variablen muss auch bei den Mitgliedern einer Struktur vor dem ersten lesenden Zugriff eine Initialisierung erfolgen. Andernfalls meldet der Compiler einen Fehler.

Die Initialisierung eines Mitglieds kann nicht direkt bei der Vereinbarung erfolgen. Die folgende Konstruktion würde also ebenfalls zu einer Fehlermeldung führen:

```
struct Kiste  
{  
    public int Breite = 10;  
    public int Hoehe;  
    public int Laenge;  
}
```

Wenn Sie die Strukturvariable als Array vereinbart haben, geben Sie wie gewohnt hinter dem Bezeichner den Index an. Dabei müssen Sie aber darauf achten, dass der Index hinter dem Bezeichner der Strukturvariablen stehen muss. Das könnte dann zum Beispiel so aussehen:

```
lagerraum[3].Breite = 10;
```

Hier wird dem Mitglied Breite des vierten Elements von lagerraum der Wert 10 zugewiesen. Die Anweisung

```
lagerraum.Breite[3] = 10;
```

dagegen versucht, dem vierten Element des Mitglieds Breite einer einfachen Strukturvariablen lagerraum einen Wert zuzuweisen. Und das klappt nur, wenn Breite als Array vereinbart wurde.

4.1.3 Arbeiten mit Strukturen

Nach den ganzen Vorüberlegungen wollen wir uns jetzt eine Struktur an einem praktischen Beispiel ansehen. Das folgende Programm liest die Breite, Höhe und Länge von einigen Kisten ein und errechnet in einer Methode das Volumen. Die Daten der Kisten legen wir dabei in einer Struktur ab.

Der Code sieht so aus:

```
/* #####  
Eine Struktur im Einsatz  
##### */  
  
using System;
```

```
namespace Cshp04d_04_01
{
    class Program
    {
        //die Vereinbarung der Struktur
        //sie erfolgt außerhalb der Methoden
        struct Kiste
        {
            public int Hoehe;
            public int Breite;
            public int Laenge;
        }

        //die Methode zum Einlesen
        //sie liefert die komplette Kiste zurück
        static Kiste Einlesen(int kistenNummer)
        {
            //eine lokale Strukturvariable
            Kiste aKiste;
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie die Höhe der {0}.");
            Kiste ein: ", kistenNummer);
            aKiste.Hoehe = Convert.ToInt32(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie die Breite der {0}.");
            Kiste ein: ", kistenNummer);
            aKiste.Breite = Convert.ToInt32(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.Write("Geben Sie die Länge der {0}.");
            Kiste ein: ", kistenNummer);
            Kiste.Laenge = Convert.ToInt32(Console.ReadLine());
            //eine leere Zeile ausgeben
            Console.WriteLine();
            return aKiste;
        }

        //die Methode zum Berechnen des Volumens
        static int Volumen(Kiste aKiste)
        {
            int volWert;
            volWert = aKiste.Breite * aKiste.Laenge * aKiste.Hoehe;
            return volWert;
        }

        static void Main(string[] args)
        {
            //ein lokales Array kleineKiste für die Struktur Kiste
            Kiste[] kleineKiste = new Kiste[3];

            //das Einlesen der Daten über die Methode Einlesen()
            for (int index = 0; index < 3; index++)
                kleineKiste[index] = Einlesen(index + 1);
        }
    }
}
```

```
//das Berechnen über die Methode Volumen()
for (int index = 0; index < 3; index++)
    //bitte in einer Zeile eingeben
    Console.WriteLine("Das Volumen von Kiste {0} ist: {1}",
        index + 1, Volumen(kleineKiste[index]));
}
```

Code 4.1: Arbeiten mit Strukturen**Tipp:**

Achten Sie beim Eingeben des Codes einmal auf die Memberliste für die Strukturvariable. Wenn Sie alles richtig vereinbart haben, werden Ihnen in dieser Liste die drei Mitglieder angeboten.

Schauen wir uns die Besonderheiten des vorigen Codes an. Da das Programm etwas komplexer ist, gehen wir die Anweisungen nicht mehr der Reihe nach durch, sondern simulieren quasi den Ablauf des Programms:

Zunächst einmal vereinbaren wir eine Struktur `Kiste`. Die Vereinbarung erfolgt dabei wie bei einer Klassenvariablen außerhalb jeder Methode, aber innerhalb der Klasse. Bitte beachten Sie, dass wir hier nur die Struktur vereinbaren und nicht die Strukturvariable.

In der Methode `Main()` vereinbaren wir dann eine lokale Strukturvariable `kleineKiste` für die Struktur `Kiste` als Array mit drei Elementen. Die Werte für die einzelnen Kisten werden über die Schleife

```
for (int index = 0; index < 3; index++)
    kleineKiste[index] = Einlesen(index + 1);
```

eingelesen. Dabei rufen wir die Methode `Einlesen()` auf und übergeben als Argument den aktuellen Wert von `index + 1`. Die Addition von 1 dient dabei eigentlich nur der Optik. Denn sonst würde die erste Kiste von unserem Programm mit der Nummer 0 bezeichnet.

In der Methode `Einlesen()` selbst vereinbaren wir eine weitere lokale Strukturvariable `aKiste` für die Struktur `Kiste` und belegen dann wie gewohnt die Mitglieder mit den Werten, die über die Tastatur eingegeben werden. Da eine Methode ja immer nur **einen** Wert zurückliefern kann, lassen wir die komplette Strukturvariable in einem Rutsch zurückgeben. Dabei werden automatisch auch die Werte der drei Mitglieder transportiert. Das funktioniert allerdings nur, wenn Sie als Rückgabetyp der Methode den Typ unserer Struktur angeben.

Die zurückgelieferten Werte werden dann mit der Anweisung

```
kleineKiste[index] = Einlesen(index + 1);
```

ebenfalls in einem Rutsch dem aktuellen Element von `kleineKiste` zugewiesen.

Merken Sie sich:

Wenn Sie nur den Bezeichner einer Strukturvariablen angegeben, beziehen Sie sich immer auf alle Mitglieder der Struktur gleichzeitig. Auf diese Weise können Sie zum Beispiel die Werte aus einer Strukturvariablen sehr leicht in eine andere Strukturvariable kopieren. Das klappt allerdings nur dann, wenn die Typen der Quelle und des Ziels identisch sind.



Danach wird für jede Kiste das Volumen über die Methode `Volumen()` berechnet. Als Argumente übergeben wir dabei zum einen eine laufende Nummer und zum anderen sämtliche Mitglieder der Strukturvariablen. Die Übergabe der Mitglieder erfolgt dabei wieder durch die Angabe des Bezeichners der Strukturvariablen.

So viel erst einmal zu Strukturen. Im weiteren Verlauf werden Sie bei der objektorientierten Programmierung noch die **Klassen** kennenlernen. Diese Klassen haben große Ähnlichkeit mit Strukturen, können aber unter anderem auch ihre Eigenschaften an andere Klassen vererben.

Kommen wir nun noch kurz zu Tupeln und Aufzählungstypen.

4.2 Tupel

Mit Tupeln können Sie ebenfalls mehrere Datenobjekte mit unterschiedlichen Typen zusammenfassen. Tupel haben Sie bereits eingesetzt – nämlich bei der Rückgabe von mehreren Werten aus einer Methode.

Zur Erinnerung:

Ein Tupel ist eine geordnete Menge von Werten.



Im einfachsten Fall besteht ein Tupel einfach nur aus einer Aufzählung von Werten in runden Klammern. Die einzelnen Werte werden dabei durch ein Komma getrennt. Die Anweisung

```
var testTupel = ("Text", 1, 2.09)
```

erzeugt ein Tupel mit dem Namen `testTupel` und den Werten „Text“, 1 und 2,09. Wie Sie sehen, lassen sich dabei auch ohne Weiteres unterschiedliche Datentypen mischen.

Der Zugriff auf die Werte erfolgt durch einen Punkt, den Text „Item“ und eine laufende Nummer hinter dem Bezeichner des Tupels. Die Anweisung

```
Console.WriteLine("Das erste Element hat den Wert {0}",  
    testTupel.Item1);
```

gibt also den ersten Wert im Tupel `testTupel` aus – in unserem Fall die Zeichenkette „Text“.

Da der Zugriff über die laufende Nummer nicht sonderlich komfortabel ist, können Sie auch Bezeichner verwenden. Die folgende Anweisung erstellt zum Beispiel ein Tupel mit dem Namen `kiste` und weist ihm Werte zu.

```
var kiste = (name: "Kiste 1", breite: 1.00, hoehe: 2.09,
laenge: 2.50)
```

Der Zugriff kann dann ebenfalls über den Namen erfolgen – zum Beispiel

```
Console.WriteLine("Der Name ist {0}", kiste.name);
```



Bitte beachten Sie:

Der Bezeichner `name` alleine ist außerhalb der Vereinbarung des Tupels nicht bekannt. Sie müssen immer auch den Bezeichner des Tupels und den Punkt angeben.

Sie können die Werte eines Tupels überall da verwenden, wo auch die Werte allein stehen könnten. Denkbar sind also auch Berechnungen wie im folgenden Beispiel:

```
var volumen = kiste.breite * kiste.laenge * kiste.hoehe
Console.WriteLine("Das Volumen beträgt: {0}.", volumen);
```

Auch die Zuweisung von Werten aus einem Tupel auf einfache Datenobjekte ist möglich. Neben der direkten Zuweisung können Sie dabei auch eine Kurzform wie im folgenden Beispiel verwenden:

```
var (name, breite, hoehe, laenge) = kiste
Console.WriteLine("Der Name ist {0}. Das Volumen beträgt {1}",
name, breite * hoehe * laenge);
```

Hier werden die Werte aus dem Tupel `kiste` auf die vier Datenobjekte `name`, `breite`, `hoehe` und `laenge` verteilt. Der Typ wird dabei automatisch bestimmt. Anschließend werden die Daten ausgegeben beziehungsweise in einer Berechnung verwendet.



Bitte beachten Sie:

Wenn Sie Werte aus einem Tupel zuweisen, können Sie dieselben Bezeichner verwenden, müssen es aber nicht. Sie können in dem Beispiel oben auch Bezeichner wie `wert1`, `wert2`, `wert3` und `wert4` verwenden.

Wenn Sie dieselben Bezeichner verwenden, müssen Sie beim Zugriff sehr sorgfältig zwischen dem Bezeichner im Tupel und dem Bezeichner außerhalb des Tupels unterscheiden.

Im praktischen Einsatz finden Sie Tupel noch einmal im folgenden Code:

```
/* #####
Tupel im Einsatz
#####
*/
using System;
namespace Cshp04d_04_02
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            //ein Tupel mit einer Zuweisung
            var testTupel = ("Text", 1, 2.09);
            //bitte jeweils in einer Zeile eingeben
            //das erste Element ausgeben
            Console.WriteLine("Das erste Element hat den Wert
{0}", testTupel.Item1);

            //ein Tupel mit benannten Elementen
            var kiste = (name: "Kiste 1", breite: 1.00, hoehe:
2.09, laenge: 2.50);
            //den Namen über den Namen ausgeben
            Console.WriteLine("Der Name ist {0}", kiste.name);

            //eine Rechnung mit den Elementen
            //bitte jeweils in einer Zeile eingeben
            var volumen = kiste.breite * kiste.laenge *
kiste.hoehe;
            Console.WriteLine("Das Volumen beträgt: {0}.",
volumen);

            //Zuweisung der Elemente auf einfache Datenobjekte
            var (name, breite, hoehe, laenge) = kiste;
            //bitte in einer Zeile eingeben
            Console.WriteLine("Der Name ist {0}. Das Volumen
beträgt {1}", name, breite * hoehe * laenge);
        }
    }
}
```

Code 4.2: Tupel im Einsatz

So viel zu den Tupeln. Kommen wir jetzt noch zu den Aufzählungstypen.

4.3 Aufzählungstypen

Über Aufzählungstypen können Sie die möglichen Werte eines Typs auf eine bestimmte Auswahl begrenzen – zum Beispiel auf Wochentage. Die Vereinbarung eines Aufzählungstyps wird durch das Schlüsselwort `enum`¹⁴ eingeleitet. Darauf folgen der Name des Typs und in geschweiften Klammern eine Liste mit den Werten. Abgeschlossen wird die Vereinbarung durch ein Semikolon.

Die Vereinbarung für einen Aufzählungstyp mit den Wochentagen würde zum Beispiel so aussehen:

```
enum Tage {Montag, Dienstag, Mittwoch, Donnerstag, Freitag,
Samstag, Sonntag};
```

14. `enum` steht als Abkürzung für *enumeration* (engl.: „Aufzählung“).

Einer Variablen vom Typ `Tage` können Sie dann nur einen der angegebenen Werte zuweisen.



Bitte beachten Sie:

Die Werte in einem Aufzählungstyp werden intern immer als ganze Zahlen abgebildet. Die Nummerierung beginnt dabei im Standard bei 0. Den Wert `Montag` in unserem Typ `Tage` betrachtet der Compiler also als 0, den Wert `Dienstag` als 1 und so weiter. Eine direkte Zuweisung eines numerischen Werts an die Variable eines Aufzählungstyps ist aber nur nach einem Casting möglich.

Die Vereinbarung eines Aufzählungstyps muss genau wie die Vereinbarung einer Struktur außerhalb jeder Methode erfolgen.

Schauen wir uns einen Aufzählungstyp nun in einem Programm an.

```
/*
#####
Aufzählungstyp
#####
*/
using System;

namespace Cshp04d_04_03
{
    class Program
    {
        //ein Aufzählungstyp
        bitte in einer Zeile eingeben
        enum Tage { Montag, Dienstag, Mittwoch, Donnerstag,
        Freitag, Samstag, Sonntag };

        static void Main(string[] args)
        {
            //eine Variable für den Aufzählungstyp
            Tage meineWoche;

            //Zuweisung auf die Variable des Aufzählungstyps
            meineWoche = Tage.Montag;
            //eine direkte Zuweisung eines numerischen Werts
            //erfordert ein Casting
            meineWoche = (Tage)(1);

            //Abfrage der Variablen
            if (meineWoche == Tage.Montag)
                Console.WriteLine("Heute ist Montag");
            else
                Console.WriteLine("Heute ist kein Montag");
        }
    }
}
```

Code 4.3: Aufzählungstypen

In dem Code wird zunächst einmal ein Aufzählungstyp für die Wochentage vereinbart. In der Methode `Main()` erzeugen wir eine Variable `meineWoche` für unseren Aufzählungstyp und weisen ihr den Wert `Montag` zu. Dabei müssen Sie vor den Wert noch einmal den Namen des Aufzählungstyps setzen und die beiden Angaben durch einen Punkt trennen. Andernfalls kennt der Compiler den Wert `Montag` nicht.

Anschließend überprüfen wir den Wert in einer `if`-Abfrage. Auch hier muss vor den eigentlichen Wert der Name des Aufzählungstyps gesetzt werden. Neben der direkten Abfrage des Wertes können Sie auch einen Vergleich mit dem numerischen Wert durchführen. Dabei ist dann allerdings wieder ein Casting erforderlich. Statt

```
if (meineWoche == Tage.Montag)
```

wäre auch

```
if (meineWoche == (Tage) 0)
```

möglich.

Da Aufzählungstypen nur in sehr speziellen Fällen eingesetzt werden, wollen wir uns mit diesen benutzerdefinierten Typen hier nicht weiter beschäftigen.

Zusammenfassung

Strukturen, Tupel und Aufzählungstypen sind benutzerdefinierte Datentypen.

Bei einer Struktur müssen Sie neben der eigentlichen Vereinbarung des Typs auch noch eine Variable für den Typ anlegen.

Der Zugriff auf die Mitglieder einer Struktur erfolgt in der Form

```
<Name_der_Strukturvariable>.<Name_des_Mitglieds>.
```

Über den Namen der Strukturvariablen können Sie die Mitglieder einer Struktur komplett an eine Methode übergeben oder in eine andere Variable derselben Struktur kopieren.

Über Tupel können Sie Daten ebenfalls zusammenhängend verarbeiten. Der Zugriff erfolgt über den Namen des Tupels und den Namen des Elements beziehungsweise die Angabe `Item` und eine laufende Nummer.

Über Aufzählungstypen können Sie die möglichen Werte eines Typs auf eine bestimmte Auswahl beschränken.

Die Vereinbarung eines Aufzählungstyps wird mit dem Schlüsselwort `enum` eingeleitet. Dann folgen der Name für den Typ und die Liste der möglichen Werte. Die Liste der Werte wird durch geschweifte Klammern umfasst.

Aufgaben zur Selbstüberprüfung

- 4.1 Sehen Sie sich bitte die folgende Vereinbarung einer Struktur genau an. Können Sie mit dieser Vereinbarung von außen auf die Mitglieder zugreifen? Wenn nein, was muss korrigiert werden?

```
struct Test
{
    int Mitglied1;
    float Mitglied2;
}
```

- 4.2 Ist der folgende Vergleich für eine Variable `tage` der Aufzählung `Wochentage` möglich? Begründen Sie bitte Ihre Antwort und korrigieren Sie gegebenenfalls die Anweisung.

```
tage == Montag;
```

- 4.3 Was unterscheidet eine Struktur von einem Tupel?

Schlussbetrachtung

Dieses Heft enthielt einige recht komplexe Themen. Sie dürfen zu Recht stolz auf sich sein, wenn Sie es durchgearbeitet haben.

Sie können in Ihren Programmen nun Variablen mit unterschiedlichen Bezugsrahmen verwenden, Arrays einsetzen und kennen einige wichtige Techniken für die Arbeit mit Zeichenketten.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Eine Klassenvariable muss außerhalb jeder Methode vereinbart werden – auch außerhalb der Methode `Main()`. Die Vereinbarung muss aber innerhalb der Klasse für das Programm erfolgen.
- 1.2 `e` ist als **Klassenvariable** gültig in `Main()` und `Ausgabe()`.
`ergebnis` ist **lokal** gültig in `Ausgabe()`.
`zahl` ist **lokal** gültig in `Ausgabe()`.
`eingabe` ist **lokal** gültig in `Main()`.
- 1.3 Der Wert der Variablen `ausgabe` ist auch nach dem Aufruf der Methode `EinenAbziehen()` 100. In der Methode wird eine lokale Kopie des übergebenen Wertes erstellt und nur diese Kopie wird verändert. Das Original bleibt unverändert.

Kapitel 2

- 2.1 Das erste gültige Element hat den Index 0. Das letzte gültige Element hat den Index Anzahl – 1; also 5.
- 2.2 Die Fehler liegen im Aufruf der Methode `Eingabe()` aus `Main()` und in der Parameterliste von `Eingabe()`.
Beim Aufruf fehlt die Angabe des Arrays als Argument. Korrekt ist `Eingabe(testArray)`.
In der Parameterliste von `Eingabe()` wird zum einen der falsche Typ verwendet, zum anderen auch nur ein Element aus dem Array übergeben. Korrekt ist `static void Eingabe(float[] argArray)`.
- 2.3
 - a) 1
 - b) 7
 - c) 4
 - d) -2
 - e) -3
 - f) ungültig, da es kein fünftes Element in der dritten Dimension gibt

```

2.4    for (int index = 0; index < 10; index++)
        Console.WriteLine("Das Element hat den Wert
{0}" + test[index] + ,

        foreach (float element in test)
            Console.WriteLine("Das Element hat den Wert {0}",
element);

        for (int index = 0; index < test.Length; index++)
            Console.WriteLine("Das Element hat den Wert {0}",
test[index]);

```

Kapitel 3

- 3.1 Nein, die Anweisung ist nicht möglich. Über den Index sind beim Typ `string` nur lesende Zugriffe möglich, aber keine schreibenden.
- 3.2 Für eine leere Zeichenkette steht die Eigenschaft `String.Empty`.
- 3.3 Wenn die beiden Zeichenketten identisch sind, liefert die Methode `String.Compare()` den Wert 0.

Kapitel 4

- 4.1 Nein, der Zugriff ist nicht möglich. Es fehlt die Angabe von `public`. Richtig wäre die folgende Form:

```

struct Test
{
    public int Mitglied1;
    public float Mitglied2;
}

```

- 4.2 Nein, der Vergleich ist so nicht möglich, da der Compiler den Bezeichner `Montag` nicht kennt. Vor dem Bezeichner müssen Sie noch den Aufzählungstyp angeben. Der Vergleich muss also lauten:

```
tage == Wochentage.Montag;
```

- 4.3 Bei einer Struktur muss die Vereinbarung getrennt erfolgen. Bei einem Tupel können Sie die Vereinbarung einfach durch eine Zuweisung vornehmen.

B. Glossar

Array	In einem Array werden mehrere Variablen desselben Typs zu einer Einheit zusammengefasst. Ein Array kann mehrere Dimensionen haben.
Arrayelement	Ein Arrayelement ist eine Variable in einem Array. Sie wird durch den Index eindeutig identifiziert.
Aufzählungstyp	Über Aufzählungstypen können Sie die möglichen Werte eines Typs auf eine bestimmte Auswahl begrenzen – zum Beispiel auf Wochentage. Ein Aufzählungstyp ist ein benutzerdefinierter Datentyp.
Bezugsrahmen	Der Bezugsrahmen ist der Bereich eines Programms, in dem ein Datenobjekt angesprochen werden kann.
Binäre Suche	Die binäre Suche ist ein spezielles, vergleichsweise schnelles Suchverfahren. Es setzt voraus, dass die zu durchsuchenden Elemente sortiert vorliegen.
Call by reference	Bei einem call by reference wird die Referenz eines Objekts übergeben. Dadurch wird der Wert des Objekts selbst geändert – zum Beispiel in einer Methode.
Call by value	Bei einem call by value wird der Wert eines Objekts als Kopie übergeben. Nur diese Kopie wird verändert. Der eigentliche Wert des Objekts bleibt erhalten.
Element	Siehe Arrayelement
Feld	Als Feld wird in der Programmiersprache C# ein Attribut einer Klasse bezeichnet.
Index	Über den Index werden die einzelnen Elemente eines Arrays oder einer Listenstruktur angesprochen.
Instanzvariable	Eine Instanzvariable ist eine Variable, die für eine Instanz gültig ist. Sie entspricht einem Feld.
Klassenvariable	Eine Klassenvariable ist eine Variable, die für die gesamte Klasse gilt.
Lebensdauer	Die Lebensdauer bestimmt, wie lange ein Datenobjekt existiert.
Lokale Variable	Eine lokale Variable gilt nur innerhalb eines Anweisungsblocks.
Lokaler Bezugsrahmen	Bei einem lokalen Bezugsrahmen gilt ein Datenobjekt nur in einem bestimmten Teil des Programms.

Memberliste	Die Memberliste erscheint automatisch beim Eingeben von Quelltexten und schlägt Ihnen anhand Ihrer Eingaben mögliche Eigenschaften oder Methoden für ein Objekt vor.
Mitglied	Als Mitglied wird ein Teil einer Struktur bezeichnet.
Namensraum	Über den Namensraum kann die Gültigkeit von Bezeichnern festgelegt werden.
Referenz	Eine Referenz ist ein Verweis auf ein Objekt.
String	<i>String</i> ist eine andere Bezeichnung für Zeichenkette.
Tupel	Ein Tupel ist eine geordnete Menge von Werten.
Überdeckung	Bei der Überdeckung überlagert eine lokale Variable eine Klassenvariable oder ein Feld mit demselben Namen.
Zeichenkette	Eine Zeichenkette ist eine beliebige Folge von Zeichen. Eine Zeichenkette wird in C# durch doppelte Anführungszeichen umfasst.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019. Ideal für Programmieranfänger.* 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 2.1	Der Programmabbruch durch einen Zugriff außerhalb des gültigen Bereichs	25
Abb. 2.2	Ein mehrdimensionales Array als Tabelle	27
Abb. 2.3	Die Tabelle nach der Wertzuweisung auf das Element tabellenArray[2, 3]	28
Abb. 2.4	Das Verändern der Referenz eines Arrays durch eine direkte Zuweisung	32

E. Tabellenverzeichnis

Tab. 3.1 Methoden zur Veränderung von Zeichenketten 44

F. Codeverzeichnis

Code 1.1	Beispiel für eine lokale Variable	3
Code 1.2	Beispiel für einen nicht erlaubten Zugriff auf eine lokale Variable	4
Code 1.3	Code 1.2 – jetzt mit einer anderen Fehlermeldung	5
Code 1.4	Code 1.3 – jetzt ohne Fehlermeldung	7
Code 1.5	Ein Beispiel für einen sehr engen Bezugsrahmen	8
Code 1.6	Ein Problem mit lokalen Variablen	9
Code 1.7	Beispiel für eine Klassenvariable	11
Code 1.8	Gemeinsame Verwendung von lokalen Variablen und Klassenvariablen	12
Code 1.9	SO BITTE NICHT!	14
Code 1.10	Lokale Variablen statt Klassenvariablen	15
Code 2.1	Beispiel für ein Array	22
Code 2.2	Beispiel für die Überschreitung der Arraygrenzen	24
Code 2.3	Sicherer Zugriff auf die Elemente eines Arrays	27
Code 2.4	Ein mehrdimensionales Array	29
Code 2.5	Ein Array als Argument für eine Methode	31
Code 2.6	Eine „unechte“ Kopie eines Arrays	33
Code 2.7	Sonderfunktionen für Arrays	36
Code 3.1	Zeichenweise Ausgabe einer Zeichenkette	41
Code 3.2	Zeichenketten vergleichen	42
Code 3.3	Einige Veränderungen an Zeichenketten	45
Code 3.4	Alle Vorkommen eines Zeichens in einer Zeichenkette zählen	46
Code 3.5	Die Klasse StringBuilder	47
Code 4.1	Arbeiten mit Strukturen	56
Code 4.2	Tupel im Einsatz	59
Code 4.3	Aufzählungstypen	60

G. Sachwortverzeichnis

A	
Array	18
kopieren	32
mehrdimensionales	27
Sonderfunktionen	35
Übergabe an Methoden	30
Vereinbarung	18
Zugriff auf die Elemente	21
Arrayelement	18
Aufzählungstyp	59
B	
Bezugsrahmen	3
C	
call	
by reference	32
by value	9
D	
Datentyp	
benutzerdefinierter	50
Datentyp Code	40
Datentyp string	40
E	
Eigenschaft Length	25
Element	18
F	
Feld	10
foreach-Schleife	26
I	
Index	18
Instanzvariable	10
K	
Klasse StringBuilder	46
Klassenvariable	10
Kontext	3
L	
Length	25
M	
Member	50
Mitglied	50
R	
Referenz	19
S	
string	40
StringBuilder	46
Struktur	50
Vereinbarung	50
Zugriff auf Mitglieder	52
Strukturvariable	51
Suche	
binäre	35
T	
Tupel	50, 57
V	
Variable	
lokale	3
Z	
Zeichenkette	40
verändern	43
vergleichen	41

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP04D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Erstellen Sie eine „Lagerverwaltung“, die folgende Daten für Kisten speichern kann:

- eine eindeutige Nummer zur Identifikation jeder einzelnen Kiste,
- die Breite, Länge und Höhe jeder Kiste sowie
- das Volumen der Kiste.

Die Nummer zur Identifikation der Kiste können Sie nach einem beliebigen Schema selbst vergeben. Stellen Sie aber durch geeignete Verfahren sicher, dass bei der Eingabe einer neuen Kiste nicht eine bereits vergebene Nummer benutzt wird.

Das Volumen der Kiste soll automatisch vom Programm anhand der Breite, Länge und Höhe berechnet werden können.

Das Programm soll maximal Daten von 75 Kisten verwalten können und folgende Funktionen anbieten:

- Eingabe einer neuen Kiste,
- Löschen der Daten einer vorhandenen Kiste,
- Ändern der Daten einer vorhandenen Kiste,
- Anzeigen der Daten einer vorhandenen Kiste und
- eine Listenfunktion, die die Daten aller vorhandenen Kisten anzeigt.

Beim Löschen, Ändern und Anzeigen soll der Zugriff auf die Daten der Kiste über die Nummer der Kiste erfolgen.

Für die Umsetzung gelten folgende Vorgaben:

- Speichern Sie die Daten in einer Struktur und legen Sie ein Array in der erforderlichen Größe für diese Struktur an. Erstellen Sie dieses Array **lokal** in der Methode **Main()**. Verwenden Sie keine Klassenvariable.
- Stellen Sie sicher, dass beim Zugriff auf die Daten der Kisten die Arraygrenzen nicht verlassen werden.
- Erstellen Sie für das Eingeben, Löschen, Ändern, Anzeigen und Auflisten jeweils eigene Methoden.

- Sorgen Sie dafür, dass beim Löschen, Ändern, Anzeigen und Auflisten nur auf Einträge zugegriffen werden kann, für die bereits Daten erfasst wurden. Dazu können Sie zum Beispiel beim Start des Programms die Nummer jeder Kiste zunächst auf den Wert 0 setzen und beim Zugriff überprüfen, ob die Nummer der Kiste noch den Wert 0 hat. Um eine Kiste zu löschen, reicht es dann, die Nummer der Kiste wieder auf den Wert 0 zu setzen.
- Erstellen Sie in der Methode `Main()` ein Auswahlmenü für den Zugriff auf die einzelnen Funktionen der Lagerverwaltung.

Verwenden Sie bitte folgenden Programmkopf:

```
/* ##### Einsendeaufgabe 4 ##### */
```

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Einführung in die objektorientierte
Software-Entwicklung

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP05D

Objektorientierte Software-Entwicklung mit C#

Einführung in die objektorientierte Software-Entwicklung

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Einführung in die objektorientierte Software-Entwicklung

Inhaltsverzeichnis

Einleitung	1
1 Theorie und Begriffe der objektorientierten Programmierung	3
1.1 Objekte, Klassen und Instanzen	3
1.2 Attribute, Attributwerte und Methoden	9
1.3 Objektidentität	10
1.4 Kommunikation zwischen Objekten	11
Zusammenfassung	14
2 Klassen und Objekte in C#	16
2.1 Vereinbarung einer Klasse	18
2.2 Instanzen erzeugen	20
2.3 Zugriff auf Methoden und Felder	21
Zusammenfassung	24
3 Die verschiedenen Methoden und Variablen unter der Lupe	26
3.1 Was Klassenmethoden und Instanzmethoden unterscheidet	26
3.2 Was Klassenvariablen und Felder unterscheidet	27
3.3 Das Problem der Überdeckung	30
Zusammenfassung	33
4 Zusammenarbeit zwischen Methoden	35
Zusammenfassung	41
5 Eine einfach verkettete Liste	42
5.1 Die Version mit Hilfskonstruktion	43
5.2 Die Version mit „echten“ Objekten	50
5.3 Stacks und Warteschlangen	60
Zusammenfassung	61
Schlussbetrachtung	62

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	63
B.	Glossar	65
C.	Literaturverzeichnis	69
D.	Abbildungsverzeichnis	70
E.	Codeverzeichnis	71
F.	Sachwortverzeichnis	72
G.	Einsendeaufgabe	75

Einleitung

Ab diesem Studienheft werden wir uns intensiver mit der **objektorientierten Programmierung** beschäftigen. Dabei lernen Sie zunächst etwas Theorie und die Begrifflichkeiten. Anschließend zeigen wir Ihnen an einigen Beispielen, wie Sie die objektorientierte Programmierung in C# umsetzen. Den Abschluss dieses Studienhefts bildet dann ein etwas größeres Projekt – eine einfach verkettete Liste, die sich dynamisch selbst Speicherplatz beschafft.

Im Einzelnen lernen Sie in diesem Studienheft,

- was sich hinter den Begriffen Objekt, Klasse und Instanz verbirgt,
- wie Objekte unterschieden werden,
- was Attribute und Methoden eines Objekts sind,
- wie Objekte kommunizieren,
- wie Sie Klassen in C# vereinbaren,
- wie Sie Instanzen in C# erzeugen,
- wie Sie auf Felder und Methoden einer Instanz zugreifen,
- was Klassenmethoden und Instanzmethoden unterscheidet,
- was Klassenvariablen und Felder unterscheidet,
- wie Sie mit dem Problem der Überdeckung umgehen,
- wie Methoden zusammenarbeiten können und
- wie Sie eine einfach verkettete Liste mit Objekten programmieren.

Christoph Siebeck

1 Theorie und Begriffe der objektorientierten Programmierung

In diesem Kapitel beschäftigen wir uns mit der Theorie und den Begriffen der objektorientierten Programmierung.

In vielen älteren Programmiersprachen wurde zwischen den eigentlichen Daten und dem Verhalten der Daten streng getrennt. So wurden einige Variablen vereinbart und diese Variablen dann über Funktionen verändert. Solch eine Funktion entspricht von der Arbeitsweise her einer Methode, befindet sich allerdings nicht in einer Klasse.

Ein direkter Zusammenhang zwischen den Daten und den Operationen, die mit den Daten ausgeführt wurden, bestand dabei eigentlich nicht. Die Daten und die Operationen befanden sich häufig lediglich im selben Quelltext.

Diese Trennung von Daten und ihrem Verhalten entspricht nun aber keineswegs dem üblichen menschlichen Denken. Niemand betrachtet zum Beispiel ein Auto getrennt nach seinen Daten (Größe, Farbe und so weiter) und seinem Verhalten (zum Beispiel kann fahren). In der Regel werden Daten und Verhalten immer gemeinsam betrachtet – wenn auch mit unterschiedlichen Schwerpunkten.

Genau diesem Ansatz folgt auch die **objektorientierte Programmierung**: Daten werden direkt mit den Operationen, die für die Daten ausgeführt werden können – dem Verhalten – verknüpft. Damit wird die Trennung zwischen den Daten und ihrem Verhalten aufgehoben. Daten und ihr Verhalten werden als ein Ganzes betrachtet – als ein **Objekt**.

C# ist eine vollständig objektorientierte Programmiersprache. In anderen Programmiersprachen – wie zum Beispiel C++ – können Sie objektorientiert programmieren, müssen es aber nicht.



Schauen wir uns jetzt genauer an, was sich hinter diesen Begriffen verbirgt.

1.1 Objekte, Klassen und Instanzen

Das **Objekt** ist der Dreh- und Angelpunkt der Objektorientierung.

Jede in sich abgeschlossene Einheit wird als Objekt betrachtet.



Dabei spielt es keine Rolle, ob es sich um einen Gegenstand aus der realen Welt handelt – zum Beispiel eine Person, ein Auto, einen Computer oder dieses Studienheft – oder um den Bestandteil eines Computerprogramms – zum Beispiel ein Dialogfenster.

Jedes Objekt besteht aus zwei Teilen:

- den Daten, die in dem Objekt enthalten sind und die den Zustand des Objekts beschreiben, sowie
- dem Verhalten des Objekts.



Abb. 1.1: Ein Objekt

Das Verhalten kann den Zustand des Objekts ändern. Diese Änderung ist dabei nicht nur auf das Objekt selbst beschränkt. Es ist auch möglich, dass das Verhalten von Objekt A den Zustand von Objekt B verändert.

Schauen wir uns den Zustand und das Verhalten an einem Beispiel aus der realen Welt an – einem Auto.

Das Auto ist das **Objekt**. Zum **Zustand** gehören dann zum Beispiel:

- der Motor läuft beziehungsweise läuft nicht,
- die Position des Autos,
- die Richtung, in die das Auto fährt,
- seine Geschwindigkeit und
- noch vieles mehr.

Zum **Verhalten** des Autos gehören unter anderem:

- Starten durch Drehen des Zündschlüssels,
- Lenken durch Drehen am Lenkrad und
- Beschleunigen durch das Gaspedal.

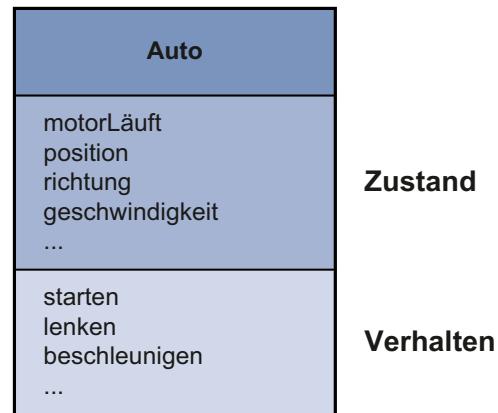
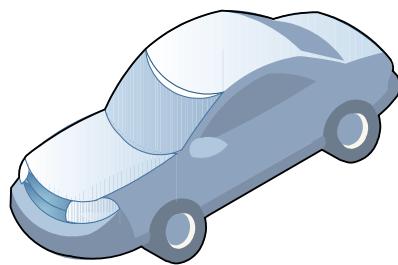


Abb. 1.2: Das Objekt „Auto“

Das Verhalten des Objekts „Auto“ verändert in unseren Beispielen immer auch den Zustand des Objekts „Auto“:

- Wenn Sie am Zündschlüssel drehen, ändert sich der Zustand „Motor läuft“.
- Wenn Sie lenken, ändert sich die Richtung, in die das Auto fährt.
- Wenn Sie beschleunigen, erhöht sich die Geschwindigkeit.

Das Objekt „Auto“ kann auch den Zustand eines anderen Objekts verändern – zum Beispiel den Zustand eines Anhängers. Wenn das Objekt „Auto“ das Objekt „Anhänger“ zieht, verändert sich durch ein Verhalten des Objekts „Auto“ der Zustand des Objekts „Anhänger“.

Ein reales Auto verfügt nun über eine ganze Reihe von Informationen, die den Zustand beschreiben. So könnten Sie neben dem Zustand des Motors, der Position des Autos, der Richtung und der Geschwindigkeit zum Beispiel auch die Farbe, den Ölstand und die Profiltiefe der Reifen als einen Teil des Zustands ansehen. Diese Reihe ließe sich nahezu beliebig fortführen.

Um das Objekt „Auto“ im Computer abzubilden, ist ein Großteil der möglichen Informationen in der Regel gar nicht erforderlich. Wenn Sie zum Beispiel berechnen wollen, wie lange ein Auto von einem Ort zu einem anderen Ort unterwegs ist, spielt die Farbe des Autos überhaupt keine Rolle. Sie hat keine Auswirkung auf die Geschwindigkeit des Autos und kann daher ignoriert werden.

Die Objekte werden daher **abstrahiert** dargestellt. Es werden nur die Informationen über den Zustand gespeichert, die zur Lösung eines konkreten Problems erforderlich sind. Für die Berechnung der Reisedauer könnten das zum Beispiel die Geschwindigkeit und die Position sein.



Abb. 1.3: Das abstrahierte Objekt „Auto“

Bitte beachten Sie:

Wie die Abstraktion eines Objekts erfolgt, hängt immer von der Aufgabenstellung ab. Für die Berechnung der Reisedauer wäre die Reifengröße zum Beispiel uninteressant. Für einen Reifenhändler wären die entsprechenden Daten dagegen von großer Bedeutung.



Die Informationen, die zu einem Objekt gespeichert werden, können bei einzelnen Objekten gleich oder sehr ähnlich sein. So treffen zum Beispiel die Informationen Position und Geschwindigkeit für jedes Auto zu. Um diese gleichen oder sehr ähnlichen Informationen nicht immer wieder neu für jedes Objekt anlegen zu müssen, werden Objekte, die gleiche oder sehr ähnliche Eigenschaften haben, in **Klassen** zusammengefasst.

Eine Klasse entspricht einer Art Bauplan, der die wesentlichen Eigenschaften und das wesentliche Verhalten eines Objekts beschreibt.



So könnten wir zum Beispiel eine Klasse „Auto“ erstellen, die die wesentlichen Eigenschaften und das wesentliche Verhalten aller Autos beschreibt.

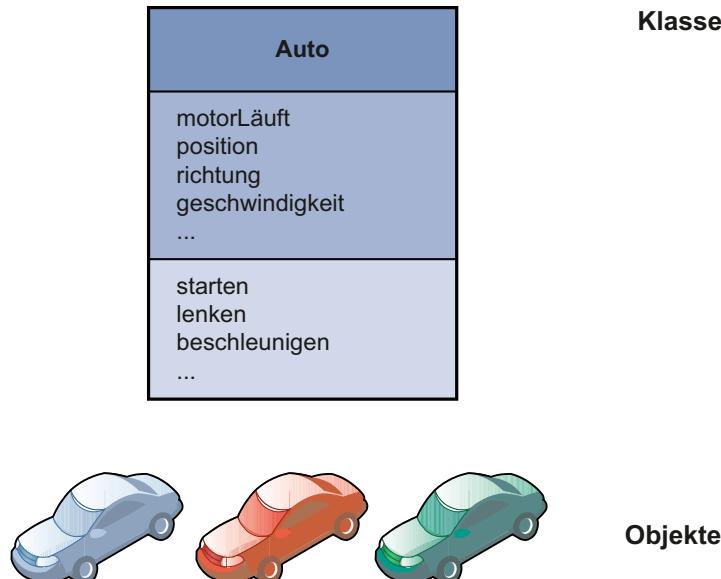


Abb. 1.4: Die Klasse „Auto“

Um möglichst flexibel zu bleiben, werden in der Praxis mehrere Klassen in einen hierarchischen Zusammenhang gebracht. Diese **Klassenhierarchie** beschreibt dann in immer feineren Details die jeweiligen Eigenschaften.

Schauen wir uns das an einem Beispiel an:



Beispiel 1.1:

Sie wollen einen Sportwagen und einen Geländewagen beschreiben. Sie könnten nun für die Sportwagen eine Klasse anlegen und eine andere Klasse für die Geländewagen. Dabei müssten Sie aber wesentliche Eigenschaften wie die Position und die Geschwindigkeit mehrfach beschreiben, obwohl kein Unterschied besteht. Durch die Erstellung mehrerer Klassen, die hierarchisch strukturiert werden, lässt sich dieses Problem elegant umgehen.

Sie erstellen zuerst ein Klasse „Auto“, die die Spitze der Hierarchie bildet. In dieser Klasse werden die Eigenschaften beschrieben, die für alle Autos gelten – also zum Beispiel die Geschwindigkeit und die Position.

Unterhalb dieser Klasse legen Sie dann zwei neue Klassen an: eine Klasse für die Sportwagen und eine Klasse für die Geländewagen. In diesen Klassen werden dann nur noch die typischen Eigenschaften der jeweiligen Klassen beschrieben – zum Beispiel die Motorisierung, die Bereifung oder die Form der Karosserie.

Die anderen allgemeingültigen Eigenschaften werden aus der übergeordneten Klasse **abgeleitet**. In unserem Beispiel bedeutet das: Die beiden Klassen „Sportwagen“ und „Geländewagen“ haben sämtliche Eigenschaften – sowohl Zustände als auch Verhalten – der Klasse „Auto“ übernommen und wurden zusätzlich um besondere Eigenschaften erweitert.

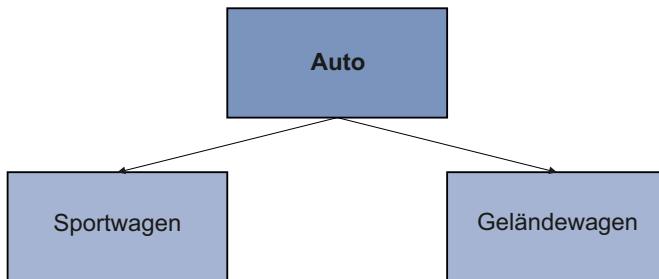


Abb. 1.5: Klassenhierarchie

Auf den Computer übertragen könnten Sie zum Beispiel eine übergeordnete Klasse „Fenster“ erstellen. Aus dieser Klasse können Sie dann die Klassen „Programmfenster“, „Eingabefenster“ und „Ausgabefenster“ ableiten. In der Klasse „Fenster“ werden die allgemeinen Eigenschaften beschrieben – zum Beispiel die Größe und die Position auf dem Bildschirm. In den untergeordneten Klassen finden sich dann die Besonderheiten – zum Beispiel die Anzahl der Symbolleisten oder der Schaltflächen im Fenster.

Wie Sie ja bereits wissen, ist eine Klasse immer nur der Bauplan für ein Objekt. Die einzelnen Objekte einer Klasse – die **Instanzen** – müssen erst angelegt beziehungsweise erstellt werden.

Eine Instanz ist ein Objekt genau einer Klasse.



Bitte beachten Sie, dass die Zuordnung eines Objekts zu einer Klasse für die gesamte Lebensdauer des Objekts gilt. Ein Objekt der Klasse „Sportwagen“ können Sie also nicht nachträglich zu einem Objekt der Klasse „Geländewagen“ machen.

Zu einer Klasse kann es auch mehrere Instanzen geben. Das ist zum Beispiel dann der Fall, wenn Sie mehrere Sportwagen „bauen“. Jeder dieser Sportwagen ist eine eigene Instanz der Klasse „Sportwagen“.

Klassen, für die Instanzen erzeugt werden können, werden **konkrete Klassen** genannt. Konkrete Klassen befinden sich in der Klassenhierarchie in der Regel sehr weit unten.



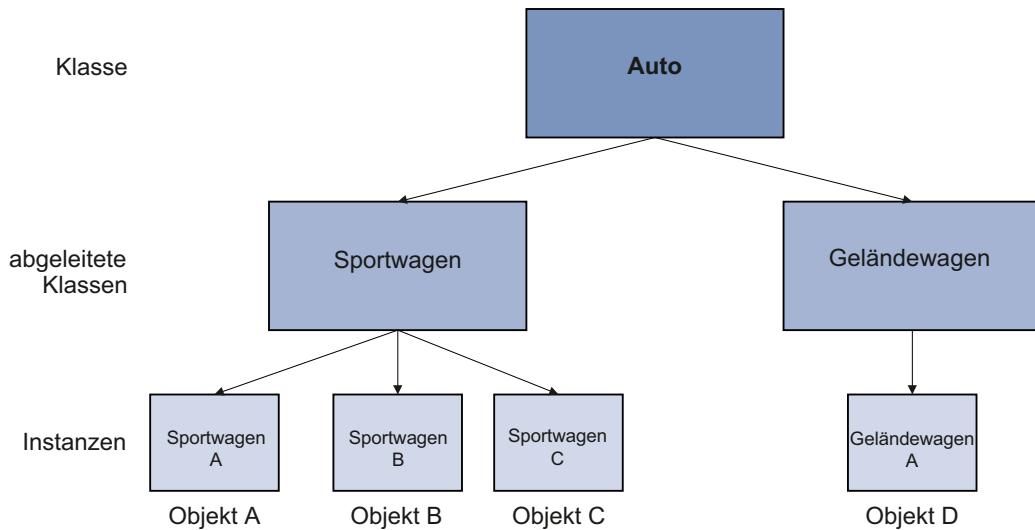


Abb. 1.6: Klassen und Instanzen

Allerdings gibt es nicht für jede Klasse auch Instanzen. So besitzt in der vorigen Abbildung zum Beispiel die Klasse „Auto“ keine direkte Instanz, sondern lediglich abgeleitete Klassen.



Für die Ableitung werden auch **abstrakte Klassen** genutzt.

In abstrakten Klassen kann ein allgemeines Verhalten vorgegeben werden, das in den darunter liegenden Klassen konkreter spezifiziert wird.

Von abstrakten Klassen können keine Instanzen erzeugt werden.

Mit der Modellierung von Klassen werden wir uns noch sehr detailliert im weiteren Verlauf beschäftigen, wenn wir unser großes Beispielprojekt entwickeln. Dann werden Sie unter anderem auch lernen, wie Sie Klassenhierarchien mit Klassendiagrammen darstellen.

1.2 Attribute, Attributwerte und Methoden

Wie Sie bereits wissen, besitzt ein Objekt einen Zustand und ein Verhalten.

Der Zustand in einem Objekt wird durch **Attribute** beschrieben. Jedes Attribut hat dabei einen Wert – den **Attributwert**.

Beispiel 1.2:

Ein Attribut eines Objekts „Auto“ ist „geschwindigkeit“. Der Attributwert für „geschwindigkeit“ wäre dann zum Beispiel 90 Kilometer pro Stunde.



Hinweis:

Zwischen dem Attribut und dem Attributwert wird nicht immer eindeutig unterschieden. Der Begriff Attribut steht dann sowohl für das eigentliche Attribut als auch für den Attributwert.

In C# werden die Attribute **Felder** genannt. Mit dem Unterschied zwischen Klassenvariablen, Feldern und lokalen Variablen werden wir uns im weiteren Verlauf des Studienhefts noch intensiv beschäftigen.

Das Verhalten wird durch die **Methoden** eines Objekts beschrieben. Die Methoden ändern in der Regel den Zustand des Objekts – also einen Attributwert beziehungsweise den Wert eines Feldes. Einen speziellen Typ dieser Methoden haben Sie ja bereits kennengelernt – nämlich die Klassenmethoden.

Die Methoden werden auch **Operationen** genannt.



Beispiel 1.3:

Eine Methode „beschleunigen“ für das Objekt „Auto“ ändert den Wert für das Attribut „geschwindigkeit“ – zum Beispiel auf 95 Kilometer pro Stunde.

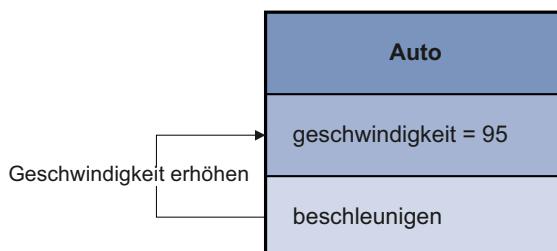


Abb. 1.7: Änderung der Attributwerte durch eine Methode

Die Methoden eines Objekts können entweder von **außen** aufgerufen werden – zum Beispiel durch ein anderes Objekt – oder von **innen** – also vom Objekt selbst. Auch hierzu ein Beispiel:



Beispiel 1.4:

Wenn das Objekt „Fahrer“ den Zündschlüssel umdreht, um den Motor zu starten, wird die Methode „starten“ von außen aufgerufen. Wenn dagegen der Motor Informationen über die aktuelle Drehzahl an den Drehzahlmesser weiterleitet, wird die entsprechende Methode vom Objekt selbst – also von innen – angestoßen.

Ein Objekt könnte auch die Attributwerte eines anderen Objekts verändern. Wenn das Objekt „Auto“ das Objekt „Anhänger“ zieht, ändert es ja auch den Wert des Attributs „geschwindigkeit“ des Objekts „Anhänger“. Grundsätzlich gilt aber:



Die Attributwerte eines Objekts sollten nur dem Objekt selbst bekannt sein.

Damit sollte auch nur das Objekt selbst Änderungen an seinen Attributwerten vornehmen können – und zwar über seine Methoden. Warum das so ist, erfahren Sie im weiteren Verlauf des Lehrgangs, wenn wir uns mit der Datenkapselung beschäftigen.

Schauen wir uns jetzt an, wie Objekte unterschieden werden.

1.3 Objektidentität

Nehmen wir einmal an, ein Autohersteller produziert mehrere baugleiche Sportwagen. Diese Objekte sind sich zwar alle gleich, haben aber trotzdem alle eine eigene **Identität**, die sie eindeutig voneinander unterscheidet. Andernfalls könnte man ja gar nicht auseinanderhalten, zu welchem Fahrer beziehungsweise Kunden ein Sportwagen gehört.



Die Objektidentität ist eindeutig und kann sich nicht ändern. Sie ist unabhängig von den anderen Eigenschaften eines Objekts.

Die Objektidentität kann zum Beispiel über einen eindeutigen **Objektbezeichner** ausgedrückt werden. Dieser Bezeichner wird sozusagen bei der „Geburt“ des Objekts vergeben und identifiziert dann das Objekt so lange eindeutig, bis es nicht mehr existiert.

So könnte zum Beispiel das erste Objekt „Sportwagen“ den Objektbezeichner „Sportwagen 1“ erhalten, das zweite Objekt den Bezeichner „Sportwagen 2“ und so weiter. Mehrere Fensterobjekte ließen sich über die Bezeichner „Fenster 1“, „Fenster 2“ und so weiter unterscheiden. Über den Objektbezeichner wäre immer eindeutig klar, welches Objekt denn nun genau gemeint ist.

Hinweis:

Die Objektidentität und der Objektbezeichner lassen sich mit dem Bezeichner einer Variablen vergleichen. Wie Sie ja wissen, muss auch ein Variablenbezeichner innerhalb eines Bezugsrahmens immer eindeutig sein.

Bitte beachten Sie, dass Objekte immer ihre eigene Identität haben. Das ist auch dann der Fall, wenn es zufällig zwei Objekte mit absolut identischen Eigenschaften gibt – zum Beispiel eineiige Zwillinge. Diese Objekte sind zwar gleich, können aber nicht dieselbe Objektidentität haben. Andernfalls wäre ja keine eindeutige Unterscheidung mehr möglich.

Bei Objekten, die absolut identisch sind, spricht man auch von Objektgleichheit.



1.4 Kommunikation zwischen Objekten

Nachdem Sie jetzt die grundlegenden Begriffe der objektorientierten Programmierung kennen, wollen wir uns mit der Kommunikation zwischen Objekten beschäftigen.

Wie Sie ja bereits wissen, sollte ein Objekt nie direkt auf die Attribute eines anderen Objekts zugreifen. Da stellt sich die Frage: Wie können dann mehrere Objekte zusammenarbeiten?

Schauen wir uns dazu ein **Beispiel** an:

Beispiel 1.5:



In sehr vielen Programmen finden Sie die Funktion **Datei öffnen**. Diese Funktion zeigt normalerweise ein Dialogfenster an, in dem der Anwender die gewünschte Datei markiert und die Auswahl bestätigt.

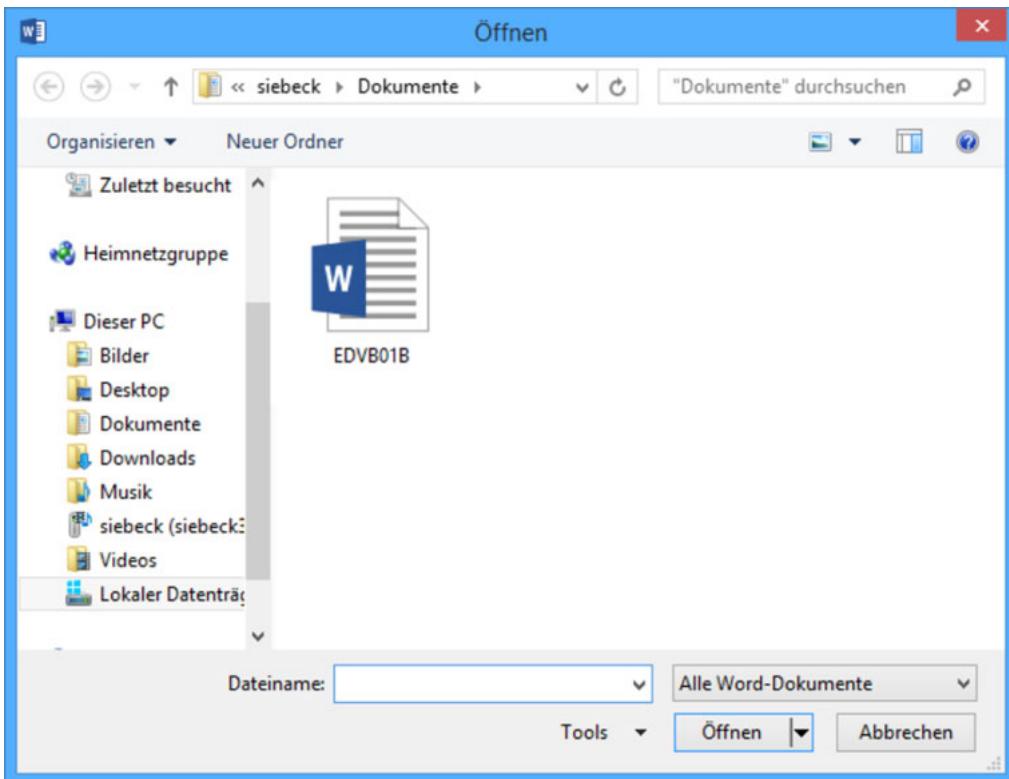


Abb. 1.8: Ein Dialog zum Öffnen von Dateien
(hier: Microsoft Word)

Dieses Dialogfenster ist aber nur dann sinnvoll, wenn die Datei nach dem Bestätigen auch in dem Fenster der eigentlichen Anwendung angezeigt werden kann. Hier muss also das eine Objekt – das Dialogfenster – dem anderen Objekt – dem Fenster der Anwendung – mitteilen, was geschehen soll. Die Objekte müssen **kommunizieren**.



Die Kommunikation zwischen Objekten erfolgt über **Nachrichten**. Die Menge aller Nachrichten, auf die ein Objekt reagieren kann, wird **Protokoll** genannt. Das Protokoll lässt sich mit der Sprache vergleichen, in der ein Objekt kommuniziert.

Wie bei jeder anderen Art von Kommunikation gibt es auch bei der Kommunikation zwischen Objekten einen **Sender** oder *sender* und einen **Empfänger** – auch *receiver*¹ genannt. Der Sender schickt die Nachricht zum Empfänger. Der Empfänger verarbeitet dann die Nachricht mit einer speziellen Methode für die Nachrichtenbearbeitung. Der Sender muss dabei nicht wissen, was der Empfänger mit der Nachricht macht – er stößt lediglich die Ausführung beim Empfänger an. Wie die Ausführung erfolgt, ist völlig unabhängig vom Sender.

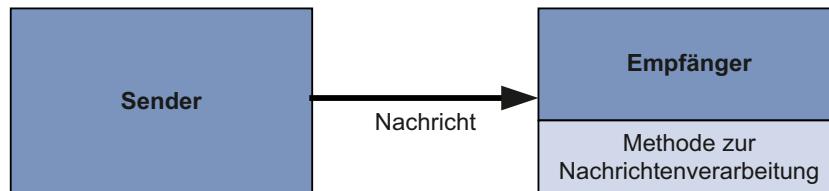


Abb. 1.9: Kommunikation durch Nachrichten

Schauen wir uns das einmal an dem Beispiel mit dem Dialog **Öffnen** von oben an:



Beispiel 1.6:

In dem Dialog wählt der Anwender zuerst die gewünschte Datei aus. Danach klickt er auf die Schaltfläche **Öffnen**. Der Sender – das Dialogfenster – verschiickt jetzt an den Empfänger – das Fenster der eigentlichen Anwendung – eine Nachricht wie „Öffne die Datei Brief“. Der Empfänger nimmt die Nachricht entgegen und verarbeitet sie weiter. Was der Empfänger mit der Nachricht macht und wie er es macht, ist dabei für den Sender völlig uninteressant. Er hat nach dem Verschicken der Nachricht seine Arbeit erledigt.

Grundsätzlich lassen sich zwei Arten von Nachrichten unterscheiden:

- der **Operationsaufruf** und
- das **Signal**.

Beim **Operationsaufruf** sind genau zwei Objekte beteiligt. Der Sender ruft eine Methode des Empfängers auf. Der Empfänger muss dann die angeforderte Operation ausführen und ändert dabei möglicherweise seinen Zustand oder andere Daten. Der Sender „befiehlt“ also dem Empfänger, was er zu tun hat. Was der Empfänger dann allerdings genau mit diesem „Befehl“ macht, kann der Sender nicht beeinflussen.

1. *Receiver* ist die englische Übersetzung für „Empfänger“.

Bei einem **Signal** wird eine Nachricht von einem Sender abgesetzt und **kann** von anderen Objekten empfangen werden. Ob ein Empfänger überhaupt auf ein Signal reagiert und wie er das Signal verarbeitet, hängt alleine vom Empfänger ab. Anders als bei einem Operationsaufruf können bei der Kommunikation über Signale auch mehrere Empfänger gleichzeitig aktiv sein.

Die Kommunikation über Signale finden Sie beispielsweise bei Ampelanlagen mit der Schaltung einer „grünen Welle“. Wenn eine Ampel auf Grün geschaltet wird, sendet sie ein Signal an die nächste Ampel. Dieses Signal führt dann bei der empfangenden Ampel dazu, dass sie ebenfalls auf Grün schaltet. Anders als beim Operationsaufruf teilt der Sender dem Empfänger bei der Kommunikation über Signale nicht mit, was er zu tun hat. Die sendende Ampel meldet also nur „Ich bin auf Grün geschaltet“. Welche Zustandsänderung dieses Signal beim Empfänger bewirkt, hängt vom Empfänger ab.

Schauen wir uns den Unterschied zwischen einem Operationsaufruf und einem Signal noch an einem anderen Beispiel an:

Beispiel 1.7:

In einem Dialogfenster gibt es eine Schaltfläche. Wenn diese Schaltfläche beim Anklicken eine Nachricht an ein anderes Objekt schickt, was dieses Objekt machen soll, handelt es sich um einen Operationsaufruf.



Schickt die Schaltfläche beim Anklicken dagegen nur eine Nachricht los „Ich bin angeklickt worden“, handelt es sich um ein Signal. Ob irgendein anderes Objekt auf dieses Signal reagiert, hängt allein von den Objekten ab, die das Signal empfangen können.

Abhängig von der Anzahl der aktiven Empfänger kann man auch zwischen der **sequenziellen** und der **parallelen Nachrichtenübertragung** unterscheiden.

Bei der **sequenziellen Übertragung** ist immer nur ein Empfänger aktiv. Die sequenzielle Übertragung wird zum Beispiel für die „grüne Welle“ bei Ampeln eingesetzt. Die erste Ampel schickt ein Signal an die zweite Ampel, die zweite Ampel schickt dann ein Signal an die dritte Ampel und so weiter.

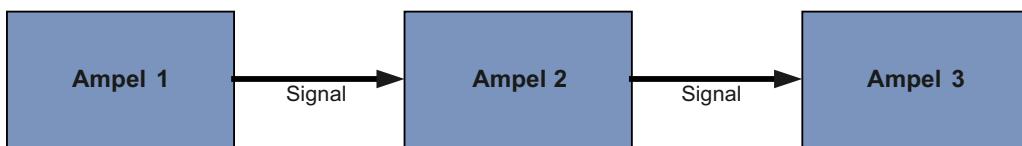


Abb. 1.10: Sequenzielle Übertragung

Bei der **parallelen Übertragung** dagegen können auch mehrere Empfänger die Nachricht entgegennehmen. Das heißt, durch ein Signal können auch mehrere Objekte ihren Zustand ändern. Eine parallele Übertragung findet zum Beispiel statt, wenn bei einer Ampelschaltung mehrere Ampeln gleichzeitig auf Rot geschaltet werden. Die erste Ampel schickt gleichzeitig ein Signal an die Ampeln 2 und 3.

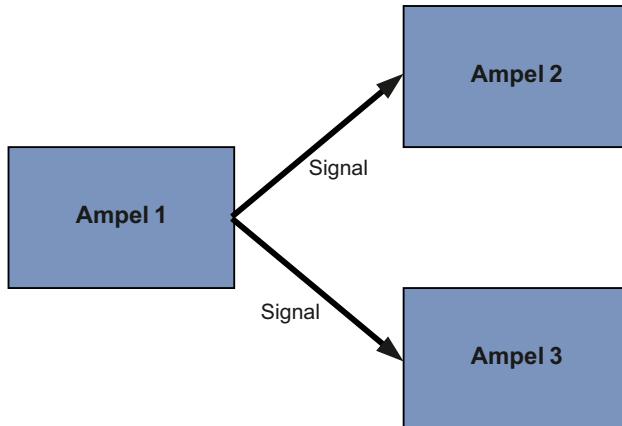


Abb. 1.11: Parallele Übertragung

Ein weiteres Unterscheidungsmerkmal für die Nachrichtenübertragung ist das Verhalten des Senders nach dem Versenden der Nachricht.

Wenn der Sender wartet, bis seine Nachricht verarbeitet wurde und gegebenenfalls eine Nachricht zurückkommt, nennt man das **synchrone Kommunikation**. Synchrone Kommunikation findet zum Beispiel statt, wenn ein Objekt ein anderes mit der Datensuche beauftragt und das Ergebnis der Suche zurückgeliefert wird, bevor der Sender weiterarbeitet.

Bei der **asynchronen Kommunikation** wartet der Sender dagegen nicht, bis der Empfänger die Nachricht verarbeitet hat. Der Rückgabewert wird in einer separaten Nachricht zurückgeschickt. Asynchrone Kommunikation findet zum Beispiel statt, wenn ein Objekt ein anderes auffordert, eine Aktion im Hintergrund durchzuführen. Der Sender wartet nicht ab, bis die Aktion durchgeführt ist, sondern arbeitet weiter. Ein typisches Beispiel für asynchrone Kommunikation ist das Drucken eines Dokuments, während Sie das Dokument weiter bearbeiten.

So viel an dieser Stelle zu den Grundlagen der objektorientierten Programmierung.

Zusammenfassung

Ein Objekt bildet eine in sich geschlossene Einheit der Realität ab. Es fasst Verhalten und Eigenschaften zusammen.

Ähnliche Objekte werden zu Klassen zusammengefasst. Klassen können in einem hierarchischen Verhältnis stehen.

Die Attributwerte eines Objekts sollten nur über die Methoden des Objekts geändert werden – also nur vom Objekt selbst.

Jedes Objekt hat eine eindeutige Identität.

Objekte kommunizieren durch den Versand von Nachrichten. Dabei werden zwei Arten unterschieden: der Operationsaufruf und das Signal.

Abhängig vom Verhalten des Senders wird zwischen synchroner und asynchroner Kommunikation unterschieden.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Nennen Sie mindestens drei charakteristische Eigenschaften eines Objekts.

- 1.2 Was ist eine Instanz einer Klasse?

- 1.3 Was unterscheidet konkrete Klassen und abstrakte Klassen?

- 1.4 Handelt es sich bei der Kommunikation zwischen einem Objekt „Fernbedienung“ und einem Objekt „Fernseher“ um synchrone oder um asynchrone Kommunikation?

2 Klassen und Objekte in C#

In diesem Kapitel werden Sie die Theorie in die Praxis umsetzen. Sie erstellen Ihre ersten eigenen Klassen und Objekte mit C#.

Hinweis:

Sie haben bereits mit Klassen, Methoden und Klassenvariablen gearbeitet. So handelt es sich ja bei jedem Programm, das Sie bisher erstellt haben, um eine Klasse. Und die Ausgabeanweisung `WriteLine()` ist eine Methode der Klasse `System.Console`. Bei all diesen Klassen handelt es sich aber – wenn Sie so wollen – um vordefinierte Klassen beziehungsweise um spezielle Klassen. In diesem Kapitel werden Sie zum ersten Mal „richtige“ eigene Klassen erstellen.

Als Beispiel verwenden wir ein Programm, das zwei sehr stark vereinfachte Autos als Objekte abbildet. Wie gewohnt zeigen wir Ihnen zunächst den kompletten Code und sehen uns dann die einzelnen Zeilen der Reihe nach an.

```
/*#####
Einführung in Klassen
#####*/
using System;
namespace Cshp05d_02_01
{
    //die Vereinbarung der Klasse Autoklasse
    //Sie erfolgt außerhalb der Klasse für das Programm
    class Autoklasse
    {
        //ein Feld
        int geschwindigkeit;

        //die Methoden
        //zum Initialisieren
        public void Initialisiere(int standard)
        {
            geschwindigkeit = standard;
        }

        //zum Bremsen
        public void Bremsen(int aenderung)
        {
            if (geschwindigkeit - aenderung < 0)
                geschwindigkeit = 0;
            else
                geschwindigkeit = geschwindigkeit - aenderung;
        }

        //zum Gasgeben
        public void Gasgeben(int aenderung)
        {
            geschwindigkeit = geschwindigkeit + aenderung;
        }
    }
}
```

```
}

//zur Ausgabe der Geschwindigkeit
public void Ausgeben()
{
    //bitte in einer Zeile eingeben
    Console.WriteLine("Die aktuelle Geschwindigkeit beträgt
        {0}.", geschwindigkeit);
}

class Program
{
    static void Main(string[] args)
    {
        //zwei neue Instanzen für Autoklasse erzeugen
        Autoklasse auto1 = new Autoklasse();
        Autoklasse auto2 = new Autoklasse();

        //die Methode Initialisiere() für die beiden
        //Autos aufrufen
        auto1.Initialisiere(0);
        auto2.Initialisiere(10);

        //die Geschwindigkeit ausgeben
        Console.WriteLine("Nach der Initialisierung");
        auto1.Ausgeben();
        auto2.Ausgeben();

        //Methode Gasgeben() aufrufen
        auto1.Gasgeben(20);
        auto2.Gasgeben(100);

        Console.WriteLine("Nach Gasgeben");
        auto1.Ausgeben();
        auto2.Ausgeben();

        //Methode Bremsen() aufrufen
        auto1.Bremsen(10);
        auto2.Bremsen(50);

        Console.WriteLine("Nach Bremsen");
        auto1.Ausgeben();
        auto2.Ausgeben();
    }
}
```

Code 2.1: Einführung in Klassen

Vieles im vorigen Code sollte Ihnen schon bekannt vorkommen. Schauen wir uns die einzelnen Zeilen trotzdem der Reihe nach an. Denn es gibt einige Besonderheiten. Beginnen wir mit der Vereinbarung der Klasse.

2.1 Vereinbarung einer Klasse

Die Vereinbarung der Klasse Autoklasse erfolgt direkt zu Beginn des Codes mit den Zeilen

```
class Autoklasse
{
    //ein Feld
    int geschwindigkeit;
    //die Methoden
    //zum Initialisieren
    public void Initialisiere(int standard)
    {
        geschwindigkeit = standard;
    }

    //zum Bremsen
    public void Bremsen(int aenderung)
    {
        if (geschwindigkeit - aenderung < 0)
            geschwindigkeit = 0;
        else
            geschwindigkeit = geschwindigkeit - aenderung;
    }

    //zum Gasgeben
    public void Gasgeben(int aenderung)
    {
        geschwindigkeit = geschwindigkeit + aenderung;
    }

    //zur Ausgabe der Geschwindigkeit
    public void Ausgeben()
    {
        Console.WriteLine("Die aktuelle
        Geschwindigkeit beträgt {0}.",
        geschwindigkeit);
    }
}
```

Eingeleitet wird die Vereinbarung über das Schlüsselwort `class` gefolgt vom Bezeichner der Klasse. Nach den C#-Konventionen beginnen Klassenbezeichner mit einem Großbuchstaben.

Dann folgen in geschweiften Klammern die Felder – die Attribute – und die Methoden der Klasse. Abgeschlossen wird die Vereinbarung mit einer schließenden geschweiften Klammer.

Wenn Sie sich die Klassenvereinbarung aus dem Beispiel noch einmal sehr genau ansehen, sollte Ihnen – neben der eigentlichen Vereinbarung über das Schlüsselwort `class` – vor allem eins auffallen: Sowohl bei der Vereinbarung der Variablen als auch in den Methodenköpfen fehlt das Schlüsselwort `static`, das wir bisher immer benutzt haben, um eine Variable oder eine Methode außerhalb der Klasse des Programms zu vereinba-

ren. Und genau hier liegt auch der entscheidende Unterschied: **Mit dem Schlüsselwort static vereinbaren Sie eine Klassenvariable beziehungsweise eine Klassenmethode, ohne das Schlüsselwort dagegen ein Feld beziehungsweise eine Instanzmethode.**

Statt Instanzmethode wird in der Regel einfach nur der Ausdruck Methode benutzt. Methoden, die keine Instanzmethoden sind, werden dagegen Klassenmethoden genannt.



Felder einer Klasse werden auch **Instanzvariablen** genannt.

Was den Unterschied zwischen der Vereinbarung mit `static` beziehungsweise ohne `static` genau ausmacht, werden wir uns in Kapitel 3 noch detailliert ansehen. Hier lassen wir es erst einmal dabei, dass wir das Feld und auch die Methoden ohne das Schlüsselwort `static` vereinbaren.

Schauen wir uns jetzt die Anweisungen in den Methoden einmal etwas genauer an.

In der Methode `Initialisiere()` setzen wir den Wert des Feldes `geschwindigkeit` auf einen Wert, den wir an die Methode übergeben. Dass das Feld `geschwindigkeit` zur Klasse `Autoklasse` gehört, erkennt der Compiler dabei von alleine.

Auch die Anweisungen in der Methode `Bremsen()` sollten Sie nicht vor große Probleme stellen. Es wird zunächst der Wert von `geschwindigkeit - aenderung` überprüft und dann der Wert des Feldes verändert.

In der Methode `Gasgeben()` ändern wir ebenfalls den Wert des Feldes `geschwindigkeit`, und mit der Methode `Ausgeben()` geben wir den aktuellen Wert des Feldes aus. Dabei gibt es ebenfalls keine Besonderheiten.

Bitte beachten Sie, dass alle Methoden zusätzlich durch das Schlüsselwort `public` gekennzeichnet werden. Es ermöglicht wie bei den Strukturen den Zugriff von außen.

Damit ist unsere Klasse vollständig. Wir haben jetzt also einen Bauplan für ein Auto mit dem Attribut – dem Feld – `geschwindigkeit` sowie den Methoden `Initialisiere()`, `Bremsen()`, `Gasgeben()` und `Ausgeben()` angelegt.

Hinweis:

In welcher Reihenfolge Sie die Felder und Methoden in der Vereinbarung einer Klasse aufführen, ist relativ beliebig. Normalerweise werden aber erst die Felder und dann die Methoden genannt.

2.2 Instanzen erzeugen

Im nächsten Schritt müssen wir nun für unsere Klasse `Autoklasse` Instanzen erzeugen.



Zur Erinnerung:

Mit der Vereinbarung einer Klasse erstellen Sie zunächst einmal nur einen Bauplan für die Objekte der Klasse. Es werden nicht automatisch Instanzen für diese Klasse angelegt. Das heißt, mit der Vereinbarung erzeugen Sie lediglich die Klasse an sich.

Um eine Instanz zu erzeugen, legen Sie zunächst eine Variable mit dem Typ der Klasse an. Das eigentliche Erzeugen der Instanz erfolgt dann mit dem Operator `new`. Dahinter geben Sie den Namen der Klasse und die runden Klammern `()` an.

Das Erzeugen der Instanzen übernehmen in unserem Beispielcode die ersten beiden Anweisungen aus der Methode `Main()`

```
Autoklasse auto1 = new Autoklasse();
Autoklasse auto2 = new Autoklasse();
```

Eine sehr ähnliche Technik haben wir ja auch bereits bei den Arrays benutzt. Die einzige Besonderheit liegt in den runden Klammern hinter dem Typ beim Operator `new`. Wenn Sie diese Klammern weglassen, beschwert sich der Compiler.



Mit den runden Klammern hinter dem Typ rufen Sie eine besondere Methode einer Klasse auf, die automatisch angelegt wird – den **Standardkonstruktor**. Was sich genau dahinter verbirgt, erfahren Sie später. Hier sollten Sie sich einfach nur merken, dass beim Erzeugen einer Instanz für eine Klasse die runden Klammern hinter der Typangabe beim Operator `new` zwingend erforderlich sind.

Wenn Sie möchten, können Sie die Anweisungen zum Erzeugen der Instanzen auch jeweils auf zwei Zeilen verteilen. Sie würden dann so aussehen:

```
Autoklasse auto1;
Autoklasse auto2;
auto1 = new Autoklasse();
auto2 = new Autoklasse();
```

Denken Sie aber auch hier unbedingt daran, die Instanz tatsächlich über `new` zu erzeugen. Andernfalls wird sich der Compiler beschweren, dass eine nicht zugewiesene Variable verwendet wird.

Nach dem Ausführen dieser Anweisungen haben wir zwei Instanzen der Klasse `Autoklasse` – also zwei voneinander unabhängige Objekte. Die eine Instanz können wir über den Objektbezeichner `auto1` ansprechen und die andere über den Objektbezeichner `auto2`.

2.3 Zugriff auf Methoden und Felder

Nach dem Erzeugen der Instanzen greifen wir auf die Methoden der Klasse zu. Zuerst werden die beiden Instanzen durch die Methode `Initialisiere()` initialisiert. Das übernehmen die beiden Anweisungen

```
auto1.Initialisiere(0);  
auto2.Initialisiere(10);
```

Bitte beachten Sie, dass in unserem Beispiel vor dem Punkt der Name einer Instanz stehen muss und nicht der Name einer Klasse. Die Anweisung

```
Autoklasse.Initialisiere(10);
```

führt zu einer Fehlermeldung, da `Autoklasse` ja die Klasse selbst ist und keine Instanz der Klasse. Wenn Sie sich die Fehlermeldung von C# allerdings etwas genauer ansehen, werden Sie feststellen, dass es hier wohl Probleme mit einem nicht statischen Feld beziehungsweise einer nicht statischen Methode gibt. Damit werden wir uns gleich noch etwas genauer beschäftigen.

Hinweis:

Denken Sie bitte daran, dass Sie beim Aufruf einer Methode in jedem Fall die runden Klammern angeben müssen – auch wenn Sie keine Argumente übergeben. Mit der Anweisung

```
auto1.Initialisiere;
```

versuchen Sie, auf das **Feld** `Initialisiere` der Instanz `auto1` zuzugreifen. Dieser Versuch wird vom Compiler mit einer Fehlermeldung quittiert, da das Feld `Initialisiere` für `auto1` nicht bekannt ist.

Felder werden genau wie Klassenvariablen automatisch mit dem Nullwert des Datentyps initialisiert. Das können Sie ganz einfach testen, indem Sie im Code die Anweisung

```
auto1.Initialisiere(0);
```

auskommentieren. Sie werden sehen, die Geschwindigkeit für die Instanz `auto1` beträgt beim ersten Aufruf der Methode `Ausgeben()` trotzdem 0.

Sie können ein Feld aber auch, wie von lokalen Variablen gewohnt, direkt mit der Zuweisung eines Wertes initialisieren.

Auch Felder behalten ihren Wert beim Verlassen einer Methode. Denn sie gelten ja für alle Methoden einer Klasse.

Nach der Methode `Initialisiere()` rufen wir dann noch weitere Methoden für die beiden Instanzen auf – zum Beispiel die Methode `Gasgeben()` mit den Anweisungen

```
auto1.Gasgeben(20);  
auto2.Gasgeben(100);
```

Wie Sie an den Ausgaben sehen können, werden die beiden Instanzen `auto1` und `auto2` eindeutig unterschieden, obwohl sie zur selben Klasse gehören. Die beiden Instanzen haben also ihre eigene Identität und auch ihre eigenen Felder und Methoden.

Wenn Sie sich den Code einmal genauer ansehen, werden Sie feststellen, dass wir auf das Feld `geschwindigkeit` der beiden Instanzen nie direkt zugreifen, sondern immer nur indirekt über die Methoden. Grundsätzlich wäre auch ein direkter Zugriff auf `geschwindigkeit` möglich, wenn das Feld wie die Methoden ebenfalls als `public` vereinbart wäre.

Dann könnten Sie zum Beispiel mit der Anweisung

```
auto1.geschwindigkeit = 30;
```

in der Methode `Main()` das Attribut `geschwindigkeit` von `auto1` auf den Wert 30 setzen.

Hinweis:

In unserem Beispielcode führt diese Anweisung zu einem Fehler, da das Feld nicht als `public` definiert ist. Damit ist ein direkter Zugriff von außen nicht möglich.

Dieser direkte Zugriff von außen auf die Felder einer Klasse gilt aber als ausgesprochen schlechter Programmierstil, da ähnliche Probleme drohen wie beim Einsatz von Klassensymbolen. Sie verlieren sehr schnell den Überblick, wann und wie Sie ein Feld verändert haben, und müssen unter Umständen mühselig nach Fehlern suchen. Außerdem widerspricht dieser Zugriff – wie Sie ja bereits wissen – dem Grundsatz, dass nur ein Objekt selbst direkt auf seine Attribute zugreifen sollte.

Hinweis:

Der direkte Zugriff aus der Methode `Main()` auf ein Feld der Klasse `Autoklasse` entspricht einem Zugriff von außen, da die Methode `Main()` nicht zu der Klasse gehört. Sie befindet sich zwar im selben Quelltext wie die Klasse `Autoklasse`, ist aber kein Teil der Klasse `Autoklasse`, sondern gehört zur Klasse `Program` für das eigentliche Programm.

Gewöhnen Sie sich also von vornherein an, auf die Felder einer Klasse nur über die Methoden der Klasse zuzugreifen. Wenn Sie den Wert eines Feldes außerhalb einer Methode der Klasse benötigen, erstellen Sie eine Methode für die Klasse, die Ihnen den benötigten Wert als Rückgabe liefert. Dieses Verfahren mag vielleicht am Anfang recht umständlich und überflüssig wirken, stellt aber vor allem bei komplexeren Programmen sicher, dass Sie die Fehlersuche auf eine Klasse beschränken können.

Eine Methode für unsere Klasse `Autoklasse`, die die aktuelle Geschwindigkeit zurückgibt, könnte zum Beispiel so aussehen:

```
public int GetGeschwindigkeit()
{
    return geschwindigkeit;
}
```

Die Ausgabe in `Main()` oder eine Zuweisung erfolgt dann wieder mit den gewohnten Techniken – also zum Beispiel

```
Console.WriteLine("Die Geschwindigkeit beträgt {0}.",
    auto1.GetGeschwindigkeit());
```

oder

```
wert = auto2.GetGeschwindigkeit();
```

Hinweis:

Methoden, die nur den Wert eines Feldes liefern, werden häufig durch den Zusatz `Get2` zu Beginn des Methodennamens gekennzeichnet. Danach folgt dann der Name des Feldes. Damit ist auf einen Blick klar, was die Methode macht.

Methoden, die nur den Wert eines Feldes setzen, erhalten dagegen oft die Kennzeichnung `Set3` zu Beginn des Methodennamens. Anschließend wird wieder der Name des Feldes angegeben.

Zum Abschluss dieses Kapitels wollen wir in einem Exkurs noch einen kurzen Blick auf die Speicherverwaltung von C# werfen und uns ansehen, was beim Operator `new` genau geschieht.

Exkurs: Die Speicherverwaltung von C#

Wie Sie ja bereits wissen, erzeugt der Operator `new` zur Laufzeit eines Programms ein neues Objekt im Speicher und liefert eine Referenz auf dieses Objekt zurück. Der Speicherbereich für dieses neue Objekt wird automatisch reserviert und steht damit nicht mehr für andere Objekte zur Verfügung.

Dieses Konzept der **dynamischen Speicherverwaltung** gibt es auch in vielen anderen Programmiersprachen – zum Beispiel in C++. Dort sind Sie als Programmierer allerdings für die Verwaltung des dynamisch reservierten Speichers im Wesentlichen selbst verantwortlich. Sie müssen also nicht nur Speicher reservieren, sondern den reservierten Speicher auch selbst wieder freigeben. Und diese manuelle Speicherverwaltung hat einige Tücken. Wenn Sie zum Beispiel vergessen, einen reservierten Speicherbereich wieder freizugeben, erzeugen Sie „Speicherleichen“, die unnötig Platz verschwenden. Geben Sie dagegen Speicher voreilig wieder frei, drohen Abstürze, wenn Sie später doch wieder auf den vermeintlich reservierten Bereich zugreifen wollen. Solche Fehler sind – gerade zu Beginn – zum Leidwesen vieler geplagter C++-Programmierer alles andere als selten.

Außerdem entstehen durch das manuelle Reservieren und Freigeben schnell „Löcher“ im Speicher. Wenn Sie beispielsweise hintereinander mehrere Speicherbereiche reservieren und dann den zuerst reservierten Bereich wieder freigeben, wird der Speicher nicht automatisch reorganisiert. Das heißt, irgendwann haben Sie unter Umständen zwar noch viel freien Platz im Speicher, der sich aber auf sehr viele kleine Bereiche verteilt, die nicht zusammenhängen. Einen größeren Speicherbereich können Sie dann nicht mehr reservieren, da Sie einen zusammenhängenden Block benötigen.

Als C#-Programmierer können Sie sich jetzt ganz entspannt zurücklehnen. Denn: Sie haben mit der gesamten Speicherverwaltung eigentlich nichts tun. Sie müssen lediglich Speicher reservieren, den Rest erledigt dann C# über die **Garbage Collection^{a)}**.

a) Wörtlich übersetzt bedeutet *garbage collection* „Müllabfuhr“.

Die Garbage Collection „entrümpelt“ quasi den Speicher. Dabei werden nicht mehr benötigte Speicherbereiche freigegeben und der Speicher reorganisiert.



Der Prozess, der die „Entrümpelung“ durchführt, heißt **Garbage Collector**. Er wird automatisch gestartet.

2. `get` bedeutet übersetzt so viel wie „hole“.
3. `set` bedeutet übersetzt so viel wie „setze“.

Zusammenfassung

Die Vereinbarung einer Klasse wird durch das Schlüsselwort `class` eingeleitet. Danach folgen der Bezeichner der Klasse sowie in geschweiften Klammern die Attribute – die Felder – und die Methoden der Klasse.

Felder und Instanzmethoden werden immer ohne das Schlüsselwort `static` vereinbart.

Um eine Instanz zu erzeugen, legen Sie zunächst eine Variable mit dem Typ der Klasse an. Das eigentliche Erzeugen der Instanz erfolgt mit dem Operator `new`, dem Namen der Klasse und den runden Klammern `()`.

Der Zugriff auf die Methoden erfolgt durch den Namen der Instanz, einen trennenden Punkt und den Namen der Methode.

Einen direkten Zugriff von außen auf Felder sollten Sie nach Möglichkeit vermeiden.

Aufgaben zur Selbstüberprüfung

- 2.1 Sie haben eine Klasse `Auto` vereinbart. Geben Sie bitte die Anweisungen an, mit denen Sie eine Instanz `wagen` für diese Klasse erzeugen.

- 2.2 Sie haben eine Instanz `wagen` für eine Klasse `Auto` erstellt. Rufen Sie die Methode `Lenken()` für diese Instanz auf. Übergeben Sie dabei einen beliebigen `int`-Wert als Argument.

- 2.3 Sie haben eine Instanz `wagen` für eine Klasse `Auto` erstellt. Was unterscheidet die beiden Anweisungen `wagen.Stoppen()` und `wagen.Stoppen?`

2.4 Welche Aufgabe übernimmt der Garbage Collector von C#?

3 Die verschiedenen Methoden und Variablen unter der Lupe

In diesem Kapitel werden wir uns noch einmal die Unterschiede zwischen Klassenmethoden und Instanzmethoden sowie Klassenvariablen, Feldern beziehungsweise Instanzvariablen und lokalen Variablen ansehen.

3.1 Was Klassenmethoden und Instanzmethoden unterscheidet

Wenn Sie noch einmal an die Beispiele für Klassenmethoden zurückdenken, sollte Ihnen mit Ihren neuen Kenntnissen zur objektorientierten Programmierung ein Punkt auffallen:

Wir haben dort zwar eine Klasse vereinbart – nämlich für das eigentliche Programm –, aber nie eine Instanz dieser Klasse erzeugt. Trotzdem konnten wir die Klassenmethoden ohne Probleme aufrufen.

Beim Code mit den beiden Autos dagegen klappt der Aufruf ohne die Instanz nicht. Bei einer Anweisung

```
Initialisiere();
```

in der Methode `Main()` beschwert sich der Compiler sofort, dass er die Methode nicht kennt. Das liegt zum einen daran, dass diese Methode zur Klasse `Autoklasse` gehört, zum anderen aber auch daran, dass es sich um eine Instanzmethode handelt – also eine Methode, die sich ohne eine Instanz einer Klasse gar nicht aufrufen lässt.

Auch der gut gemeinte Versuch

```
Autoklasse.Initialisiere();
```

wird vom Compiler mit einer Fehlermeldung quittiert. Sie lautet

`Für das nicht statische Feld, die Methode oder die Eigenschaft "Autoklasse.Initialisiere(int)" ist ein Objektverweis erforderlich.`

Wenn Sie diese Meldung sinngemäß umdrehen – also „Für statische Felder ist *kein* Objektverweis erforderlich“, kommen Sie auch schnell hinter das Geheimnis, warum wir unsere Klassenmethoden ohne Instanz aufrufen konnten. Denn Klassenmethoden gehören nicht zu einer Instanz oder einem konkreten Objekt, sondern nur zu der Klasse. Sie lassen sich daher auch nur über den vorangestellten Namen der Klasse beziehungsweise einfach nur über den Namen der Methode aufrufen. Der Aufruf nur über den Namen der Methode ohne den vorangestellten Klassennamen funktioniert allerdings nur dann, wenn die Klassenmethode in der Klasse vereinbart wurde, in der auch der Aufruf erfolgt. Das war in all unseren Beispielen bisher der Fall.



Klassenmethoden gehören nicht zur Instanz einer Klasse, sondern nur zu der Klasse an sich.

Die Methoden unserer Klasse `Autoklasse` sind aber keine Klassenmethoden, da das Schlüsselwort `static` in der Vereinbarung fehlt. Daher funktioniert auch der Aufruf über den Namen der Klasse nicht.

Um sich das Erzeugen der Instanzen zu ersparen, könnten Sie jetzt kurzerhand alle Methoden über das Schlüsselwort `static` zu Klassenmethoden machen. Dann wird sich aber der Compiler sofort beschweren, dass für das Feld `geschwindigkeit` ein Objektverweis erforderlich ist.

Sie können mit Klassenmethoden nur auf Klassenvariablen beziehungsweise auf lokale Variablen zugreifen und nicht auf Felder.



Und was passiert, wenn Sie das Feld `geschwindigkeit` in unserem Code zu einer Klassenvariablen machen, sehen wir uns jetzt an.

3.2 Was Klassenvariablen und Felder unterscheidet

Ergänzen Sie im Code 2.1 mit den beiden Autos bitte das Schlüsselwort `static` vor der Vereinbarung des Feldes `geschwindigkeit`. Die geänderte Zeile sollte dann so aussehen:

```
class Autoklasse
{
    static int geschwindigkeit;
    ...
}
```

Code 3.1: Aus dem Feld wird eine Klassenvariable

Damit wird das Feld zur Klassenvariablen.

Hinweis:

Sie können aus einer „normalen“ Methode ohne Weiteres auf eine Klassenvariable zugreifen. Der Zugriff aus einer Klassenmethode auf ein Feld dagegen ist nicht möglich.

Den geänderten Code finden Sie auch im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **Cshp05d_03_01**.

Wenn Sie das Programm dann ausführen lassen, werden die Auswirkungen dieser „kleinen“ Änderung sofort deutlich. Für beide Autos – also für beide Instanzen – werden immer dieselben Werte ausgegeben. Denn die Instanzen der Klasse `Autoklasse` teilen sich jetzt die Klassenvariable `geschwindigkeit`. Es gibt also nicht mehr für jede Instanz eine eigene Variable `geschwindigkeit`, sondern nur noch eine einzige Variable `geschwindigkeit`, auf die sämtliche Instanzen zugreifen. Das heißt, eine Änderung, die eine Instanz an der Klassenvariablen durchführt, wirkt sich auch auf alle anderen Instanzen aus.

Schauen wir uns das einmal im Detail an.

Zuerst wird über die Methode `Initialisiere()` die Geschwindigkeit für die beiden Autos gesetzt. Der erste Aufruf setzt den Wert auf 0, der zweite dagegen auf 10. Da diese Änderung beide Instanzen betrifft, erscheint auch für beide Instanzen bei der folgenden Ausgabe der Wert 10 auf dem Bildschirm. Danach ändern wir die Geschwindigkeit über

die Methode `Gasgeben()`. Beim ersten Aufruf ändert sich der Wert der Klassenvariablen in 30, beim nächsten Aufruf werden noch einmal 100 addiert. Ausgegeben wird daher der Wert 130 für die Geschwindigkeit.

Noch besser sichtbar wird der gemeinsame Zugriff der beiden Instanzen auf die Klassenvariable, wenn Sie die Werte im Quelltext nach jeder Änderung ausgeben lassen.

Brauchbar sind solche Effekte in der Regel natürlich nicht. Denn wenn Sie auf der Autobahn beschleunigen, werden die anderen Autos ja nicht ebenfalls automatisch schneller.

Trotzdem haben Klassenvariablen und auch Klassenmethoden durchaus ihre Berechtigung. Denn es gibt viele Fälle, in denen Sie überhaupt keine Instanz einer Klasse benötigen – zum Beispiel für mathematische Berechnungen, für das Einlesen von Daten oder auch für das Umformen einer Zeichenkette in einen numerischen Wert. Hier interessiert in der Regel nur das Ergebnis an sich – unabhängig von einer Instanz.

Solche Klassenmethoden haben Sie ja auch schon reichlich in diesem Lehrgang eingesetzt – zum Beispiel bei `Convert.ToInt32()` oder auch bei `Console.ReadLine()`. Vor dem Punkt finden Sie dabei jeweils den Namen der Klasse und hinter dem Punkt den Namen der Klassenmethode.

Eine selbst erstellte Klasse, die über eine Klassenvariable und eine Klassenmethode das Quadrat einer Zahl errechnet, könnte zum Beispiel so aussehen:

```
/*#####
Eine Klasse mit Klassenmethode
und Klassenvariable
#####*/
using System;

namespace Cshp05d_03_02
{
    //die selbst erstellte Klasse
    class Quadrat
    {
        //eine Klassenvariable
        static int ergebnis;

        //eine Klassenmethode
        public static int Berechnen(int wert)
        {
            ergebnis = wert * wert;
            return ergebnis;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //der Aufruf der Klassenmethode aus Quadrat
            //bitte in einer Zeile eingeben
            Console.WriteLine("Das Quadrat von 4 ist {0}",
```

```
        Quadrat.Berechnen(4));  
    }  
}
```

Code 3.2: Eine selbst erstellte Klasse mit Klassenvariable und Klassenmethode

Da hier nur das Ergebnis der Berechnung interessiert und innerhalb der Klasse Quadrat nicht weiterverarbeitet werden muss, könnten Sie auf die Klassenvariable ergebnis auch komplett verzichten und die Klassenmethode direkt den Ausdruck wert * wert zurückliefern lassen.

Außerdem brauchen Sie Klassenmethoden, um eine C#-Anwendung zu starten. Denn wenn Sie das Schlüsselwort `static` bei der Vereinbarung der Methode `Main()` entfernen, machen Sie die Methode `Main()` zu einer Instanzmethode – und damit müssten Sie erst im Programm selbst eine Instanz der Klasse für das Programm erzeugen, um das Programm starten zu können. Und das klappt nicht, da sich das Programm ja gar nicht starten lässt.

Probieren Sie das einfach einmal aus. Entfernen Sie das Schlüsselwort `static` in der Vereinbarung der Methode `Main()` und lassen Sie das Programm dann noch einmal ausführen. Sie werden sehen, es erscheint eine Fehlermeldung, dass keine als Einstiegs-
punkt geeignete Methode `Main()` gefunden wird.

Schauen wir uns nun noch ein Beispiel an, wie Sie Felder und Klassenvariablen kombinieren können. Im folgenden Code wird über eine Klassenvariable `autoZaehler` mitgezählt, wie viele Instanzen der Klasse `Autoklasse` erzeugt wurden. Die Anzahl der Autos wird dabei über die Klassenmethode `GetAutoZaehler()` zurückgeliefert. Damit Sie die geänderten Stellen schneller wiederfinden, haben wir sie fett markiert.

```
/*#####
Kombination von Klassenvariablen und Feldern
#####*/
using System;

namespace Cshp05d_03_03
{
    //die Vereinbarung der Klasse Autoklasse
    //Sie erfolgt außerhalb der Klasse für das Programm
    class Autoklasse
    {
        //die Klassenvariable
        static int autoZaehler;
        //das Feld
        int geschwindigkeit;

        //die Methoden
        //eine Klassenmethode, die die Anzahl der Instanzen liefert
        public static int GetAutoZaehler()
        {
            return autoZaehler;
        }
    }
}
```

```

//die "echten" Methoden
//zum Initialisieren
//hier wird jetzt auch die Klassenvariable
//autoZaehler um den Wert 1 erhöht
public void Initialisiere(int standard)
{
    geschwindigkeit = standard;
    autoZaehler++;
}
...
}

class Program
{
    static void Main(string[] args)
    {
        //zwei neue Instanzen für Autoklasse erzeugen
        Autoklasse auto1 = new Autoklasse();
        Autoklasse auto2 = new Autoklasse();

        //die Methode Initialisiere() für die beiden
        //Autos aufrufen
        auto1.Initialisiere(0);
        auto2.Initialisiere(10);

        //die Anzahl der Autos über die Klassenmethode ausgeben
        //bitte in einer Zeile eingeben
        Console.WriteLine("Es gibt {0} Autos.",
        Autoklasse.GetAutoZaehler());
        ...
    }
}

```

Code 3.3: Eine Klassenvariable zum Zählen der Instanzen

Hinweis:

Einige Anweisungen, die sich nicht geändert haben, sind nicht noch einmal abgedruckt, sondern durch ... gekennzeichnet.

Das Programm zählt nur dann korrekt die Instanzen mit, wenn Sie für jede neue Instanz auch die Methode `Initialisiere()` aufrufen.

3.3 Das Problem der Überdeckung

Wie Sie bereits gelernt haben, überdecken lokale Variablen Klassenvariablen mit demselben Namen. Dasselbe gilt auch für Felder. Sie werden ebenfalls von lokalen Variablen mit demselben Namen überdeckt.

Schauen Sie sich dazu einmal den folgenden Code an. Er soll den Wert eines Feldes `zKette` vom Typ `string` setzen. Auf den ersten Blick sieht er bei flüchtigem Hinsehen durchaus brauchbar aus.

```

#####
Das Problem der Überdeckung
#####

using System;

namespace Cshp05d_03_04
{
    class Ueberdeckt
    {
        //das Feld
        string zeichenkette;

        //die Methode soll den Wert von zeichenkette ändern
        public void ZeichenketteAendern(string zeichenkette)
        {
            zeichenkette = zeichenkette + " geändert";
        }

        //die Zeichenkette liefern
        public string GetZeichenkette()
        {
            return zeichenkette;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //eine Instanz von Ueberdeckt erzeugen
            Ueberdeckt test = new Ueberdeckt();
            test.ZeichenketteAendern("Test");
            //das führt zu seltsamen Ausgaben
            //bitte in einer Zeile eingeben
            Console.WriteLine("Der geänderte Wert ist
{0}.", test.GetZeichenkette());
            //und das ebenfalls
            if (test.GetZeichenkette() == String.Empty)
                Console.WriteLine("Die Zeichenkette ist leer");
            else
                Console.WriteLine("Die Zeichenkette ist nicht leer");
        }
    }
}

```

Code 3.4: Ein überdecktes Feld

Beim Ausführen liefert das Programm allerdings seltsame Ergebnisse. Denn in der Methode `ZeichenketteAendern()` wird nicht – wie beabsichtigt – der Wert des **Feldes** `zeichenkette` geändert, sondern der Wert der **lokalen Variable** `zeichenkette`, die als Parameter an die Methode `ZeichenketteAendern()` übergeben wird. Außerdem wird dagegen auf den Wert des Feldes

zeichenkette zugegriffen – und der ist `null`, da ja keinerlei Wertzuweisung erfolgt ist. Darauf weist Sie Visual Studio auch mit einer entsprechenden Warnung hin, wenn Sie das Programm übersetzen lassen.

Die Ausgabeanweisung in dem Beispiel liefert gar nichts und der Vergleich ergibt, dass die Zeichenkette nicht leer ist – obwohl nichts ausgegeben wird. Denn der Inhalt der Zeichenkette ist `null` und dieser Wert wird schlicht und einfach nur nicht ausgegeben.

Um solche Probleme zu umgehen, können Sie die `this`⁴-Referenz benutzen. Sie ermöglicht den Zugriff auf ein Feld, das eigentlich durch eine lokale Variable überdeckt wird. Dazu geben Sie das Schlüsselwort `this`, einen Punkt und den Bezeichner für das Feld an.



Die `this`-Referenz bezieht sich immer auf die Instanz einer Klasse, die eine Methode aufgerufen hat.

In unserem Beispiel würde der geänderte Zugriff dann so aussehen:

```
//die Methode soll den Wert von zeichenkette ändern
public void ZeichenketteAndern(string zeichenkette)
{
    this.zeichenkette = zeichenkette + " geändert";
}
```

Code 3.5: Der geänderte Zugriff auf das Feld

Damit wird in der Methode jetzt der Wert des Feldes `zeichenkette` auf den Wert der lokalen Variablen `zeichenkette` gesetzt und noch das Wort „geändert“ angehängt.

Ändern Sie den Code 3.4 einmal entsprechend und testen Sie ihn dann noch einmal. So sollte er auch funktionieren.



Bitte beachten Sie:

Der Zugriff über die `this`-Referenz ist nur aus einer „normalen“ Methode – also aus einer Instanzmethode – möglich. In einer Klassenmethode können Sie diese Art des Zugriffs nicht verwenden, da ja überhaupt keine Instanz existiert.

Wenn Sie aus einer Klassenmethode auf eine verdeckte Klassenvariable zugreifen wollen, können Sie aber statt der `this`-Referenz einfach den Namen der Klasse verwenden.

Neben dem Einsatz der `this`-Referenz gibt es aber noch eine andere Möglichkeit, das Problem der Überdeckung zu umgehen: Verwenden Sie unterschiedliche Namen für lokale Variablen und Felder. Denn beim Einsatz der `this`-Referenz und identischen Namen drohen schwer zu findende Fehler, wenn Sie nicht genau aufpassen.

Im folgenden Fragment soll zum Beispiel der Wert eines Feldes `zKette` auf den Wert einer lokalen Variablen `zKette1` gesetzt werden.

4. `this` bedeutet übersetzt so viel wie „diese“ oder „die“.

```
//die Methode soll den Wert von zKette ändern
public void zKetteAendern(string zKette1)
{
    this.zKette = zKette;
}
```

Code 3.6: Drohende Verwirrung bei ähnlichen Namen

Auf der rechten Seite der Zuweisung fehlt aber die 1 am Ende des Bezeichners. Damit würden Sie dem Feld `zKette` seinen eigenen Wert zuweisen. Diese Anweisung ist zwar unsinnig, aber syntaktisch völlig korrekt. Das Programm würde daher ausgeführt. Allerdings weist Sie Visual Studio mit einer Warnung auf die drohenden Probleme hin.

Zusammenfassung

Klassenmethoden gehören nicht zu einer Instanz, sondern zur Klasse an sich. Sie lassen sich daher ohne Instanz aufrufen.

In Klassenmethoden können Sie nur auf Klassenvariablen beziehungsweise auf lokale Variablen zugreifen.

Felder und Klassenvariablen werden von lokalen Variablen mit demselben Namen überdeckt.

Über die `this`-Referenz können Sie auf ein Feld zugreifen, das eigentlich durch eine lokale Variable überdeckt wird. Um auf eine verdeckte Klassenvariable zuzugreifen, verwenden Sie den Namen der Klasse.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie haben in einer Klasse `Auto` eine Klassenvariable `geschwindigkeit` vereinbart. Welche Besonderheit müssen Sie in unterschiedlichen Instanzen der Klasse `Auto` bei der Klassenvariablen beachten?

- 3.2 Bitte sehen Sie sich das folgende Quelltextfragment an. Hier soll in der Methode `Test()` auf ein Feld `versuch` zugegriffen werden. Ist dieser Zugriff möglich? Begründen Sie kurz Ihre Antwort.

```
public static void Test()  
{  
    versuch = 10;  
    ...  
}
```

- 3.3 Sie haben in einer Klasse ein Feld `versuch` vereinbart. Formulieren Sie eine Anweisung, mit der Sie diesem Feld in einer Methode den Wert einer gleichnamigen lokalen Variablen zuweisen können.

4 Zusammenarbeit zwischen Methoden

Nach Ihren ersten Schritten in der objektorientierten Programmierung mit C# wollen wir uns in diesem Kapitel ein etwas komplexeres Beispiel ansehen.

Nehmen wir einmal folgendes Szenario: Zwei Wasserbehälter, die jeweils 100 Liter Wasser fassen können, sind über Pumpen und Rohre miteinander verbunden. Zwischen den beiden Behältern soll beliebig Wasser hin- und hergepumpt werden können, und zwar so lange, bis entweder der eine Behälter leer ist oder der andere voll. Beide Behälter sollen zu Beginn jeweils 60 Liter Wasser enthalten.

Dieses Szenario wollen wir jetzt mit Objekten in C# nachbilden. Dazu erst einmal ein paar theoretische Vorüberlegungen:

- Wir benötigen zwei Objekte – für jeden Behälter eins. Beide Objekte bilden wir als Instanzen einer und derselben Klasse ab.
- Jedes Objekt benötigt als Attribut mindestens den aktuellen Füllstand.
- Die Pumpen, die das Wasser zwischen den Behältern hin- und hertransportieren, integrieren wir als Methode in die Objekte für die Behälter.
- Den Behälter, in den das Wasser gepumpt werden soll, geben wir über ein zusätzliches Attribut bei den Objekten für die Behälter an. Dieses Attribut ist eine Referenz, die auf den jeweils anderen Behälter positioniert ist.

Hinweis:

Diese Lösung ist ein Ansatz von mehreren Möglichkeiten. Sie könnten die Pumpen auch als eigenes Objekt erstellen oder das Umpumpen ohne Objekt umsetzen – zum Beispiel durch eine Klassenmethode. Wenn Sie Lust haben, können Sie ja einmal selbstständig versuchen, einen anderen Lösungsansatz zu programmieren.

Den vollständigen Quelltext für die Lösung der Aufgabe finden Sie im folgenden Code. Die Besonderheiten erklären wir Ihnen wie gewohnt im Anschluss.

```
/*#####
Behälter
#####*/
using System;
namespace Cshp05d_04_01
{
    //die Klasse für die Behälter
    class Behaelter
    {
        //die Felder
        int fuellstand;
        Behaelter andererBehaelter;

        //die Methoden
        public void Init(int menge)
        {

```

```
        fuellstand = menge;
    }

    public void VerbindenMit(Behaelter behaelter)
    {
        andererBehaelter = behaelter;
    }

    public int GetFuellstand()
    {
        return fuellstand;
    }

    public int Aufnehmen(int menge)
    {
        int rueckgabe;
        if (menge + fuellstand > 100)
        {
            rueckgabe = 100 - fuellstand;
            fuellstand = 100;
        }
        else
        {
            fuellstand = fuellstand + menge;
            rueckgabe = menge;
        }
        return rueckgabe;
    }

    public void Abgeben(int menge)
    {
        int gepumpt;
        if (menge > fuellstand)
            gepunkt = andererBehaelter.Aufnehmen(fuellstand);
        else
            gepunkt = andererBehaelter.Aufnehmen(menge);
        fuellstand = fuellstand - gepunkt;
    }
}

class Program
{
    //Ausgabe() ist eine Klassenmethode für die Klasse
    //des Programms
    //bitte in einer Zeile eingeben
    static void Ausgabe(Behaelter behaelter1, Behaelter
        behaelter2)
    {
        //bitte in einer Zeile eingeben
        Console.WriteLine("Behälter 1 -----
        Behälter 2");
        Console.Write(behaelter1.GetFuellstand());
        Console.Write(" ----- ");
        Console.WriteLine(behaelter2.GetFuellstand());
    }
}
```

```

static void Main(string[] args)
{
    //die Instanzen erzeugen und initialisieren
    Behaelter behaelter1 = new Behaelter();
    Behaelter behaelter2 = new Behaelter();

    behaelter1.Init(60);
    behaelter2.Init(60);

    //die "Leitungen" zwischen den Behältern legen
    behaelter1.VerbindenMit(behaelter2);
    behaelter2.VerbindenMit(behaelter1);

    //Ausgabe der Füllstände über die Klassenmethode
    Ausgabe(behaelter1, behaelter2);

    //10 Liter aus Behälter 1 umpumpen
    behaelter1.Abgeben(10);
    Ausgabe(behaelter1, behaelter2);

    //mehr umpumpen als passt
    behaelter1.Abgeben(40);
    Ausgabe(behaelter1, behaelter2);

    //60 Liter aus Behälter 2 umpumpen
    behaelter2.Abgeben(60);
    Ausgabe(behaelter1, behaelter2);
}
}

```

Code 4.1: Eine Pumpstation mit Objekten

Die erste Besonderheit gibt es bei den Feldern der Klasse Behälter. Hier vereinbaren wir neben dem Füllstand mit

Behaelter andererBehaelter;

ein Feld `andererBehaelter` vom Typ `Behaelter` – also ein Feld vom Typ der Klasse. Dieses Feld verwenden wir später, um den Behälter anzusprechen, in den das Wasser laufen soll.

Die nächste Besonderheit gibt es in der Methode `VerbindenMit()`. Hier stellen wir die Verbindung zwischen den beiden Behältern her. Dazu wird die Methode jeweils mit der Referenz des anderen Behälters als Argument aufgerufen. Das übernehmen die beiden folgenden Anweisungen in der Methode `Main()`

```
behaelter1.VerbindenMit (behaelter2);  
behaelter2.VerbindenMit (behaelter1);
```

In der Methode selbst wird die Verbindung dann durch die Anweisung

```
andererBehaelter = behaelter;
```

hergestellt. Hier wird das Feld `andererBehaelter` auf den Wert des Arguments `behaelter` gesetzt – also auf die Referenz des jeweils anderen Behälters.

Wenn Sie die Anweisungen etwas verwirren ...

Denken Sie bitte daran, dass es **zwei** Instanzen der Klasse `Behaelter` gibt. Damit gibt es auch zwei strikt voneinander getrennte Felder `andererBehaelter`. In der Instanz `behaelter1` verweist das Feld `andererBehaelter` auf die Instanz `behaelter2` und in der Instanz `behaelter2` auf die Instanz `behaelter1`.

Unsere Methode `VerbindenMit()` stellt aber nur die Verbindung in eine Richtung her. Nach dem Verbinden des ersten Behälters mit dem zweiten Behälter besteht nicht automatisch auch eine Verbindung vom zweiten Behälter zum ersten Behälter. Die Verbindung ist also so etwas wie eine „Einbahnstraße“. Daher müssen wir die Methode `VerbindenMit()` auch für jede Instanz einzeln aufrufen und dabei die Referenz auf die jeweils andere Instanz übergeben.

Richtig interessant sind dann die beiden Methoden `Abgeben()` und `Aufnehmen()`, die die Pumpen nachbilden. Denn die Methode `Abgeben()` benutzt die Methode `Aufnehmen()` des anderen Behälters. Schauen wir uns erst einmal die Methode `Abgeben()` im Detail an:

```
public void Abgeben(int menge)
{
    int gepumpt;
    if (menge > fuellstand)
        gepumpt = andererBehaelter.Aufnehmen(fuellstand);
    else
        gepumpt = andererBehaelter.Aufnehmen(menge);
    fuellstand = fuellstand - gepumpt;
}
```

Als Parameter wird die Menge übergeben, die in den anderen Behälter gepumpt werden soll. Mit der `if`-Anweisung wird dann zuerst überprüft, ob mehr abgepumpt werden soll, als überhaupt im Behälter vorhanden ist. Wenn die abzupumpende Menge größer ist als der Füllstand, wird die Methode `Aufnehmen()` des anderen Behälters mit dem aktuellen Wert des Feldes `fuellstand` aufgerufen. Hier wird dann also der gesamte vorhandene Inhalt umgepumpt. Wenn die abzupumpende Menge kleiner oder gleich dem aktuellen Füllstand ist, wird nur die angegebene Menge in den anderen Behälter gepumpt. Dazu rufen wir die Methode `Aufnehmen()` des anderen Behälters mit `menge` als Argument auf.



In beiden Fällen wird der jeweils andere Behälter über eine Referenz im Feld `andererBehaelter` angesprochen. Diese Referenz haben wir ja mit der Methode `VerbindenMit()` gesetzt.

In der Methode `Aufnehmen()` des anderen Behälters wird zunächst überprüft, ob der Behälter die gewünschte Menge überhaupt noch aufnehmen kann. Wenn das nicht der Fall ist, wird zunächst die maximal mögliche Menge ermittelt und der Füllstand auf 100 gesetzt. Das erledigen die beiden Anweisungen

```
rueckgabe = 100 - fuellstand;
fuellstand = 100;
```

Kann die gewünschte Menge dagegen in den Behälter gepumpt werden, wird der Füllstand aktualisiert und die Variable `rueckgabe` auf die umgepumpte Menge gesetzt.

Abschließend wird der Wert von `rueckgabe` an die Methode `Abgeben()` in der anderen Instanz zurückgeliefert – also an den anderen Behälter. In dieser Methode wird dann auch die umgepumpte Menge aus dem Behälter entfernt. Das übernimmt die letzte Anweisung

```
fuellstand = fuellstand - gepumpt;
```

Wie die beiden Methoden genau zusammenarbeiten, sehen wir uns gleich an einem Durchlauf mit konkreten Werten an. Sie sollten sich hier zunächst merken, dass die Methode `Abgeben()` auf die Methode `Aufnehmen()` der anderen Instanz zugreift. Die Methode `Abgeben()` für `behaelter1` greift also auf die Methode `Aufnehmen()` für `behaelter2` zu.

Bei den beiden Methoden `Init()` und `GetFuellstand()` der Klasse `Behaelter` gibt es keine Besonderheiten. Die Methode `Init()` setzt das Feld `fuellstand` auf den Wert, der als Parameter übergeben wird, und die Methode `GetFuellstand()` gibt den aktuellen Inhalt des Behälters zurück.

Zusätzlich zu den Methoden der Klasse `Behaelter` haben wir noch eine Klassenmethode `Ausgabe()` für die Klasse `Program` erstellt. Diese Methode erzeugt eine tabellarische Ausgabe der Füllstände und ruft dazu die Methode `GetFuellstand()` der Klasse `Behaelter` auf. Damit wir in dieser Methode auf beide Behälter zugreifen können, übergeben wir die Referenzen der Behälter als Parameter.

Bitte beachten Sie:

`Ausgabe()` ist **keine** Instanzmethode der Klasse `Behaelter`, sondern eine Klassenmethode für die Klasse `Program` – also für die Klasse des Programms.



In der Methode `Main()` erzeugen wir dann zwei Instanzen der Klasse `Behaelter`. Anschließend füllen wir die Behälter mit jeweils 60 Litern, stellen die Verbindung her und lassen die aktuellen Füllstände über die Klassenmethode `Ausgabe()` anzeigen.

Danach wird dann zum ersten Mal mit der Anweisung

```
behaelter1.Abgeben(10);
```

gepumpt. Schauen uns einmal genau an, was dabei geschieht. Die Werte der verschiedenen Variablen und Felder zeigen wir Ihnen dabei jeweils in Klammern.

1. Die Methode `Abgeben()` für `behaelter1` wird mit dem Wert 10 für `menge` aufgerufen.
2. In der Methode `Abgeben()` wird der Wert von `menge` (10) mit dem Wert von `fuellstand` (60) für `behaelter1` verglichen. Den Wert 60 haben wir `fuellstand` in der Methode `Init()` zugewiesen.
3. Der Wert von `menge` (10) ist kleiner als `fuellstand` (60). Es wird also der `else`-Zweig ausgeführt.
4. In diesem Zweig wird die Methode `Aufnehmen()` für `andererBehaelter` (zeigt auf `behaelter2`) mit dem Argument `menge` (10) aufgerufen. Die Zuweisung von `andererBehaelter` auf `behaelter2` haben wir in der Methode `VerbindenMit()` mit der Anweisung
`behaelter1.VerbindenMit(b behaelter2);` vorgenommen.

5. In der Methode `Aufnehmen()` wird geprüft, ob `menge` (10) und der `fuellstand` von `behaelter2` (60, auf `behaelter2` verweist ja `andererBehaelter` beim Aufruf) zusammen mehr als 100 ergeben. Das Ergebnis der Addition ist kleiner als 100, daher wird der `else`-Zweig ausgeführt.
6. Im `else`-Zweig wird `menge` (10) auf den `fuellstand` (60) addiert. Der Wert von `fuellstand` ist danach 70. Dann wird die lokale Variable `rueckgabe` auf den Wert von `menge` (10) gesetzt.
7. Der Wert von `rueckgabe` (10) wird an die Methode `Abgeben()` zurückgeliefert. Die Methode `Abgeben()` arbeitet nach wie vor auf die Instanz `behaelter1`. An dieser Stelle ist also die umgepumpte Menge bereits im Behälter `behaelter2`, aber auch noch im Behälter `behaelter1`.
8. In der Methode `Abgeben()` wird dann der Wert, den die Methode `Aufnehmen()` zurückgegeben hat (10), von `fuellstand` (60) des Behälters `behaelter1` subtrahiert. Der Füllstand von `behaelter1` ist damit 50.

Der Behälter `behaelter1` enthält also nach dem Ausführen von `behaelter1.Abgeben(10)` 50 Liter und der Behälter `behaelter2` 70 Liter. Genau diese Werte liefert auch die Klassenmethode `Ausgabe()`.

Schauen wir uns jetzt auch noch schrittweise an, was geschieht, wenn wir mehr aus einem Behälter abpumpen wollen, als in den anderen Behälter passt. Das geschieht zum Beispiel beim Aufruf von

```
behaelter1.Abgeben(40);
```

1. Die Methode `Abgeben()` für `behaelter1` wird mit dem Wert 40 für `menge` aufgerufen.
2. In der Methode `Abgeben()` wird der Wert von `menge` (40) mit dem Wert von `fuellstand` (50) für `behaelter1` verglichen.
3. Der Wert von `menge` (40) ist kleiner als `fuellstand` (50). Es wird also der `else`-Zweig ausgeführt.
4. In diesem Zweig wird die Methode `Aufnehmen()` für `andererBehaelter` (verweist auf `behaelter2`) mit dem Argument `menge` (40) aufgerufen.
5. In der Methode `Aufnehmen()` wird geprüft, ob `menge` (40) und `fuellstand` von `behaelter2` (70) zusammen mehr als 100 ergeben. Das Ergebnis der Addition ist diesmal größer als 100, daher wird der `if`-Zweig ausgeführt.
6. Im `if`-Zweig erhält die lokale Variable `rueckgabe` den Wert von $100 - fuellstand$ (70), also 30. Dann wird `fuellstand` auf 100 gesetzt. Der Behälter ist damit voll.
7. Dann wird der Wert der lokalen Variablen `rueckgabe` (30) zurückgeliefert an die Methode `Abgeben()` von `behaelter1`.
8. In der Methode `Abgeben()` wird der Wert, den die Methode `Aufnehmen()` für `behaelter2` zurückgegeben hat (30), von `fuellstand` (50) des Behälters `behaelter1` subtrahiert. Der Füllstand von `behaelter1` ist damit 20.

Behälter 1 enthält dann also 20 Liter und Behälter 2 ist vollgepumpt mit 100 Litern.

Ganz wichtig beim Ablauf ist der Wechsel der Instanz in der Methode `Abgeben()` beim Aufruf von `Aufnehmen()`. Hier wird nicht die Instanz angesprochen, die beim Aufruf der Methode `Abgeben()` angegeben wird, sondern die Instanz, zu der wir die Leitung verlegt haben – also jeweils der andere Behälter. Die Verbindung erfolgt über die Variable `andererBehaelter` in den beiden Instanzen. In `behaelter1` verweist `andererBehaelter` auf `behaelter2` und in `behaelter2` auf `behaelter1`. Entsprechend werden dann in der Methode `Aufnehmen()` auch jeweils die Felder des anderen Behälters angesprochen.

Versuchen Sie jetzt einmal, selbst schrittweise nachzuvollziehen, was beim Aufruf der Methode

```
behaelter2.Abgeben(60);
```

geschieht. Notieren Sie dabei auch die aktuellen Werte des Feldes `andererBehaelter` und schreiben Sie auf, welche Instanz gerade bearbeitet wird. Sie werden sehen, das Programm wirkt auf den ersten Blick zwar etwas verzwickt, ist aber bei etwas genauerem Hinsehen doch recht einfach nachzuvollziehen.

Gönnen Sie sich dann erst einmal eine kleine Verschnaufpause, damit sich das bisher Gelernte ein wenig „setzen“ kann. Im nächsten Kapitel werden wir uns an die Umsetzung einer einfach verketteten Liste machen.

Zusammenfassung

Ein Objekt einer Klasse kann über ein entsprechendes Feld auch eine Referenz auf ein anderes Objekt derselben Klasse enthalten.

Zwei Instanzen ein und derselben Klasse existieren strikt voneinander getrennt.

Aufgabe zur Selbstüberprüfung

- 4.1 Vereinbaren Sie eine Klasse Beispiel, die neben einem Feld vom Typ `int` eine Referenz auf eine Instanz der Klasse selbst enthält. Für die Felder können Sie beliebige Namen verwenden.

5 Eine einfach verkettete Liste

In diesem Kapitel erstellen wir eine einfache verkettete Liste.

Bei vielen anspruchsvolleren Programmieraufgaben werden Sie häufig vor dem Problem stehen, dass Sie vor dem Ausführen des Programms nicht genau wissen, wie viele Daten verarbeitet werden sollen.

Beispiel 5.1:



Nehmen wir noch einmal die „Lagerverwaltung“. Hier war ja die Anzahl der Kisten im Lager automatisch durch die Größe des Arrays fest vorgegeben.

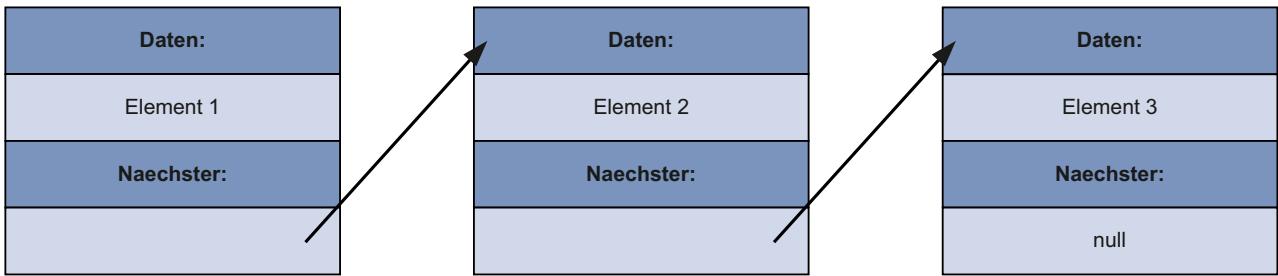
Setzen Sie die maximale Anzahl zum Beispiel auf 100, lassen sich höchstens 100 Einträge erfassen. Bei weiteren Einträgen löst das Programm eine Ausnahme aus, da die Grenzen des Arrays überschritten werden.

Jetzt können Sie natürlich hingehen und die obere Grenze auf einen Wert setzen, der wahrscheinlich nie erreicht wird – zum Beispiel 100 000. Damit verschwenden Sie aber wertvollen Speicher, wenn ein Anwender lediglich mit zehn Einträgen arbeitet. Der Speicherplatz für die restlichen 99 990 Einträge wird trotzdem reserviert und kann nicht für andere Zwecke genutzt werden. Außerdem laufen Sie auch bei dieser Methode Gefahr, dass möglicherweise ein Ausreißer nach oben eben doch mehr als 100 000 Einträge erfassen möchte.

Eine Alternative wäre es, die Daten nicht in einem Array abzuspeichern, sondern über eine Klasse. Für jeden neuen Gegenstand im Lager erzeugen Sie dann eine neue Instanz dieser Klasse. Damit wären dann zwar die Probleme mit der Begrenzung und auch mit der möglichen Speicherverschwendungen aus der Welt geschafft, allerdings „kennen“ sich die verschiedenen Einträge des Lagers nicht. Denn die Instanzen existieren ja isoliert voneinander und wissen nicht, wo sich die anderen Instanzen befinden. Damit wäre es zum Beispiel recht kompliziert, eine Liste mit sämtlichen Einträgen zu erstellen.

Eine weitere Alternative wäre es, durch eine geeignete Struktur dafür zu sorgen, dass zum einen beliebig viele Elemente angelegt werden können, zum anderen aber auch eine Verbindung zwischen den Elementen besteht. Dazu können Sie eine **einfach verkettete Liste** benutzen. Sie enthält beliebig viele gleichartige Elemente, die untereinander verbunden sind. Eine solche Liste werden wir in diesem Kapitel programmieren.

Das Grundprinzip einer einfach verketteten Liste ist eigentlich recht einfach. Neben den Daten wird in jedem Listenelement zusätzlich noch ein Verweis auf das folgende Element in der Liste gespeichert – zum Beispiel über eine Referenz. Grafisch lässt sich eine einfache verkettete Liste so darstellen:

**Abb. 5.1:** Eine einfach verkettete Liste

Naechster: unten in den Kästen steht dabei für die Referenz, die jeweils auf das nächste Element der Liste verweist. Beim letzten Element der Liste hat diese Referenz den Wert null. Damit wird angezeigt, dass kein weiteres Element mehr folgt.

5.1 Die Version mit Hilfskonstruktion

Schauen wir uns jetzt die einfach verkettete Liste als C#-Programm an. Wir werden dabei zunächst eine Version ohne „echte“ Objekte erstellen, damit Sie die grundsätzliche Funktion einer einfach verketteten Liste leichter nachvollziehen können. Diese „einfache“ Version bauen wir dann in einem zweiten Schritt in eine Version mit „echten“ Objekten um. Bei beiden Versionen benutzen wir zusätzlich Abbildungen, die den Zustand der einzelnen Elemente und Referenzen darstellen.

Hier also zunächst einmal die „einfache“ Version für eine einfach verkettete Liste, die Zeichenketten verarbeitet:

```

/*
#####
Eine verkettete Liste Version 1
#####
*/
using System;

namespace Cshp05d_05_01
{
    //eine Klasse für die Listenelemente
    //Sie enthält nur Felder für die Daten und den
    //Nachfolger, aber keine Methoden
    class Listenelement
    {
        public string Daten;
        public Listenelement Naechster;
    }

    class Program
    {
        //Klassenmethoden für die Verarbeitung der Liste

        //die Daten für ein Element setzen
        //übergeben werden die Zeichenkette und das Element
        //der Liste
        static void SetDaten(string datenNeu, Listenelement element)
    }
}
  
```

```

{
    //die Zeichenkette setzen
    element.Daten = datenNeu;
    //das Ende markieren
    element.Naechster = null;
}

//ein neues Element am Ende der Liste einfügen
//übergeben werden die Zeichenkette und der
//Listenanfang
//das eigentliche Einfügen erfolgt über die Methode
//SetDaten()
//bitte in einer Zeile eingeben
static void ListeAnhaengen(string datenNeu,
Listenelement listenAnfang)
{
    //eine Hilfskonstruktion zum Wandern in der Liste
    Listenelement hilfsKonstruktion;
    hilfsKonstruktion = listenAnfang;
    //durch die Liste gehen, bis das Ende erreicht ist
    while (hilfsKonstruktion.Naechster != null)
        hilfsKonstruktion = hilfsKonstruktion.Naechster;
    //neues Element am Ende der Liste einfügen
    hilfsKonstruktion.Naechster = new Listenelement();

    //Hilfskonstruktion auf das neue Element setzen
    hilfsKonstruktion = hilfsKonstruktion.Naechster;

    //die Daten eintragen
    SetDaten(datenNeu, hilfsKonstruktion);
}

//die Ausgabe der kompletten Liste
static void ListeAusgeben(Listenelement listenAnfang)
{
    //die Hilfskonstruktion
    Listenelement hilfsKonstruktion;
    hilfsKonstruktion = listenAnfang;
    //erstes Element ausgeben
    Console.WriteLine(hilfsKonstruktion.Daten);
    //und nun den Rest
    while (hilfsKonstruktion.Naechster != null)
    {
        hilfsKonstruktion = hilfsKonstruktion.Naechster;
        Console.WriteLine (hilfsKonstruktion.Daten);
    }
}

static void Main(string[] args)
{
    //ein neues Listenelement erzeugen
    Listenelement listenAnfang = new Listenelement();

    //die Daten im ersten Listenelement setzen
    SetDaten("Element 1", listenAnfang);
}

```

```
//weitere Elemente in einer Schleife anfügen
for (int element = 2; element < 4; element++)
    ListeAnhaengen("Element " + element, listenAnfang);

//die Liste ausgeben
ListeAusgeben(listenAnfang);
}
}
```

Code 5.1: Einfach verkettete Liste Version 1

Schauen wir uns die einzelnen Zeilen der Reihe nach an:

Zunächst einmal vereinbaren wir eine Klasse `Listenelement`, die lediglich die Daten für die Liste aufnehmen soll.

```
class Listenelement
{
    public string Daten;
    public Listenelement Naechster;
}
```

Neben dem Feld `Daten` für die Zeichenkette enthält die Klasse auch noch das Feld `Naechster`. Der Typ dieses Feldes entspricht dabei der Klasse. Das heißt also, eine Instanz der Klasse kann eine Referenz auf eine andere Instanz der Klasse speichern. Diese Referenz benutzen wir später, um die Position des Nachfolgelements in der Liste zu sichern.

Hinweis:

Eine sehr ähnliche Technik haben wir auch bereits im vorigen Kapitel bei der Pumpstation benutzt. Dort konnten wir aus einer Instanz ebenfalls über eine Referenz auf eine andere Instanz der Klasse zugreifen.

Wenn Sie nur Daten und keine Methoden benötigen, können Sie statt einer Klasse auch eine Struktur verwenden. In unserem Fall ist das allerdings nicht möglich, da eine Struktur keine Referenz auf sich selbst speichern kann.

In der Methode `Main()` erzeugen wir dann eine Instanz `listenAnfang` der Klasse `Listenelement`. Diese Instanz soll während der gesamten Laufzeit des Programms den Anfang der Liste – also das erste Element – speichern. Das ist zwingend erforderlich, damit wir den Anfang der Liste immer wiederfinden.

Anschließend erzeugen wir über den Aufruf der Klassenmethode `SetDaten()` das erste Element in der Liste. Als Argumente übergeben wir dabei die Zeichenkette, die in dem Element stehen soll, und das erste Element – in unserem Fall also den Listenanfang.

In der Methode `SetDaten()` selbst legen wir die Zeichenkette im Feld `Daten` ab und setzen den Wert des Nachfolgeelements – also das Feld `Naechster` – auf `null`.

Hinweis:

Das ausdrückliche Setzen des Feldes `Naechster` auf `null` ist eigentlich nicht erforderlich. Denn `null` wird auch als Standardwert für eine Variable für Objekte benutzt, wenn Sie selbst kein Objekt über `new` erzeugen und zuweisen. Und bei `Naechster` handelt es sich ja um eine Objektvariable. Wir setzen den Wert `null` aber trotzdem immer ausdrücklich, damit Sie leichter nachvollziehen können, was geschieht.

Die Situation nach dem Erzeugen des ersten Elements lässt sich grafisch so darstellen:

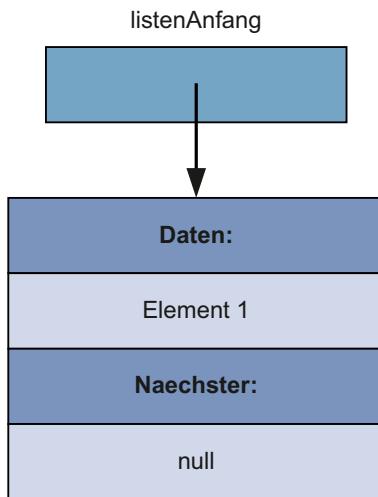


Abb. 5.2: Die Liste nach dem Erzeugen des ersten Elements

Das erste Element enthält bei den Daten die Zeichenkette „Element 1“, das Feld `Naechster` hat den Wert `null`. `listenerAnfang` zeigt auf das erste Element der Liste.

Nach dem Initialisieren der Liste fügen wir in `Main()` über eine Schleife weitere Elemente am Ende der Liste ein. Dazu benutzen wir die Klassenmethode `ListeAnhaengen()` und übergeben die Zeichenkette, die eingefügt werden soll, sowie `listenerAnfang` als Argumente. Die Zeichenkette „bauen“ wir dabei aus dem Text „Element“ und dem aktuellen Wert der Schleifenvariablen zusammen.

Da die Arbeitsweise der Klassenmethode `ListeAnhaengen()` etwas deutlicher wird, wenn sich mehrere Elemente in der Liste befinden, überspringen wir das erste Einfügen in der Schleife und schauen uns an, was beim zweiten Einfügen geschieht.

Vor dem zweiten Einfügen in der Schleife haben wir folgende Situation:

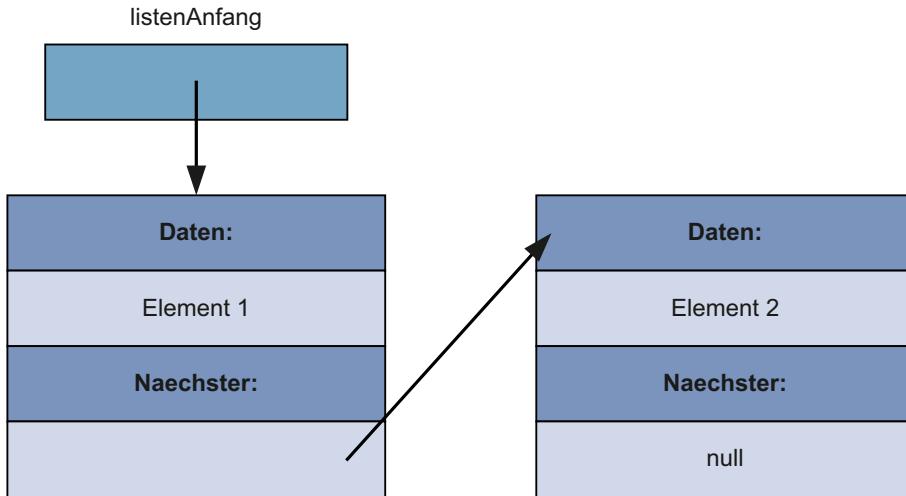


Abb. 5.3: Die Liste nach dem ersten Einfügen und vor dem zweiten Einfügen in der Schleife

`listenAnfang` steht nach wie vor auf dem ersten Element der Liste. Das Feld `Naechster` im ersten Listenelement verweist auf das zweite Listenelement, das wir im ersten Schleifendurchlauf eingefügt haben. In diesem zweiten Listenelement hat `Naechster` wieder den Wert `null`, um das Ende der Liste zu kennzeichnen.

In der Klassenmethode `ListeAnhaengen()` vereinbaren wir nun zuerst eine lokale Variable `hilfsKonstruktion`, die ebenfalls auf ein Listenelement verweisen kann. Diese Variable benutzen wir, um vom Anfang bis zum Ende der Liste wandern zu können.

Dazu setzen wir die Variable mit der Anweisung

```
hilfsKonstruktion = listenAnfang;
```

zunächst einmal auf den Anfang der Liste. Beide Variablen verweisen nach dieser Anweisung also auf das erste Element in der Liste.

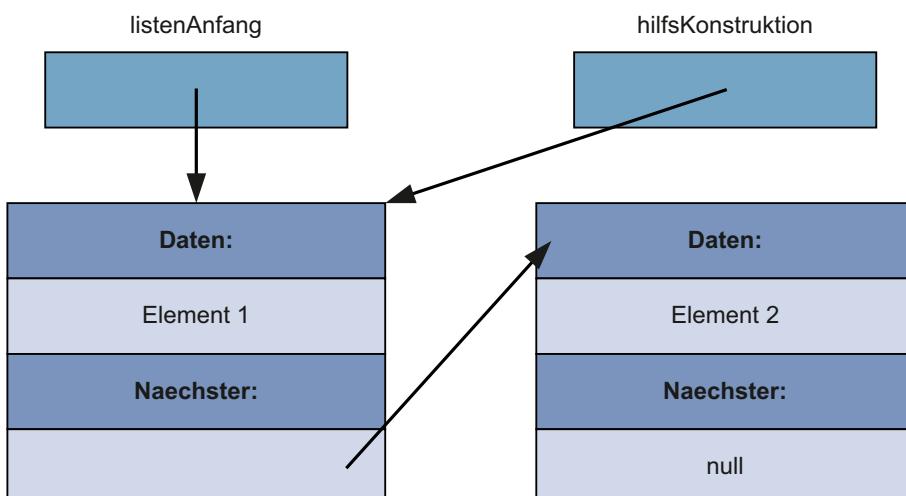


Abb. 5.4: Die Liste nach dem Setzen von `hilfsKonstruktion`

Nach dieser Operation folgt eine `while`-Schleife.

```
while (hilfsKonstruktion.Naechster != null)
    hilfsKonstruktion = hilfsKonstruktion.Naechster;
```

In der Schleife wird überprüft, ob das Feld `Naechster` des Elements, auf das die Hilfskonstruktion gerade verweist, ungleich `null` ist. Wenn das der Fall ist, wissen wir, dass es noch ein nachfolgendes Element gibt. Auf dieses nachfolgende Element setzen wir dann `hilfsKonstruktion`. Damit verweist also `hilfsKonstruktion` nicht mehr auf den Anfang der Liste, sondern auf das folgende Element. Und der Wert des Feldes `Naechster` dieses folgenden Elements wird dann auch bei der nächsten Prüfung in der Schleife benutzt.

Dieses Wandern durch die Liste wiederholen wir so lange, bis das Feld `Naechster` in einem Element den Wert `null` hat. Da `null` das Ende der Liste kennzeichnet, können wir das Durchsuchen beenden und am Ende der Liste ein neues Element einfügen. Das erfolgt über die Anweisung

```
hilfsKonstruktion.Naechster = new Listenelement();
```

Die Referenz auf das neue Element legen wir dabei im Feld `Naechster` des Elements ab, auf das die Hilfskonstruktion aktuell verweist – also im Feld `Naechster` des letzten Elements der Liste. In dem Feld befindet sich nach der Zuweisung dann nicht mehr der Wert `null`, sondern eine Referenz auf unser neues Element.

Unmittelbar nach dem Einfügen des neuen Listenelements haben wir dann folgende Situation:

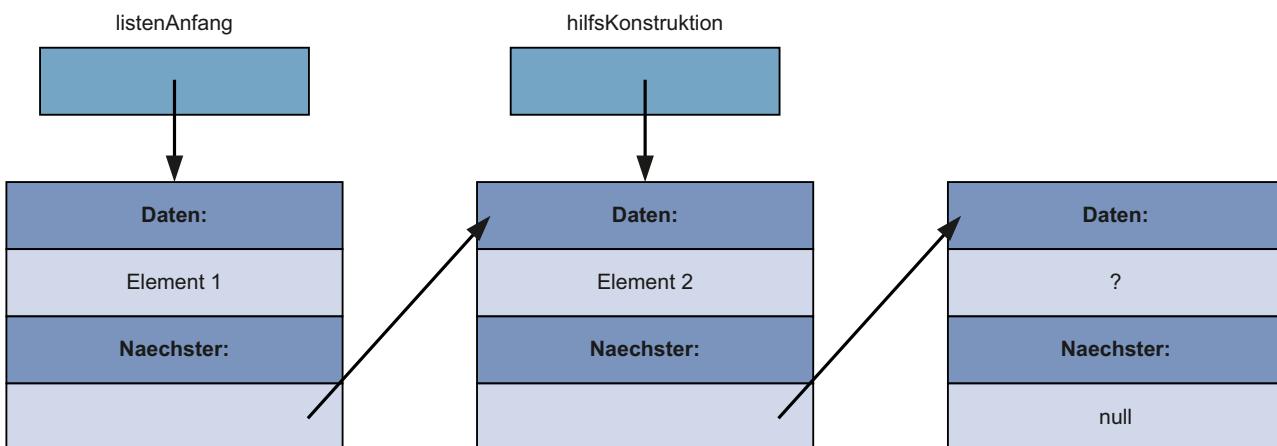


Abb. 5.5: Die Liste unmittelbar nach dem Einfügen des neuen Elements

Hinweis:

Der Wert `null` von `Naechster` im neuen Element wird automatisch beim Erzeugen des neuen Elements gesetzt. Der Wert von `Daten` ist hier noch nicht bekannt beziehungsweise ebenfalls `null`, da wir den Wert des Feldes ja noch nicht gesetzt haben.

In der folgenden Anweisung setzen wir die Hilfskonstruktion auf das neue Element in der Liste, damit wir direkt auf dieses Element zugreifen können. Die Referenz des neuen Elements steht ja in `hilfsKonstruktion.Naechster`. Wir können den Wert also einfach kopieren.

```
hilfsKonstruktion = hilfsKonstruktion.Naechster;
```

Der Zugriff auf das neue Element der Liste lässt sich auch ohne den Umweg mit dem Kopieren in die Hilfskonstruktion bewerkstelligen. Allerdings wird die Weitergabe an die Klassenmethode `SetDaten()` dann etwas komplizierter. Denn hier müssten Sie den Ausdruck `hilfsKonstruktion.Naechster` als Argument angeben, um die Referenz auf das neu erzeugte Listenelement weiterzureichen.

Da der Ablauf schon komplex genug ist, nehmen wir die zusätzliche Anweisung zum Kopieren der Hilfskonstruktion in Kauf, um den Zugriff etwas zu vereinfachen, und rufen die Klassenmethode `SetDaten()` so auf:

```
SetDaten(datenNeu, hilfsKonstruktion);
```

In der Methode `SetDaten()` werden dann die Felder für das Listenelement gesetzt, das wir über `hilfsKonstruktion` übergeben. Dabei gibt es keine Besonderheiten.

Nach dem Ausführen der Klassenmethode `SetDaten()` haben wir also die folgende Situation:

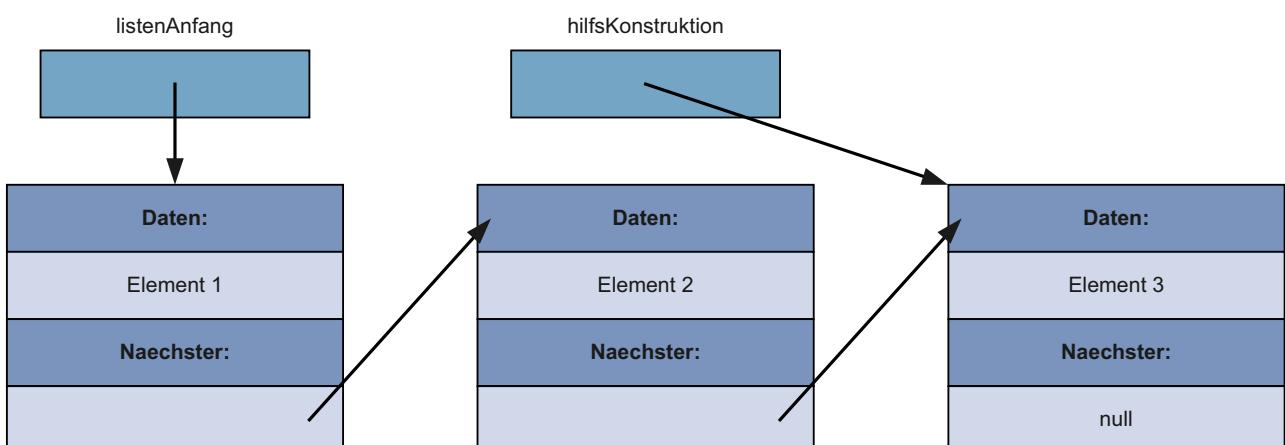


Abb. 5.6: Die Liste nach dem vollständigen Einfügen

Nach dem Einfügen der Elemente in die Liste erfolgt in `Main()` die Ausgabe über die Klassenmethode `ListeAusgeben()`. Die Arbeitsweise dieser Methode ist vergleichsweise einfach nachzuvollziehen.

Als Argument übergeben wir beim Aufruf `listenanfang` – also die Referenz auf das erste Element in der Liste. In der Methode setzen wir wieder eine Hilfskonstruktion auf den Anfang der Liste. Das übernehmen die beiden Anweisungen

```
Listenelement hilfsKonstruktion;
hilfsKonstruktion = listenanfang;
```

Dann erfolgt die Ausgabe der Daten für das Element, auf das `hilfsKonstruktion` gerade verweist:

```
Console.WriteLine(hilfsKonstruktion.Daten);
```

Anschließend lassen wir in einer `while`-Schleife die Daten aller weiteren Elemente ausgeben.

```
while (hilfsKonstruktion.Naechster != null)
{
    hilfsKonstruktion = hilfsKonstruktion.Naechster;
    Console.WriteLine(hilfsKonstruktion.Daten);
}
```

Dabei überprüfen wir wieder, ob das Feld `Naechster` ungleich `null` ist. Wenn das der Fall ist, wird die Hilfskonstruktion auf das nächste Element gesetzt und die Daten werden ausgegeben. Hat das Feld `Naechster` dagegen den Wert `null`, ist das Ende der Liste erreicht und die Ausgabe wird beendet.

Probieren Sie den Code jetzt einmal im praktischen Einsatz aus. Lassen Sie dabei zum Test auch mehr als zwei Elemente über die Schleife einfügen. Sie werden sehen, das Programm kommt auch mit mehreren Hundert Einträgen ohne Weiteres zurecht.

Unsere erste Version der einfach verketteten Liste funktioniert zwar, ist aber alles andere als objektorientiert programmiert – auch wenn wir Klassen einsetzen. Denn in unserer Klasse `Listenelement` haben wir ja lediglich die Daten für die Klasse abgebildet, aber nicht das Verhalten. Die Veränderungen an der Liste nehmen wir über Klassenmethoden vor, die – wenn Sie so wollen – außerhalb eines Listenelements arbeiten.

5.2 Die Version mit „echten“ Objekten

Wir werden die Liste daher jetzt überarbeiten und auch das Verhalten – also das Anhängen und die Ausgabe – in der Klasse selbst über Methoden abbilden. Diese zweite Version finden Sie im folgenden Code. Was dort genau geschieht, erklären wir Ihnen wieder im Anschluss.

```
/*#####
Eine verkettete Liste Version 2
##### */

using System;

namespace Cshp05d_05_02
{
    //die Klasse für die Listenelemente
    //jetzt auch mit Methoden
    class Listenelement
    {
        string daten;
        Listenelement naechster;

        //die Methode zum Setzen der Daten
        public void SetDaten(string datenNeu)
```

```
{  
    //die Zeichenkette setzen  
    daten = datenNeu;  
    //das Ende markieren  
    naechster = null;  
}  
  
//die Methode zum Anhängen eines neuen Elements  
//sie ruft sich rekursiv auf, bis das Ende erreicht ist  
public void Anhaengen(string datenNeu)  
{  
    //wenn das Ende erreicht ist, ein neues Element erzeugen  
    if (naechster == null)  
    {  
        naechster = new Listenelement();  
        naechster.SetDaten(datenNeu);  
    }  
    //sonst ruft sich die Methode selbst wieder auf  
    else  
        naechster.Anhaengen(datenNeu);  
    //zur Veranschaulichung der Rekursion  
    Console.WriteLine("Daten {0} wurden eingefügt.",  
        datenNeu);  
}  
  
//die Methode zur Ausgabe der Liste  
//sie ruft sich ebenfalls rekursiv auf, bis das  
//Ende erreicht ist  
public void Ausgeben()  
{  
    Console.WriteLine(daten);  
    if (naechster != null)  
        naechster.Ausgeben();  
}  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        //ein neues Listenelement erzeugen  
        Listenelement listenAnfang = new Listenelement();  
  
        //die Daten im ersten Listenelement setzen  
        listenAnfang.SetDaten("Element 1");  
  
        //weitere Elemente in einer Schleife anfügen  
        for (int element = 2; element < 4; element++)  
            listenAnfang.Anhaengen("Element " + element);  
        //die Liste ausgeben  
        listenAnfang.Ausgeben();  
    }
}
```

Code 5.2: Einfach verkettete Liste Version 2

Wie Sie selbst sehen, ist diese zweite Version sehr viel kompakter als die erste Version. Interessant sind vor allem die Methoden `Anhaengen()` und `Ausgeben()`. Schauen wir sie uns daher etwas genauer an.

```
public void Anhaengen(string datenNeu)
{
    //wenn das Ende erreicht ist, ein neues Element erzeugen
    if (naechster == null)
    {
        naechster = new Listenelement();
        naechster.SetDaten(datenNeu);
    }
    //sonst ruft sich die Methode selbst wieder auf
    else
        naechster.Anhaengen(datenNeu);
    //zur Veranschaulichung der Rekursion
    Console.WriteLine("Daten {0} wurden eingefügt.", datenNeu);
}
```

Beim flüchtigen Hinsehen gibt es in dieser Methode keine Besonderheiten. Wir überprüfen mit der `if`-Abfrage, ob wir das Ende der Liste erreicht haben, erzeugen dann gegebenenfalls über `new` ein neues Element und setzen die Daten für dieses Element über die Methode `SetDaten()`.

Vergleichen Sie aber einmal den Quelltext der Methode `Anhaengen()` mit dem Quelltext der Klassenmethode `ListeAnhaengen()` aus der ersten Version der einfach verketteten Liste. Achten Sie dabei vor allem darauf, wie das Ende der Liste ermittelt wird.

In der ersten Version verschieben wir in einer `while`-Schleife eine Hilfskonstruktion so lange, bis das Ende der Liste erreicht ist. In der Methode `Anhaengen()` in der zweiten Version dagegen gibt es weder die Hilfskonstruktion noch die Schleife. Wie aber wird dann dort das Ende der Liste ermittelt?

Die Antwort ist vielleicht ein wenig verblüffend. Die Funktion der `while`-Schleife aus dem Code der ersten Version übernimmt in der Methode `Anhaengen()` aus dem Code der zweiten Version die `if-else`-Konstruktion. Sehen wir uns an, was dort geschieht.

Zuerst wird überprüft, ob es ein Nachfolgeelement gibt. Dazu wird `naechster` mit dem Wert `null` verglichen.

```
if (naechster == null)
```

Trifft diese Abfrage zu, ist das Ende der Liste erreicht und es wird ein neues Element eingefügt. Dabei setzen wir die gleichen Techniken ein, die wir auch bei der einfach verketteten Liste in der ersten Version verwendet haben.

Ist das Ende der Liste dagegen noch nicht erreicht, wird der `else`-Zweig ausgeführt.

```
naechster.Anhaengen(datenNeu);
```

In dieser Anweisung ruft sich die Methode selbst wieder mit demselben Argument auf. Der Aufruf erfolgt allerdings nicht für das aktuelle Element, sondern für das Element, das über `naechster` angesprochen wird – also für das Nachfolgeelement. Auf diese Weise bewegen Sie sich in der Liste immer ein Element weiter, bis `naechster` den Wert `null` hat. Erst dann wird tatsächlich ein neues Element eingefügt. Und erst dann

werden die Daten, die als Argument beim Aufruf übergeben werden, auch tatsächlich verarbeitet. Wenn Sie so wollen, werden die Daten also von einem Element zum nächsten durchgereicht.

Der Aufruf einer Methode durch die Methode selbst wird **Rekursion** genannt. Dabei wird die Methode neu gestartet, der vorige Durchlauf allerdings nicht beendet. Er wird weiter ausgeführt, wenn alle Anweisungen des rekursiven Aufrufs komplett abgearbeitet wurden.



Da sich eine Rekursion mit einer Schleife vergleichen lässt, müssen Sie auch hier sorgfältig auf die Bedingung achten, die die Rekursion steuert. Andernfalls wird die Rekursion nie beendet und das Programm stürzt irgendwann ab. Das liegt daran, dass bei jedem rekursiven Aufruf die Adresse des vorherigen Aufrufs im Stack – einem speziellen Speicherbereich – abgelegt wird, der irgendwann „überläuft“. Dieser Überlauf wird auch *stack overflow* genannt.

Damit Sie leichter nachvollziehen können, was genau beim Anhängen neuer Elemente geschieht, zeigen wir Ihnen den Ablauf auch noch einmal mit einigen Grafiken.

Nehmen wir einmal an, unsere Liste ist noch gar nicht vorhanden. Im ersten Schritt erzeugen wir zunächst einmal eine Instanz `listenAnfang` und setzen die Daten für das erste Element. Das übernehmen im Code 5.2 von oben die ersten Anweisungen von `Main()`.

```
//ein neues Listenelement erzeugen  
Listenelement listenAnfang = new Listenelement();  
  
//die Daten im ersten Listenelement setzen  
listenAnfang.SetDaten("Element 1");
```

Die grafische Darstellung der Liste sieht dann so aus:

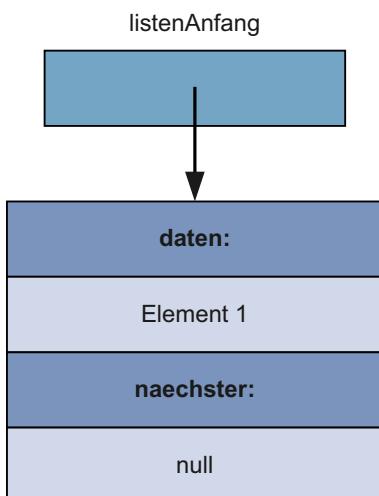


Abb. 5.7: Die Liste mit einem Element

Danach erzeugen wir in der Schleife mit der Methode `Anhaengen()` für die Instanz `listenAnfang` ein neues Element. Da `naechster` hier sofort den Wert `null` hat, wird dieses Element direkt angehängt. Der rekursive Aufruf von `Anhaengen()` erfolgt also nicht. Die Liste sieht danach so aus:

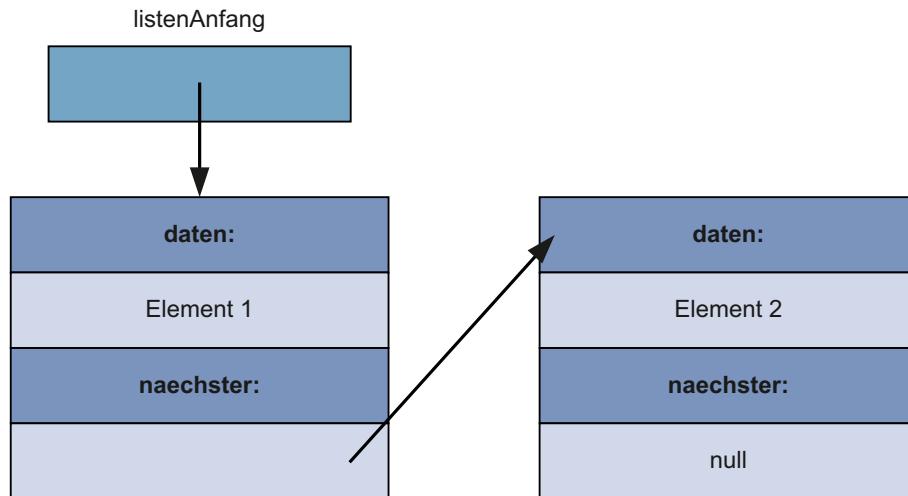


Abb. 5.8: Die Liste mit zwei Elementen

Danach rufen wir in der Schleife die Methode `Anhaengen()` für die Instanz `listenAnfang` noch einmal auf. Diese Instanz verweist die ganze Zeit auf den `Listenanfang`, da wir sie ja nicht verändern.

Hinweis:

Damit Sie in den folgenden Grafiken sofort sehen können, welcher Teil vom Quelltext noch ausgeführt werden muss, zeigen wir Ihnen jetzt immer auch unten in den Blöcken der Elemente die interessanten Anweisungen aus der Methode `Anhaengen()`. Welches Element gerade aktiv ist, erkennen Sie an dem Pfeil unterhalb der Blöcke.

Direkt nach dem Aufruf der Methode `Anhaengen()` für das Element 3 haben wir dann folgenden Zustand:

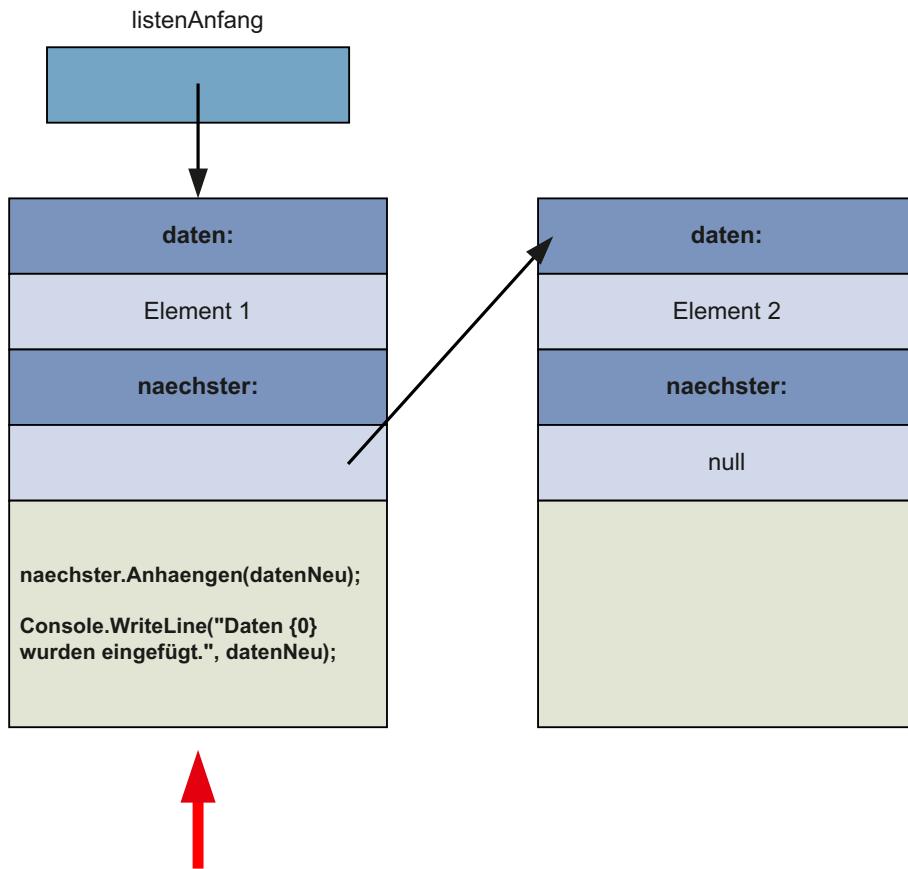


Abb. 5.9: Der Zustand direkt nach dem Aufruf der Methode `Anhaengen()` für das Element 3

Wir befinden uns im Element 1, da wir die Methode ja über die Instanz `listenAnfang` aufrufen, die die ganze Zeit auf dem Listenanfang steht. Hier wird dann die `if`-Abfrage ausgeführt. Da `naechster` ungleich `null` ist, wird die Anweisung im `else`-Zweig ausgeführt – der rekursive Aufruf der Methode `Anhaengen()` für `naechster`.

Danach haben wir folgenden Zustand:

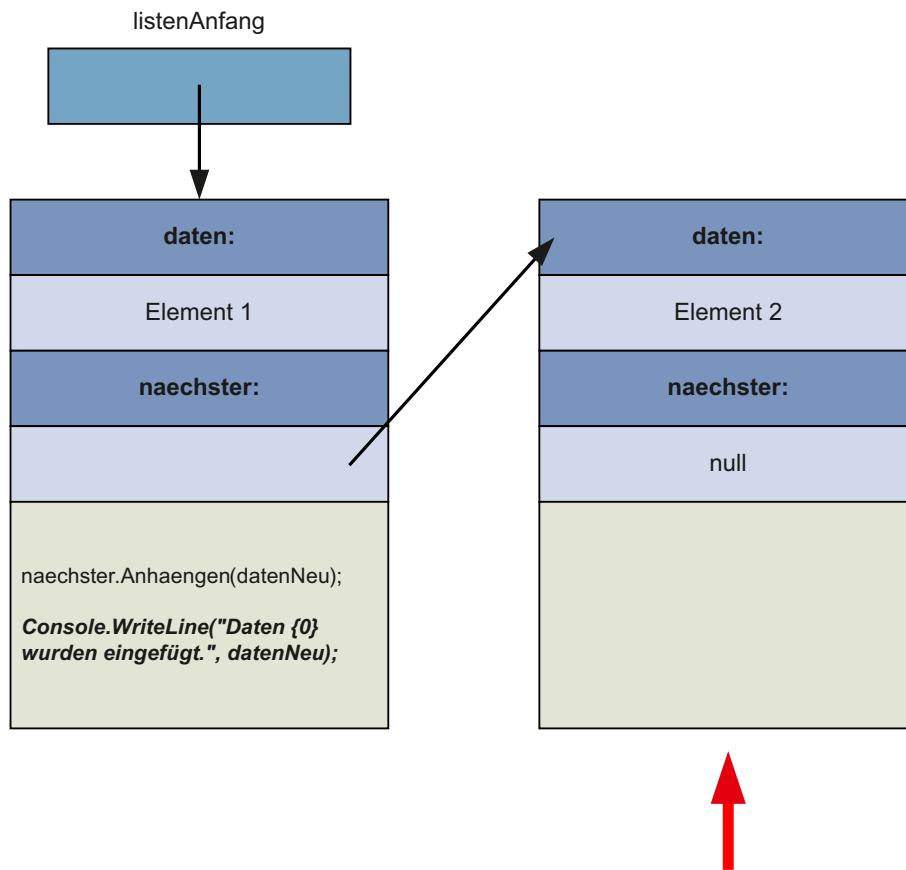


Abb. 5.10: Der Zustand nach dem ersten rekursiven Aufruf der Methode `Anhaengen()`

Bitte beachten Sie, dass nun zwar das Element 2 aktiv ist, die Ausgabeanweisung für den Aufruf mit dem Element 1 aber noch **nicht** ausgeführt wurde. Sie ist daher in der Abbildung fett und kursiv markiert. Die Anweisung wird nun nicht einfach „gelöscht“, sondern die Ausführung der Methode `Anhaengen()` für den Aufruf mit dem Element 1 wurde zunächst nur unterbrochen.

Beim Element 2 hat `naechster` den Wert `null`. Wir befinden uns also am Ende der Liste und das neue Element wird eingefügt.

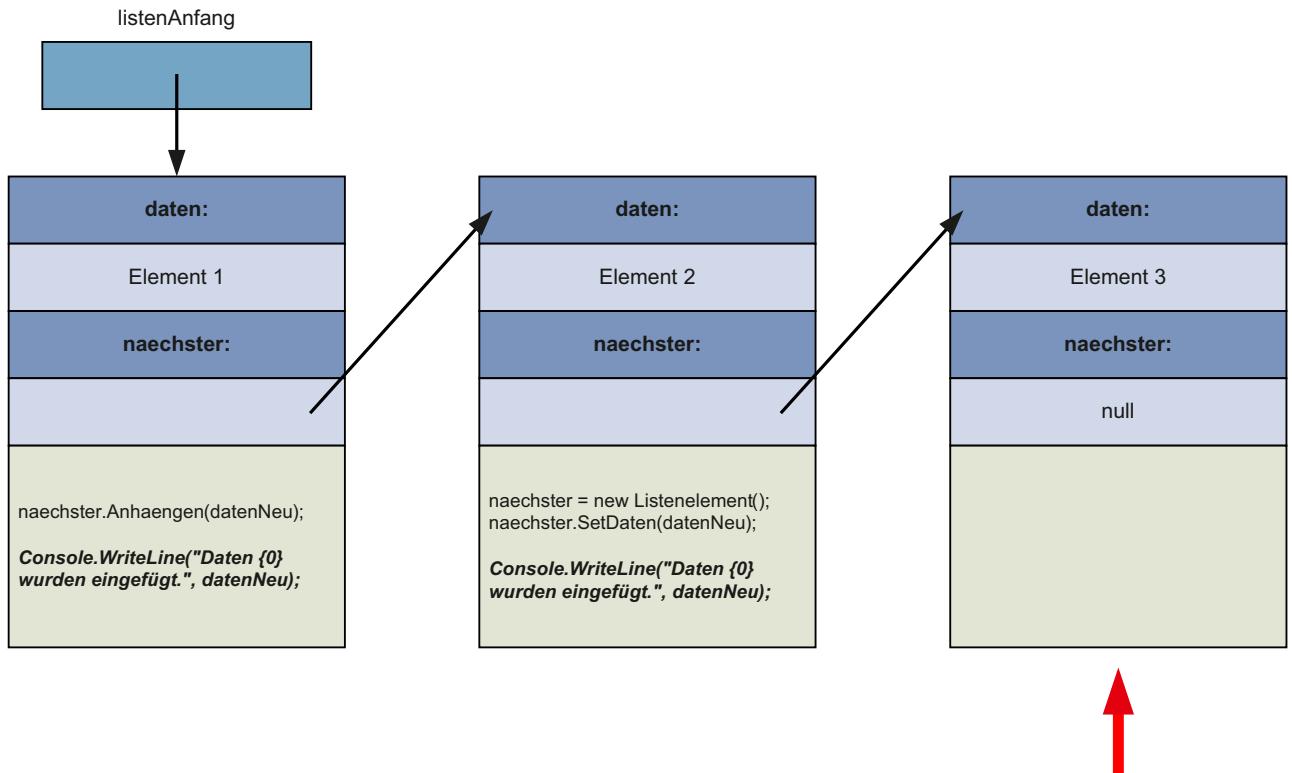


Abb. 5.11: Das neu eingefügte Element am Ende der Liste

Direkt nach dem Anhängen des zweiten Elements wird dann die Ausgabe durchgeführt – ebenfalls über die Methode `Anhaengen()` des Elements 2. Die Methode `Anhaengen()` für das Element 1 befindet sich dabei immer noch im Wartezustand. Wir haben also folgenden Zustand:

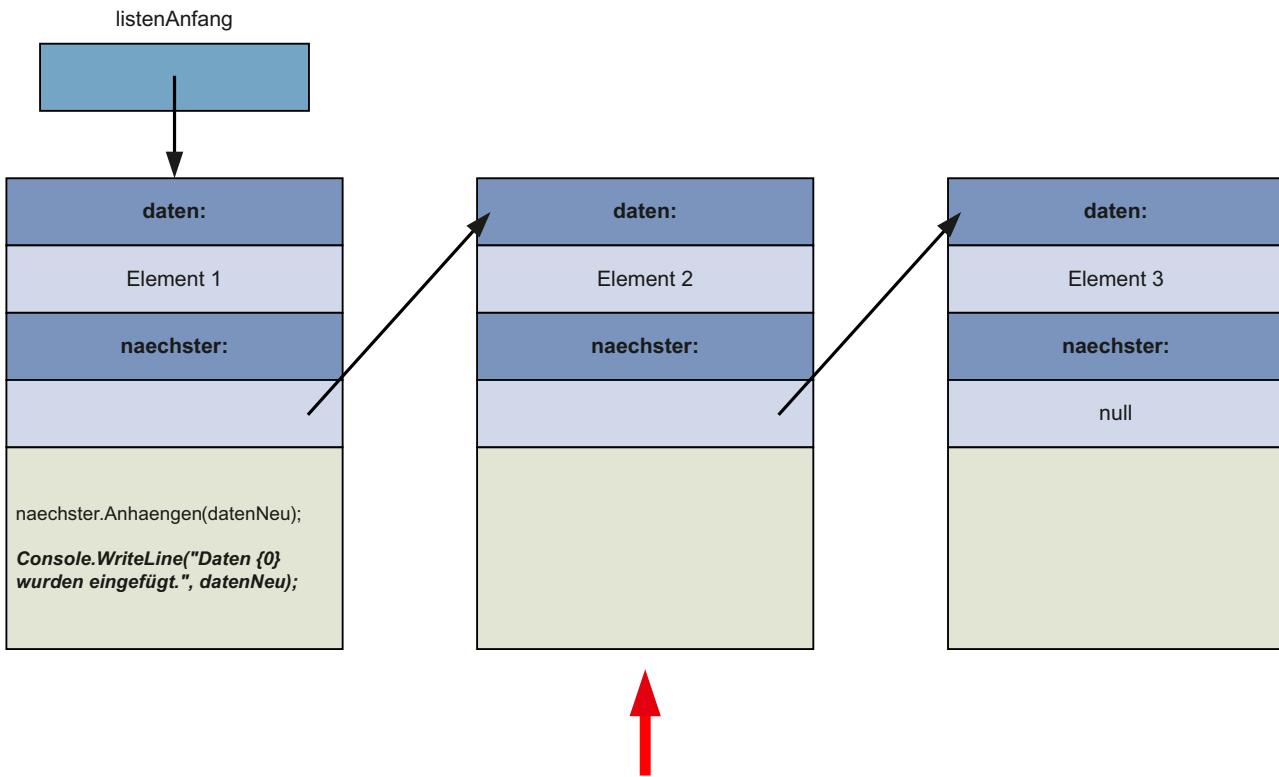


Abb. 5.12: Die Verarbeitung der „wartenden“ Methode Anhaengen() von Element 2

Dann wird die Verarbeitung mit der „wartenden“ Methode `Anhaengen()` für das Element 1 fortgesetzt und zu Ende ausgeführt. Erst wenn diese Verarbeitung abgeschlossen ist, ist der rekursive Aufruf komplett durchgeführt und die Methode `Anhaengen()` wird endgültig verlassen. Wenn Sie so wollen, wird bei der Rekursion also eine Art Stapel aufgebaut, der in umgekehrter Reihenfolge wieder abgebaut werden muss. Bei diesem Abbau werden die „wartenden“ Methoden der Reihe nach zu Ende ausgeführt.

Deshalb erfolgen in unserer Liste zum Beispiel beim Element 3 zwei Ausgaben. Denn die Methode `Anhaengen()` muss ja zweimal aufgerufen werden. Bei einem Element 4 dagegen würden drei Ausgaben erfolgen, da die Methode `Anhaengen()` dreimal aufgerufen werden muss.

Kommen wir nun noch kurz zur Methode `Ausgeben()`. Sie gibt die Daten des aktuellen Elements aus und ruft sich dann so lange über `naechster` rekursiv auf, bis das Ende der Liste erreicht ist. Auch hier wird die Liste von Anfang an bearbeitet, da der erste Aufruf in der Methode `Main()` ja über die Instanz `listenAnfang` erfolgt, die auf den Listenanfang verweist.

Durch diese rekursiven Aufrufe in den Methoden `Anhaengen()` und `Ausgeben()` können wir auch komplett auf die recht umständliche Hilfskonstruktion verzichten, da sich die Instanz `naechster` durch die rekursiven Aufrufe ja quasi „von alleine“ in der Liste weiterbewegt.

Allerdings ist die zweite Version der einfach verketteten Liste nicht mehr ganz so einfach nachzuvollziehen – auch wenn der Quelltext nur sehr kurz ist. Denn das Programm arbeitet mit einigen Konstruktionen, die möglicherweise erst beim sehr genauen Hinsehen

und mehrfachen Durcharbeiten richtig klar werden. Achten Sie deshalb vor allem auf folgende Punkte:

- Die Instanz `listenAnfang` verweist während des **gesamten** Programmablaufs ständig auf den Anfang der Liste. Sie wird einmal zu Beginn des Programms positioniert und dann nicht mehr verändert. Damit beginnen die Methoden zum Anhängen und Anzeigen immer am Anfang der Liste. Dass die Elemente bei der Methode `Anhaengen()` trotzdem am Ende der Liste eingefügt werden, liegt daran, dass sich die Methode `Anhaengen()` rekursiv mit dem nachfolgenden Element aufruft, bis das Ende der Liste erreicht ist. Erst dann erzeugt `Anhaengen()` ein **neues** Element, in dem die Daten abgelegt werden.
- Innerhalb der Methoden `Anhaengen()` und `Ausgeben()` wechselt durch den rekursiven Aufruf mit dem Nachfolger das aktive Element. Der rekursive Aufruf erfolgt also immer wieder für das nächste Element in der Liste und nicht für das Element, das ursprünglich angegeben wurde. So ist ja zum Beispiel beim ersten Aufruf der Methode `Anhaengen()` das Element 1 das aktive Element. Durch den rekursiven Aufruf wird dann aber das Element 2 zum aktiven Element.

Halten Sie sich daher beim Durcharbeiten des Codes vor Augen, welches Element gerade aktiv ist, und machen Sie sich dazu gegebenenfalls Notizen. Sie werden sehen: So schwierig ist auch die zweite Version der einfach verketteten Liste gar nicht.

Abschließend noch einige Hinweise zu der Liste und ein paar Vorschläge für eigene Experimente.

- Wenn Sie sehr viele Einträge in der Liste speichern, dauert das Anhängen neuer Elemente recht lange, da die Liste ja immer vom Anfang bis zum Ende durchlaufen werden muss, um das Ende wiederzufinden. Sie könnten hier auch eine weitere Instanz verwenden, die ständig auf das Ende der Liste positioniert ist. Damit lässt sich das Anhängen erheblich beschleunigen.
- Unsere Liste kann immer nur von vorne nach hinten durchlaufen werden, da wir ja nur das Nachfolgeelement kennen, aber nicht das Vorgängerelement. Um die Liste in beide Richtungen durchlaufen zu können, müssten Sie also noch dafür sorgen, dass auch der Vorgänger im aktuellen Element gespeichert werden kann. Solche Listen werden **doppelt verkettete Liste** genannt.

Alternativ könnten Sie aber auch die Rückwärtssausgabe durch eine Rekursion ausführen lassen, die am Ende der Liste beginnt.

- Sie können die zweite Version der Liste noch etwas kompakter gestalten, indem Sie das ausdrückliche Setzen von `naechster` in der Methode `SetDaten()` löschen und auch die Testausgabe in der Methode `Anhaengen()` entfernen.
- Wie kritisch eine endlose Rekursion werden kann, können Sie sehr einfach ausprobieren, indem Sie in der Methode `Anhaengen()` in der zweiten Version der Liste nicht die Methode `Anhaengen()` für `naechster` aufrufen, sondern nur die Methode `Anhaengen()` für das aktuelle Element. Dazu lassen Sie `naechster` vor dem Aufruf der Methode weg. Da der Aufruf nur dann erfolgt, wenn `naechster` ungleich `null` ist und sich `naechster` nicht mehr verändert, wird die Methode immer wieder aufgerufen. Das Programm wird dann recht schnell mit einem Überlauf – einem *stack overflow* – beendet.

- Bei Listen mit sehr vielen Elementen gerät das Programm ebenfalls in Schwierigkeiten, da die Rekursionen beim Anhängen neuer Elemente und auch bei der Ausgabe zu einem Überlauf führen. Sie können die Rekursionen in der zweiten Version der Liste aber auch durch Schleifen ersetzen, die ähnlich wie in der ersten Version arbeiten. Dazu vereinbaren Sie zunächst wieder eine Hilfsvariable vom Typ der Klasse für die Liste. Dieser Hilfsvariablen weisen Sie dann über `this` die Referenz auf die aktuelle Instanz zu – also der Instanz, die die Methode aufruft.

Die Methode `Ausgeben()` könnte dann zum Beispiel so aussehen:

```
public void Ausgeben()
{
    //die Hilfskonstruktion
    Listenelement hilfsKonstruktion;
    //hilfsKonstruktion auf die aktuelle Instanz setzen
    hilfsKonstruktion = this;
    //die Daten ausgeben
    Console.WriteLine(hilfsKonstruktion.daten);
    //und den Rest in einer Schleife ausgeben
    while (hilfsKonstruktion.naechster != null)
    {
        hilfsKonstruktion = hilfsKonstruktion.naechster;
        Console.WriteLine(hilfsKonstruktion.daten);
    }
}
```

Code 5.3: Der Umbau der Rekursion zu einer Schleife



Einige Erweiterungen und Umbauten der Liste warten als Einsendeaufgaben auf Sie.

5.3 Stacks und Warteschlangen

Listenkonstruktionen finden Sie übrigens sehr häufig bei der Programmierung – zum Beispiel bei einem Stack und bei Warteschlangen.

In einem **Stack** oder **Stapel** werden neue Elemente immer am Ende der Liste angefügt. Die Elemente können auch nur vom Ende der Liste wieder entnommen werden. Das heißt, das Element, das zuletzt in die Liste eingefügt wurde, muss als erstes wieder entnommen werden. Dieses Prinzip nennt man auch **LIFO-Prinzip** (*Last in, first out*, frei übersetzt: „Was zuletzt kommt, geht zuerst wieder raus“). Solch ein Stack wird auch bei Anwendungsprogrammen verwendet, um zum Beispiel die Rücksprungadressen von Methoden zu speichern. Dazu wird auf dem Stack die Adresse abgelegt, an der die Ausführung des Programms nach dem Abarbeiten einer Methode fortgeführt wird. Nach dem Rücksprung wird die Adresse automatisch wieder vom Stack genommen.

Eine **Warteschlange** dagegen arbeitet nach dem **FIFO-Prinzip** (*First in, first out*, frei übersetzt: „Was zuerst kommt, geht zuerst wieder raus“). Das Element, das zuerst in die Liste eingefügt wurde, wird auch als erstes wieder aus der Liste genommen. Dieses Prinzip finden Sie zum Beispiel häufig bei Druckerwarteschlangen wieder.

Zusammenfassung

Eine einfach verkettete Liste enthält neben den Daten immer auch einen Verweis auf das nachfolgende Listenelement.

Der Aufruf einer Methode durch die Methode selbst wird Rekursion genannt.

Typische Listenkonstruktionen sind ein Stack oder Stapel und eine Warteschlange.

Aufgaben zur Selbstüberprüfung

- 5.1 Was ist eine Rekursion? Worauf sollten Sie bei einer Rekursion besonders achten?

- 5.2 Was unterscheidet die Listenkonstruktionen Stack und Warteschlange?

Schlussbetrachtung

Herzlichen Glückwunsch!

Sie kennen nun die Grundlagen der objektorientierten Programmierung. Sie können eigene Klassen vereinbaren, Instanzen erzeugen sowie auf Felder – die Attribute – und Methoden zugreifen.

In diesem Heft haben Sie auch einige doch schon recht knifflige Aufgaben umgesetzt – zum Beispiel die Pumpstation mit der Zusammenarbeit von Methoden und die einfach verkettete Liste.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Die Darstellung in einem Objekt erfolgt abstrahiert.

Ein Objekt fasst Eigenschaften und Verhalten zusammen. Die Eigenschaften werden durch Attribute beziehungsweise Attributwerte dargestellt, das Verhalten durch Methoden.

Jedes Objekt hat eine eindeutige Identität.

Ähnliche Objekte werden zu Klassen zusammengefasst.

Für die richtige Lösung reicht es aus, wenn Sie drei der vier charakteristischen Eigenschaften genannt haben.

- 1.2 Eine Instanz ist ein Objekt genau einer Klasse.

- 1.3 Eine konkrete Klasse ist eine Klasse, für die direkte Instanzen erzeugt werden können. Eine abstrakte Klasse ist eine Klasse, die vor allem für die Ableitung anderer Klassen genutzt wird. Für eine abstrakte Klasse können keine Instanzen erzeugt werden.

- 1.4 Die Fernbedienung wartet nicht, bis der Fernseher die Operation ausgeführt hat. Damit erfolgt die Kommunikation asynchron.

Kapitel 2

- 2.1 Um eine Instanz `wagen` für eine Klasse `Auto` zu erzeugen, benutzen Sie die folgende Anweisung

```
Auto wagen = new Auto();
```

Sie können die Anweisungen auch auf zwei Zeilen verteilen. Sie sehen dann so aus:

```
Auto wagen;
wagen = new Auto();
```

- 2.2 Der Aufruf der Methode könnte zum Beispiel so erfolgen:

```
wagen.Lenken(10);
```

Der Wert in den Klammern ist beliebig.

- 2.3 Mit der Anweisung `wagen.Stoppen()` greifen Sie auf die **Methode Stoppen()** zu. Mit der Anweisung `wagen.Stoppen` dagegen greifen Sie auf das **Feld Stoppen** zu.

- 2.4 Der Garbage Collector sorgt dafür, dass nicht mehr benötigter Speicher freigegeben wird und freie nicht zusammenhängende Speicherbereiche zusammengefasst werden.

Kapitel 3

- 3.1 Die Klassenvariable hat in allen Instanzen denselben Wert. Änderungen, die Sie über eine Instanz vornehmen, wirken automatisch auch auf alle anderen Instanzen.
- 3.2 Nein, der Zugriff ist nicht möglich. Die Methode `Test()` ist durch das Schlüsselwort `static` als Klassenmethode vereinbart. Und aus einer Klassenmethode kann nicht auf ein Feld zugegriffen werden.
- 3.3 Für die Zuweisung muss die `this`-Referenz benutzt werden. Die Anweisung selbst sieht so aus:

```
this.versuch = versuch;
```

Kapitel 4

- 4.1 Die Vereinbarung könnte zum Beispiel so aussehen:

```
class Beispiel
{
    int intVariable;
    Beispiel referenz;
}
```

Kapitel 5

- 5.1 Bei einer Rekursion ruft sich eine Methode selbst auf. Sie müssen sorgfältig darauf achten, dass der rekursive Aufruf über Bedingungen gesteuert wird und nicht zu häufig erfolgt. Denn die Methode, aus der der rekursive Aufruf erfolgt, wird nicht abgebrochen, sondern bleibt im Wartezustand. Das führt bei Endlosrekursionen und sehr häufigen Wiederholungen des rekursiven Aufrufs zu einem Programmabsturz.
- 5.2 Ein Stack arbeitet nach dem LIFO-Prinzip. Der Wert, der zuletzt in die Liste geschrieben wurde, wird als erster wieder aus der Liste genommen.

Eine Warteschlange dagegen arbeitet nach dem FIFO-Prinzip. Hier wird der Wert, der zuerst in die Liste geschrieben wurde, auch als erster wieder aus der Liste genommen.

B. Glossar

Abgeleitete Klasse	Eine abgeleitete Klasse ist eine Klasse, deren Eigenschaften zum Teil von einer anderen Klasse stammen. In der abgeleiteten Klasse können zusätzliche Eigenschaften definiert werden.
Abstrakte Klasse	Über eine abstrakte Klasse kann ein allgemeines Verhalten vorgegeben werden, das in den abgeleiteten Klassen konkreter spezifiziert wird. Von abstrakten Klassen können keine Instanzen erzeugt werden. Abstrakte Klassen befinden sich in der Klassenhierarchie in der Regel sehr weit oben.
Abstraktion	Durch die Abstraktion werden die Informationen eines Objekts auf die Daten reduziert, die für die Lösung eines konkreten Problems erforderlich sind.
Asynchrone Kommunikation	Bei der asynchronen Kommunikation arbeitet der Sender weiter, ohne auf eine Rückmeldung des Empfängers zu warten.
Attribut	Attribute beschreiben den Zustand eines Objekts. Bei C# werden die Attribute Felder genannt.
Attributwert	Der Attributwert ist der Wert, den ein Attribut aktuell hat.
Datenkapselung	Die Datenkapselung ist ein Prinzip der objektorientierten Programmierung. Eine Klasse sollte nie direkt auf die Attribute einer anderen Klasse zugreifen, sondern immer nur auf die Methoden. Die Methoden ändern dann die Attribute.
Doppelt verkettete Liste	Eine doppelt verkettete Liste ist eine Listenkonstruktion, bei der ein Element immer einen Verweis zu seinem Vorgänger und seinem Nachfolger enthält. Die Liste kann in beide Richtungen durchlaufen werden.
Einfach verkettete Liste	Eine einfache verkettete Liste ist eine Listenkonstruktion, bei der ein Element immer einen Verweis zu seinem Nachfolger enthält. Die Liste kann daher nur von vorne nach hinten durchlaufen werden.
Encapsulation	<i>Encapsulation</i> ist der englische Begriff für Datenkapselung.
Feld	Als Feld wird in der Programmiersprache C# ein Attribut einer Klasse bezeichnet.

FIFO-Prinzip	FIFO steht für <i>First In – First Out</i> (frei übersetzt: „Was zuerst kommt, geht zuerst wieder raus“). Das FIFO-Prinzip wird zum Beispiel bei Listenkonstruktionen verwendet, die als Warteschlange aufgebaut sind.
Garbage Collection	Die <i>Garbage Collection</i> gibt Speicher frei und reorganisiert den Speicher. Sie wird automatisch ausgeführt.
Garbage Collector	Der <i>Garbage Collector</i> ist der Prozess, der die <i>Garbage Collection</i> ausführt.
Initialisierung	Die Initialisierung ist die erste Zuweisung eines Wertes an ein Datenobjekt.
Instanz	Eine Instanz ist ein Objekt einer Klasse.
Instanzmethode	Siehe Methode
Instanzvariable	Eine Instanzvariable ist eine Variable, die für eine Instanz gültig ist. Sie entspricht einem Feld.
Klasse	In einer Klasse werden Objekte mit ähnlichen Eigenschaften und ähnlichem Verhalten zusammengefasst. Klassen stehen in einem hierarchischen Verhältnis zueinander. Eine Klasse bildet den Bauplan für ein Objekt.
Klassenhierarchie	Die Klassenhierarchie beschreibt die Abhängigkeiten der Klassen. Aus den übergeordneten Klassen werden die Eigenschaften der untergeordneten Klassen abgeleitet.
Klassenmethode	Eine Klassenmethode ist eine Methode, die aufgerufen werden kann, ohne eine Instanz der Klasse erzeugen zu müssen. Die Vereinbarung einer Klassenmethode muss mit dem Schlüsselwort <code>static</code> erfolgen.
Klassenvariable	Eine Klassenvariable ist eine Variable, die für die gesamte Klasse gilt. Sie hat bei jeder Instanz denselben Wert. Der Zugriff ist möglich, ohne eine Instanz der Klasse erzeugen zu müssen. Die Vereinbarung einer Klassenvariablen muss mit dem Schlüsselwort <code>static</code> erfolgen.
Konkrete Klasse	Eine konkrete Klasse ist eine Klasse, für die Instanzen erzeugt werden können. Konkrete Klassen befinden sich in der Klassenhierarchie in der Regel sehr weit unten.
Konstruktor	Ein Konstruktor ist eine besondere Methode, die automatisch beim Erzeugen einer Instanz aufgerufen wird.

LIFO-Prinzip	LIFO steht für <i>Last In – First Out</i> (frei übersetzt: „Was zuletzt kommt, geht zuerst wieder raus“). Das LIFO-Prinzip wird zum Beispiel bei Listenkonstruktionen verwendet, die als Stapel aufgebaut sind.
Methoden	Methoden beschreiben das Verhalten eines Objekts.
Nachricht	Über Nachrichten kommunizieren Objekte miteinander.
Objekt	Ein Objekt in der objektorientierten Programmierung ist jede in sich geschlossene Einheit. Ein Objekt besteht aus dem Zustand und dem Verhalten.
Objektbezeichner	Der Objektbezeichner ist ein eindeutiger Name eines Objekts. Er wird einmal vergeben und kann nicht verändert werden.
Objektgleichheit	Objektgleichheit bezeichnet zwei oder mehr identische Objekte.
Objektidentität	Über die Objektidentität wird ein Objekt eindeutig identifiziert.
Objektorientierte Programmierung	Bei der objektorientierten Programmierung werden Daten- und Verhaltensaspekte gemeinsam betrachtet. Das wesentliche Element der objektorientierten Programmierung ist das Objekt.
Operationen	Siehe Methoden
Operationsaufruf	Ein Operationsaufruf ist eine Form der Kommunikation zwischen Objekten.
	Bei einem Operationsaufruf sind genau zwei Objekte beteiligt – der Sender und der Empfänger. Der Sender ruft beim Empfänger eine Methode auf.
Parallele Übertragung	Bei der parallelen Übertragung von Nachrichten können mehrere Objekte aktiv sein. Durch ein Signal können mehrere Objekte ihren Zustand ändern.
Protokoll	Das Protokoll ist die Menge aller Nachrichten, auf die ein Objekt reagieren kann.
Receiver	<i>Receiver</i> ist der englische Ausdruck für den Empfänger einer Nachricht.
Referenz	Eine Referenz ist ein Verweis auf ein Objekt.
Rekursion	Bei einer Rekursion ruft sich eine Methode selbst auf.
Sequenzielle Übertragung	Bei der sequenziellen Übertragung von Nachrichten ist immer nur ein Objekt aktiv. Wenn der Sender seine Nachricht verschickt hat, stellt er seine Arbeit ein, und ein anderes Objekt wird aktiv.

Signal	Ein Signal ist eine Form der Kommunikation zwischen Objekten.
	Ein Sender setzt eine Nachricht ab, die dann von verschiedenen Empfängern verarbeitet werden kann. Ob die Nachricht verarbeitet wird, hängt vom Empfänger ab.
Stack	Der <i>Stack</i> ist ein Teil des Speichers. Er wird weitgehend automatisch verwaltet.
	Der Begriff <i>Stack</i> wird auch für die Listenkonstruktion Stapel verwendet.
Standardkonstruktor	Ein Standardkonstruktor ist ein Konstruktor ohne Parameter.
Stapel	Ein Stapel ist eine Listenkonstruktion nach dem LIFO-Prinzip (<i>Last In – First Out</i> ; frei übersetzt: „Was zuletzt kommt, geht zuerst wieder raus“). Neue Elemente werden immer am Ende der Liste angefügt. Das Entnehmen von Elementen muss ebenfalls am Ende der Liste erfolgen.
Synchrone Kommunikation	Bei der synchronen Kommunikation wartet der Sender auf eine Meldung des Empfängers, bevor er selbst weiterarbeitet.
this-Referenz	Die <i>this</i> -Referenz bezieht sich auf die Instanz einer Klasse, die eine Methode aufgerufen hat.
Überdeckung	Bei der Überdeckung überlagert eine lokale Variable eine Klassen- oder Instanzvariable mit demselben Namen.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl. Bonn: Rheinwerk.

Theis, T. (2015). *Einstieg in Visual C# mit Visual Studio 2015. Ideal für Programmieranfänger geeignet.* 6. Auflage. Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Ein Objekt	4
Abb. 1.2	Das Objekt „Auto“	4
Abb. 1.3	Das abstrahierte Objekt „Auto“	5
Abb. 1.4	Die Klasse „Auto“	6
Abb. 1.5	Klassenhierarchie	7
Abb. 1.6	Klassen und Instanzen	8
Abb. 1.7	Änderung der Attributwerte durch eine Methode	9
Abb. 1.8	Ein Dialog zum Öffnen von Dateien	11
Abb. 1.9	Kommunikation durch Nachrichten	12
Abb. 1.10	Sequenzielle Übertragung	13
Abb. 1.11	Parallele Übertragung	14
Abb. 5.1	Eine einfach verkettete Liste	43
Abb. 5.2	Die Liste nach dem Erzeugen des ersten Elements	46
Abb. 5.3	Die Liste nach dem ersten Einfügen und vor dem zweiten Einfügen in der Schleife	47
Abb. 5.4	Die Liste nach dem Setzen von hilfsKonstruktion	47
Abb. 5.5	Die Liste unmittelbar nach dem Einfügen des neuen Elements	48
Abb. 5.6	Die Liste nach dem vollständigen Einfügen	49
Abb. 5.7	Die Liste mit einem Element	53
Abb. 5.8	Die Liste mit zwei Elementen	54
Abb. 5.9	Der Zustand direkt nach dem Aufruf der Methode Anhaengen() für das Element 3	55
Abb. 5.10	Der Zustand nach dem ersten rekursiven Aufruf der Methode Anhaengen()	56
Abb. 5.11	Das neu eingefügte Element am Ende der Liste	57
Abb. 5.12	Die Verarbeitung der „wartenden“ Methode Anhaengen() von Element 2	58

E. Codeverzeichnis

Code 2.1	Einführung in Klassen	17
Code 3.1	Aus dem Feld wird eine Klassenvariable	27
Code 3.2	Eine selbst erstellte Klasse mit Klassenvariable und Klassenmethode ..	29
Code 3.3	Eine Klassenvariable zum Zählen der Instanzen	30
Code 3.4	Ein überdecktes Feld	31
Code 3.5	Der geänderte Zugriff auf das Feld	32
Code 3.6	Drohende Verwirrung bei ähnlichen Namen	33
Code 4.1	Eine Pumpstation mit Objekten	37
Code 5.1	Einfach verkettete Liste Version 1	45
Code 5.2	Einfach verkettete Liste Version 2	51
Code 5.3	Der Umbau der Rekursion zu einer Schleife	60

F. Sachwortverzeichnis

A	
Abstraktion	5
Attribut	9
Attributwert	9
C	
C#	
Klassen und Objekte mit.....	16
Speicherverwaltung von	23
E	
Empfänger	12
F	
Feld	9
FIFO-Prinzip	60
G	
Garbage Collection	23
Garbage Collector	23
I	
Identität	10
Instanz	7
erzeugen	20
Instanzmethode	19
Instanzvariable	19
K	
Klasse	5
abstrakte	8
konkrete	7
Vereinbarung einer	18
Klassenhierarchie	6
Kommunikation	
asynchrone	14
synchrone	14
L	
LIFO-Prinzip	60
Liste	
doppelt verkettete	59
einfach verkettete	42
M	
Methode	9
N	
Nachricht	12
O	
Objekt	3
Kommunikation zwischen	11
Objektbezeichner	10
Operation	9
Operationsaufruf	12
P	
Programmierung	
objektorientierte	3
Protokoll	12
R	
receiver	12
Rekursion	53
S	
Sender	12
Signal	13
Speicherverwaltung	
dynamische	23
Stack	60
Standardkonstruktor	20
Stapel	60
T	
this-Referenz	32
U	
Überdeckung	30
Übertragung	
parallele	13
sequenzielle	13
V	
Verhalten	4

W

Warteschlange 60

Z

Zugriff auf Methoden und Felder 21

Zustand 4

G. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP05D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

- Erstellen Sie ein Programm, das einen Fernseher als Objekt abbildet. Der Fernseher soll mindestens folgende Eigenschaften haben:
 - Lautstärke,
 - Programm und
 - eingeschaltet beziehungsweise ausgeschaltet.

Alle drei Eigenschaften sollen geändert werden können.

Beachten Sie bei der Umsetzung bitte folgende Vorgaben:

- Eine Änderung von Lautstärke und Programm soll nur dann möglich sein, wenn der Fernseher eingeschaltet ist.
- Schalten Sie den Fernseher in Ihrem Programm mindestens einmal ein und ändern Sie die Lautstärke sowie das Programm. Schalten Sie den Fernseher dann wieder aus.
- Geben Sie nach jeder Änderung den aktuellen Zustand des Fernsehers auf der Konsole aus.
- Greifen Sie nur über die Methoden der Klasse auf die Felder zu. Das gilt auch für die Ausgabe der Werte.

Verwenden Sie bitte folgenden Programmkopf:

```
/* ##### Einstudeaufgabe 5.1 ##### */
```

20 Pkt.

- Erweitern Sie die zweite Version der einfach verketteten Liste so, dass das Listenende beim Anhängen nicht immer wieder neu ermittelt werden muss, sondern neue Elemente direkt am Ende der Liste angehängt werden können.

Dazu ein paar Hilfestellungen:

- Sie müssen neben dem Anfang der Liste jetzt auch das Ende der Liste in einer eigenen Instanz speichern können.

- Erstellen Sie eine Methode, die Ihnen das aktuelle Ende der Liste zurückliefert. Alternativ können Sie sich das Listenende auch von der Methode zum Anhängen liefern lassen.
- Setzen Sie den Wert der Instanz für das Listenende nach dem Anhängen neuer Elemente jeweils auf das aktuelle Ende der Liste und rufen Sie dann die Methode zum Anhängen neuer Listenelemente mit diesem Wert neu auf.

Verwenden Sie für die Lösung bitte folgenden Programmkopf:

```
/* #####  
Einsendeaufgabe 5.2  
##### */
```

40 Pkt.

3. Erweitern Sie die zweite Version der einfach verketteten Liste so, dass die Liste auch rückwärts ausgegeben werden kann.

Erstellen Sie dazu eine entsprechende Methode, die sich rekursiv aufruft.

Verwenden Sie für die Lösung bitte folgenden Programmkopf:

```
/* #####  
Einsendeaufgabe 5.3  
##### */
```

40 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Fortgeschrittene Techniken
bei der objektorientierten Programmierung

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP06D

Objektorientierte Software-Entwicklung mit C#

Fortgeschrittene Techniken bei der objektorientierten Programmierung

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Fortgeschrittene Techniken bei der objektorientierten Programmierung

Inhaltsverzeichnis

Einleitung	1
1 Datenkapselung	3
1.1 Private und öffentliche Vereinbarung von Methoden und Feldern	4
1.2 Eigenschaften	11
1.3 Regeln für das Arbeiten mit privaten und öffentlichen Vereinbarungen	15
Zusammenfassung	15
2 Konstruktoren und Destruktoren	17
2.1 Konstruktoren	17
2.2 Destruktor	22
Zusammenfassung	24
3 Überladen	26
3.1 Überladen von Methoden	26
3.2 Überladen von Konstruktoren	29
Zusammenfassung	32
4 Vererbung	34
4.1 Das Konzept der Vererbung	34
4.2 Vererbung in C#	39
4.3 Zugriff auf geerbte Attribute	47
Zusammenfassung	49
5 Polymorphismus und überschriebene Methoden	51
5.1 Ein wenig Theorie	51
5.2 Die praktische Umsetzung	53
5.3 Dynamische Bindung	57
Zusammenfassung	59
6 Wiederverwendung von Quelltexten	61
Zusammenfassung	64
Schlussbetrachtung	66

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	67
B.	Glossar	70
C.	Literaturverzeichnis	74
D.	Abbildungsverzeichnis	75
E.	Tabellenverzeichnis	76
F.	Codeverzeichnis	77
G.	Sachwortverzeichnis	78
H.	Einsendeaufgabe	79

Einleitung

In diesem Studienheft werden wir tiefer in die objektorientierte Programmierung einsteigen und uns mit fortgeschrittenen Techniken beschäftigen – zum Beispiel mit der Datenkapselung und der Vererbung.

Im Einzelnen lernen Sie in diesem Studienheft,

- was sich hinter dem Begriff Datenkapselung verbirgt,
- was private und öffentliche Vereinbarungen von Methoden und Feldern sind,
- wie Sie diese Vereinbarungen einsetzen,
- was eine Eigenschaft einer Klasse ist,
- wie Sie Eigenschaften für eine Klasse erstellen,
- was Konstruktoren und Destruktoren sind,
- wie Sie Konstruktoren und Destruktoren für eine Klasse erstellen,
- wie Sie Methoden überladen,
- wie Sie für eine Klasse unterschiedliche Konstruktoren erstellen,
- was die Vererbung ist und wie Sie die Vererbung in C# umsetzen,
- wie Sie den Konstruktor einer übergeordneten Klasse aufrufen,
- was Polymorphismus bedeutet,
- wie Sie Methoden überschreiben,
- was die statische und die dynamische Bindung unterscheidet und
- wie Sie Quelltexte wiederverwenden.

Beginnen werden wir mit der Datenkapselung.

Christoph Siebeck

1 Datenkapselung

In diesem Kapitel beschäftigen wir uns mit der Datenkapselung.

Wie Sie bereits wissen, sollte ein Zugriff auf die Attribute einer Klasse nicht von außen erfolgen, sondern ausschließlich über die Methoden der Klasse. Eine Klasse sollte also nicht direkt auf die Attribute einer anderen Klasse zugreifen, sondern nur Methoden der anderen Klasse aufrufen, die dann ihrerseits die Attribute verändern.

Dieses Prinzip bezeichnet man als **Datenkapselung** oder **encapsulation^{a)}**.



a) *Encapsulation* bedeutet wörtlich übersetzt so viel wie „Kapselung“.

Was sich zunächst einmal sehr umständlich anhört, hat gleich zwei Vorteile:

- Ihre Programme werden übersichtlicher und leichter zu pflegen. Denn durch die direkte Veränderung eines Attributs von außen kann die Fehlersuche extrem aufwendig werden. Bei Fehlern müssten Sie ja unter Umständen den gesamten Quelltext durchsuchen. Bei einer streng gekapselten Klasse dagegen, die keinen direkten Zugriff auf ihre Attribute ermöglicht, können Sie die Fehlersuche auf die Klasse selbst beschränken.
- Konsequent umgesetzt ermöglicht die Datenkapselung außerdem eine problemlose Wiederverwendung von einmal erstellten Klassen. So können Sie allgemeingültige Klassen programmieren – zum Beispiel zum Erzeugen von bestimmten Dialogfenstern oder zum Suchen und Ersetzen von Daten, die Sie dann in beliebigen anderen Quelltexten einsetzen können.

Wenn Sie solche Klassen einmal erstellt haben, müssen Sie sich nicht mehr darum kümmern, wie die Funktionalität intern ausgeführt wird. Sie müssen lediglich wissen, wie Sie die Methoden der Klasse aufrufen und welche Daten Sie übergeben müssen.

Dieses Verfahren wird durch das **Geheimnisprinzip** – auch **information hiding^{a)}** genannt – ermöglicht. Wichtig ist vor allem das **Was** einer Klasse und nicht das **Wie**.



a) *Information hiding* bedeutet übersetzt so viel wie „Information verstecken“.

Genauso interessant ist das Geheimnisprinzip natürlich, wenn Sie nicht selbst programmierte Elemente verwenden, sondern vorgefertigte Software-Bausteine, die von Dritten zur Verfügung gestellt werden. Hier rufen Sie dann lediglich den gewünschten Baustein auf und übergeben die erforderlichen Daten. Um alles Weitere müssen Sie sich nicht kümmern.

Die Datenkapselung in C# erfolgt vor allem durch die Festlegung der **Sichtbarkeit** von Methoden und Feldern.

Die Sichtbarkeit legt fest, wie streng die Datenkapselung erfolgt – also, von welcher Stelle aus Zugriffe auf die Methoden und Felder einer Klasse möglich sind.



Schauen wir uns das jetzt an einigen Beispielen an.

1.1 Private und öffentliche Vereinbarung von Methoden und Feldern

Als Beispiel nehmen wir eine Klasse `Esel` mit den Feldern `sturheit` und `gewicht`. Die Vereinbarung dieser Klasse sieht so aus:

```
class Esel
{
    int sturheit;
    int gewicht;
}
```

Im folgenden Code erzeugen wir dann eine Instanz `eselchen` für die Klasse `Esel` und versuchen, den Wert des Feldes `gewicht` aus der Methode `Main()` zu ändern. Dieser direkte Zugriff auf das Feld ist zwar schlechter Programmierstil, wir wollen ihn hier aber zur Demonstration trotzdem einsetzen.

```
/*#####
Zugriff auf ein Feld von außen
Das Programm lässt sich nicht übersetzen
#####*/
using System;

namespace Cshp06d_01_01
{
    //die Klasse Esel
    class Esel
    {
        int sturheit;
        int gewicht;
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Instanz eselchen erzeugen
            Esel eselchen = new Esel();

            //den Wert von gewicht von außen setzen
            eselchen.gewicht = 10;

            //bitte in einer Zeile eingeben
            Console.WriteLine("Der Esel wiegt {0} Kilo.",
                eselchen.gewicht);
        }
    }
}
```

Code 1.1: Zugriff auf ein Feld von außen (das Programm lässt sich so nicht übersetzen)

Wenn Sie versuchen, diesen Code auszuführen, erhalten Sie lediglich zweimal die Fehlermeldung

Der Zugriff auf "Esel.gewicht" ist aufgrund des Schutzgrads nicht möglich.

Sowohl die Veränderung des Feldes gewicht der Instanz eselchen durch die Anweisung

```
eselchen.gewicht = 10;
```

als auch die Ausgabe des Gewichts durch die Anweisung

```
Console.WriteLine("Der Esel wiegt {0} Kilo.",  
    eselchen.gewicht);
```

scheitern.

Die Erklärung ist einfach.

Ohne weitere Angaben sind sowohl die Felder als auch die Methoden einer Klasse immer privat. Auf private Felder und Methoden kann nicht von außen zugegriffen werden.



Ob ein Feld oder eine Methode privat ist, können Sie auch in der Klassenansicht nachsehen. Klicken Sie dazu auf den Eintrag **Klassenansicht** im Menü **Ansicht**. Öffnen Sie anschließend in der Klassenansicht, die rechts auf dem Bildschirm angezeigt wird, den Zweig für die Anwendung und den Namensraum jeweils durch einen Mausklick auf das Symbol vor dem Eintrag und klicken Sie mit der Maus auf den Eintrag der gewünschten Klasse. Im unteren Bereich der Klassenansicht werden danach die Felder und Methoden der Klasse mit entsprechenden Symbolen angezeigt. Das kleine verriegelte Vorhängeschloss steht dabei zum Beispiel für die Sichtbarkeit privat.

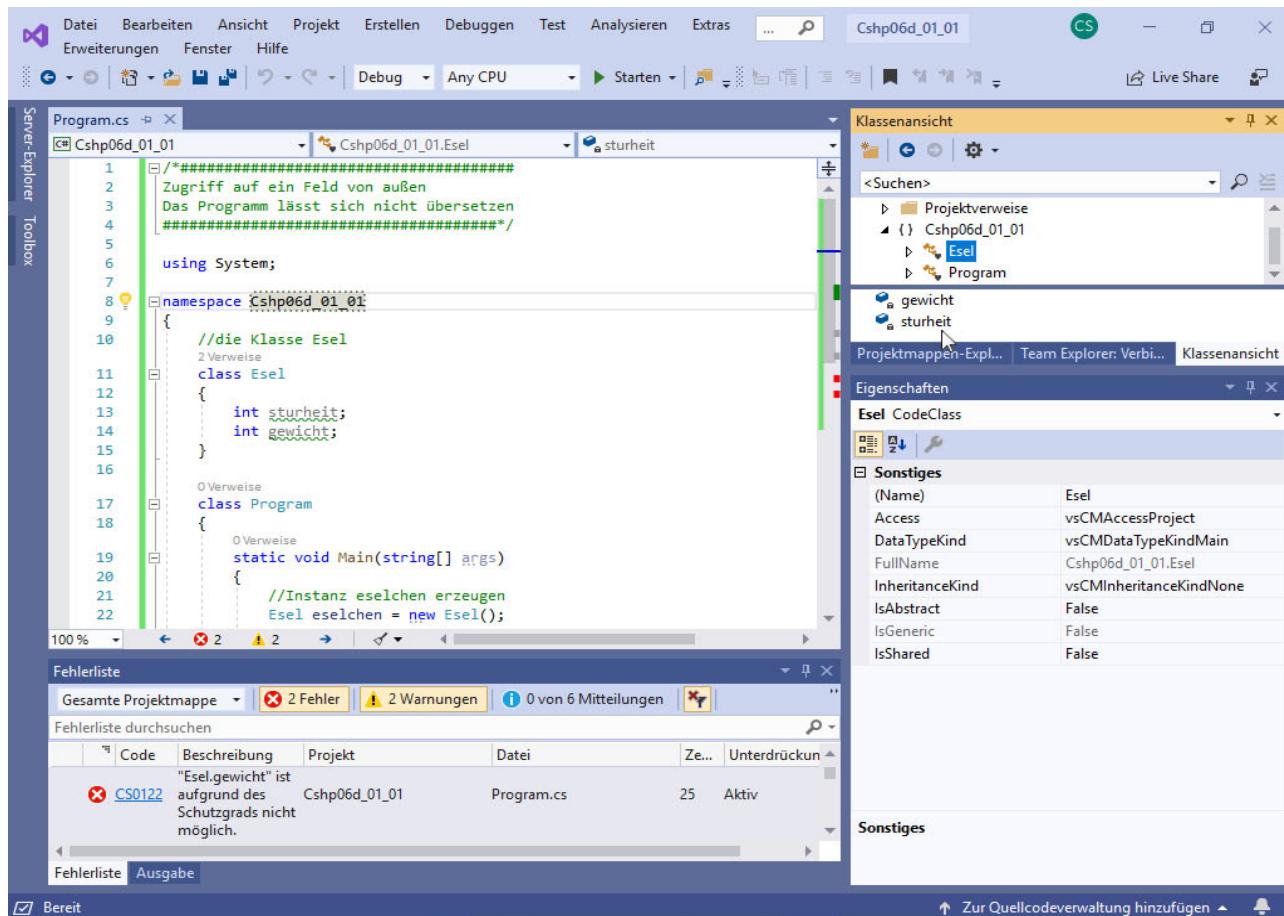


Abb. 1.1: Die Klassenansicht für die Klasse Esel
(die Einträge der beiden Felder befinden sich oberhalb des Mauszeigers)

Nun ist aber eine Klasse, die nur private Felder und Methoden hat, nicht sonderlich nützlich. Sie könnten zwar die Klasse erstellen und auch Instanzen anlegen, aber weder die Methoden der Klasse aufrufen noch den Feldern Werte zuweisen. Damit wäre die Klasse sinnlos.

C# kennt daher das Schlüsselwort `public`¹. Dieses Schlüsselwort haben Sie ja bereits eingesetzt, um zum Beispiel auf die Mitglieder einer Struktur zuzugreifen oder die Methoden einer Klasse von außen aufrufen zu können. Wir wollen es uns jetzt einmal ein wenig genauer ansehen.



Mit dem Schlüsselwort `public` werden Felder und Methoden einer Klasse als öffentlich vereinbart. Damit ist der Zugriff auf die Felder und Methoden von außen möglich.

1. *Public* bedeutet übersetzt „öffentlicht“.

Beispiel 1.1:

Mit der folgenden Vereinbarung werden sowohl das Feld `sturheit` als auch das Feld `gewicht` der Klasse `Esel` als öffentlich festgelegt.



```
class Esel
{
    public int sturheit;
    public int gewicht;
}
```

Bei der Vereinbarung

```
class Esel
{
    int sturheit;
    public int gewicht;
}
```

ist dagegen nur das Feld `gewicht` öffentlich. Das Feld `sturheit` ist nach wie vor privat, da keine besondere Angabe der Sichtbarkeit erfolgt.

Mit dieser geänderten Klassenvereinbarung sollte auch der direkte Zugriff von außen auf das Feld `gewicht` funktionieren. Der Code sieht dann so aus:

```
#####
Zugriff auf ein Feld von außen
Mit der Sichtbarkeit public klappt es
#####

using System;

namespace Cshp06d_01_02
{
    //die Klasse Esel
    class Esel
    {
        int sturheit;
        public int gewicht;
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Instanz eselchen erzeugen
            Esel eselchen = new Esel();
```

```
//das Attribut gewicht von außen setzen
eselchen.gewicht = 10;

//bitte in einer Zeile eingeben
Console.WriteLine("Der Esel wiegt {0} Kilo.",
eselchen.gewicht);
}
}
```

Code 1.2: Beispiel für eine öffentliche Feldvereinbarung

Jetzt wird das Programm ohne Probleme übersetzt und auch ausgeführt. Wie Sie an der Ausgabe sehen können, ist der Wert von `gewicht` korrekt auf 10 gesetzt worden.

Wenn Sie ein Feld oder eine Methode ausdrücklich als privat vereinbaren wollen, verwenden Sie das Schlüsselwort `private`². In der folgenden Klassenvereinbarung ist zum Beispiel das Feld `sturheit` öffentlich und das Feld `gewicht` dann wieder privat:

```
class Esel
{
    public int sturheit;
    private int gewicht;
}
```

Die verschiedenen Sichtbarkeiten können Sie beliebig oft und in beliebiger Reihenfolge in einer Klassenvereinbarung verwenden. Möglich wäre also auch folgende Vereinbarung für eine Klasse `Test`:

```
class Test
{
    private int a;
    private double b;
    public float c;
    public int d;
    string e;
    int f;
}
```

Die Felder `a` und `b` sind privat, `c` und `d` öffentlich und `e` und `f` wieder privat.



Noch einmal zur Erinnerung:

Ohne Angaben einer Sichtbarkeit wird automatisch die Sichtbarkeit `private` benutzt.

Die Sichtbarkeiten lassen sich nun nicht nur für Felder setzen, sondern auch für Methoden. So sind zum Beispiel die Felder einer Klasse häufig privat, die Methoden dagegen öffentlich beziehungsweise ebenfalls privat, wenn sie nur innerhalb der Klasse benutzt werden.

2. *Private* bedeutet übersetzt „privat“.

Schauen wir uns auch dazu ein Beispiel an:

Unsere Klasse `Esel` soll neben den privaten Feldern `gewicht` und `sturheit` noch die Methoden `Arbeiten()`, `Fressen()`, `Wiegen()` und `Init()` erhalten. Die entsprechende Vereinbarung der Klasse und den Einsatz der Methoden finden Sie im folgenden Code.

```
#####
# Beispiel für die gemischte Verwendung von
# öffentlicher und privater Vereinbarung
#####

using System;

namespace Cshp06d_01_03
{
    //die Klasse Esel
    class Esel
    {
        //die privaten Felder
        private int sturheit;
        private int gewicht;

        //die öffentlichen Methoden
        public void Arbeiten()
        {
            if (sturheit > 9 || gewicht < 10)
                Console.WriteLine("Der Esel kann nicht arbeiten!");
            else
            {
                sturheit++;
                gewicht--;
                Console.WriteLine("Der Esel hat jetzt gearbeitet.");
            }
        }

        public void Fressen()
        {
            gewicht++;
            if (sturheit > 0)
                sturheit--;
            Console.WriteLine("Der Esel hat jetzt gefressen.");
        }

        public int Wiegen()
        {
            return gewicht;
        }
    }
}
```

```

public void Init()
{
    gewicht = 15;
    sturheit = 5;
}

class Program
{
    static void Main(string[] args)
    {
        //Instanz eselchen erzeugen
        Esel eselchen = new Esel();

        //die Methoden aufrufen
        eselchen.Init();
        eselchen.Arbeiten();
        eselchen.Fressen();
        //bitte in einer Zeile eingeben
        Console.WriteLine("Der Esel wiegt {0} Kilo.", eselchen.Wiegen());
    }
}

```

Code 1.3: Gemischte Verwendung von öffentlicher und privater Vereinbarung

Hinweis:

Die Angabe von `private` bei den Vereinbarungen in der Klasse `Esel` kann auch wegfallen. Wir haben das Schlüsselwort nur angegeben, damit die verschiedenen Sichtbarkeiten eindeutig zu erkennen sind.

Die einzelnen Anweisungen des Codes sollten Sie nicht vor große Probleme stellen.

Mit der Methode `Init()` weisen wir dem Esel Anfangswerte für das Gewicht und die Sturheit zu. Die Methode wird direkt nach dem Start des Programms aufgerufen.

Die Methoden `Arbeiten()` und `Fressen()` verändern die Werte der Felder `sturheit` und `gewicht`.

Mit der Methode `Wiegen()` schließlich lassen wir das Gewicht des Esels zurückliefern.

Da sowohl das Feld `sturheit` als auch das Feld `gewicht` als `private` vereinbart sind, können sie nicht direkt von außen abgefragt und verändert werden, sondern ausschließlich über die Methoden der Klasse `Esel` selbst. So erfolgt im vorigen Code die Ausgabe des Gewichts quasi „indirekt“ über den Aufruf der Methode `Wiegen()`, die uns den aktuellen Wert des privaten Feldes `gewicht` liefert.

Hinweis:

Neben den Vereinbarungen als `private` und `public` gibt es auch noch weitere Sichtbarkeiten für Felder und Methoden beziehungsweise für die gesamte Klasse. Damit werden wir uns im Verlauf dieses Studienhefts noch intensiver beschäftigen.

1.2 Eigenschaften

Zusätzlich zu den Feldern einer Klasse können Sie in C# auch **Eigenschaften** oder *Properties* einsetzen. Diese Eigenschaften sehen auf den ersten Blick wie „normale“ Felder aus, werden aber immer über zwei spezielle Methoden verarbeitet – eine `get()`-Methode zum Abrufen des Wertes und eine `set()`-Methode zum Setzen eines Wertes.

Eigenschaften ähneln Feldern. Die Werte werden aber immer über spezielle Methoden gelesen und auch gesetzt.



Die Vereinbarung einer Eigenschaft unterscheidet sich nicht weiter von der Vereinbarung eines normalen Feldes. Die einzige Besonderheit liegt in der Vereinbarung der `get()`- und `set()`-Methoden. Sie erfolgt direkt bei der Vereinbarung in geschweiften Klammern.

So könnten Sie zum Beispiel eine Eigenschaft `GewichtInGramm` für unsere Eselklasse erstellen, die über das Feld `gewicht` das aktuelle Gewicht des Esels in Gramm liefert. Diese Eigenschaft mit der `get()`-Methode würde dann so aussehen:

```
public int GewichtInGramm
{
    //die get() -Methode
    get
    {
        return gewicht * 1000;
    }
}
```

Besonderheiten gibt es bei der `get()`-Methode eigentlich nicht. Sie liefert einen `int`-Typ zurück, der aus dem Wert des Feldes `gewicht` multipliziert mit 1 000 ermittelt wird.

Hinweis:

Achten Sie sorgfältig auf die geschweiften Klammern bei der Vereinbarung einer Eigenschaft. Sowohl die Methoden der Eigenschaft als auch die Anweisungen der Methoden selbst müssen durch geschweifte Klammern umfasst werden.

Der lesende Zugriff auf die Eigenschaft erfolgt jetzt genau wie der Zugriff auf ein „normales“ Feld – zum Beispiel durch die Anweisung

```
Console.WriteLine("Der Esel wiegt {0} Gramm.",
    eselchen.GewichtInGramm);
```

Damit könnten Sie also auch von außen auf das private Feld `gewicht` der Instanz `eselchen` zugreifen – vorausgesetzt, die Eigenschaft `GewichtInGramm` ist öffentlich vereinbart. Der Zugriff erfolgt dabei „sauber“ über die `get()`-Methode der Eigenschaft.

Ein schreibender Zugriff in der Form

```
eselchen.GewichtInGramm = 10000;
```

wäre mit der Vereinbarung von oben allerdings nicht möglich. Dazu müssten Sie auch eine entsprechende `set()`-Methode programmieren.



Auf Eigenschaften, die nur eine `get()`-Methode haben, können Sie nur lesend zugreifen. Der Wert einer Eigenschaft, die nur eine `set()`-Methode hat, kann nur geschrieben werden.

In der `set()`-Methode können Sie dann mit dem Schlüsselwort `value` auf den übergebenen Wert zugreifen. Die Eigenschaft `GewichtInGramm` mit einer `get()`- und einer `set()`-Methode könnte dann zum Beispiel so aussehen:

```
public int GewichtInGramm
{
    //die get()-Methode
    get
    {
        return gewicht * 1000;
    }

    //die set()-Methode
    //value steht für den Wert, der übergeben wird
    set
    {
        gewicht = value / 1000;
    }
}
```



Bitte beachten Sie:

Eigenschaften müssen genau wie Felder und Methoden innerhalb der Klassenvereinbarung festgelegt werden. Andernfalls weiß der Compiler nicht, zu welcher Klasse die Eigenschaft gehören soll.

Auch wenn der Wert einer Eigenschaft eigentlich über Methoden verarbeitet wird, müssen Sie ihn wie ein normales Feld behandeln – also zum Beispiel über den Operator `=` zuweisen. Denn die jeweiligen Methoden werden bei lesenden und schreibenden Zugriffen automatisch vom Compiler aufgerufen.

Die `get()`- und `set()`-Methoden einer Eigenschaft können nur Werte zurückliefern, die zum Typ der Eigenschaft passen.

Sie können in der `get()`-Methode einer Eigenschaft nicht lesend auf den Wert derselben Eigenschaft zugreifen und in der `set()`-Methode nicht schreibend. Denn damit würden Sie ja die jeweiligen Methoden selbst wieder aufrufen. Und das führt zu einer endlosen Rekursion – also einem ständigen Aufruf der Methode durch die Methode selbst. Diese Rekursion lässt das Programm sehr schnell abstürzen. Probieren Sie das einfach einmal in dem Code unten aus. Ergänzen Sie zum Beispiel in der `get()`-Methode der Eigenschaft `GewichtInGramm` eine `if`-Abfrage, die auf den Wert von `GewichtInGramm` zugreift. Sie werden sehen, das Programm wird sehr schnell mit einer Fehlermeldung beendet.

Im folgenden Code finden Sie die Eigenschaft `GewichtInGramm` für unsere Eselklasse noch einmal im praktischen Einsatz. Nutzen Sie diesen Code auch für eigene Experimente.

```
#####
# Einsatz von Eigenschaften
#####
using System;

namespace Cshp06d_01_04
{
    //die Klasse Esel
    class Esel
    {
        //die privaten Felder
        private int sturheit;
        private int gewicht;

        //die Eigenschaft
        public int GewichtInGramm
        {
            //die get()-Methode
            get
            {
                return gewicht * 1000;
            }

            //die set()-Methode
            //value steht für den Wert, der übergeben wird
            set
            {
                gewicht = value / 1000;
            }
        }

        //die öffentlichen Methoden
        public void Arbeiten()
        {
            if (sturheit > 9 || gewicht < 10)
                Console.WriteLine("Der Esel kann nicht arbeiten!");
            else
            {
                sturheit++;
                gewicht--;
                Console.WriteLine("Der Esel hat jetzt gearbeitet.");
            }
        }

        public void Fressen()
        {
            gewicht++;
            if (sturheit > 0)
                sturheit--;
            Console.WriteLine("Der Esel hat jetzt gefressen.");
        }
    }
}
```

```

public int Wiegen()
{
    return gewicht;
}

public void Init()
{
    gewicht = 15;
    sturheit = 5;
}
}

class Program
{
    static void Main(string[] args)
    {
        //Instanz eselchen erzeugen
        Esel eselchen = new Esel();

        //die Methoden aufrufen
        eselchen.Init();
        eselchen.Arbeiten();
        eselchen.Fressen();
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Der Esel wiegt {0} Kilo.",eselchen.Wiegen());

        //die Eigenschaft gewichtInGramm abfragen
        Console.WriteLine("Der Esel wiegt {0} Gramm.",eselchen.GewichtInGramm);
        //die Eigenschaft gewichtInGramm setzen
        eselchen.GewichtInGramm = 10000;
        //dadurch verändert sich auch das Gewicht
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Der Esel wiegt {0} Gramm.",eselchen.GewichtInGramm);
        Console.WriteLine("Der Esel wiegt {0} Kilo.",eselchen.Wiegen());
    }
}
}

```

Code 1.4: Eine Eigenschaft im praktischen Einsatz

Tipp:

Eigenschaften erkennen Sie in der Klassenansicht und der Memberliste an dem Symbol .

Zum Abschluss dieses Kapitels wollen wir uns noch kurz ansehen, wann Sie welche Sichtbarkeit für Felder, Eigenschaften und Methoden einer Klasse einsetzen.

1.3 Regeln für das Arbeiten mit privaten und öffentlichen Vereinbarungen

Grundsätzlich gilt bei der Arbeit mit privaten und öffentlichen Vereinbarungen von Feldern, Eigenschaften und Methoden einer Klasse:

So privat wie möglich, so öffentlich wie nötig.



So werden zum Beispiel Felder einer Klasse normalerweise privat vereinbart, Methoden und Eigenschaften dagegen öffentlich. Wenn Sie eine Methode nur innerhalb einer Klasse benötigen, können Sie diese Methode ebenfalls privat vereinbaren.

Lassen Sie sich auf keinen Fall dazu verleiten, aus Bequemlichkeit einfach sämtliche Felder und Methoden einer Klasse öffentlich zu vereinbaren. Denken Sie bitte daran: Die öffentliche Vereinbarung eines Feldes verstößt gegen das Prinzip der Datenkapselung.

Bei kleinen überschaubaren Programmen muss das nicht zwangsläufig zu Problemen führen. Bei größeren Projekten aber laufen Sie Gefahr, dass unter Umständen sehr unangenehme Fehler auftreten, die nur mit sehr viel Aufwand zu finden und auch zu beheben sind.

Auch wenn Sie im Moment vielleicht noch den Eindruck haben, dass die Datenkapselung nur für überflüssige Arbeit sorgt – spätestens, wenn Sie das erste Mal eine komplexere Klasse in einem anderen Programm einsetzen wollen, werden Sie feststellen, dass dieses Prinzip enorme Vorteile hat. Wenn Sie es konsequent umsetzen, können Sie sich viel Arbeit sparen, da die Klassen sehr einfach wiederzuverwenden sind.

Im nächsten Kapitel werden wir uns mit einigen speziellen Methoden beschäftigen, die beim Erzeugen und beim Zerstören einer Instanz ausgeführt werden.

Zusammenfassung

Private Felder und Methoden stehen nur der Klasse selbst zur Verfügung. Ein Zugriff von außen ist nicht möglich.

Auf öffentliche Felder und Methoden kann von außen zugegriffen werden.

Ohne weitere Angaben sind Felder und Methoden einer Klasse immer privat.

Eigenschaften einer Klasse ähneln „normalen“ Feldern. Die Werte einer Eigenschaft werden aber immer über spezielle `get()`- und `set()`-Methoden verarbeitet.

Methoden und Felder einer Klasse sollten so privat wie möglich und so öffentlich wie nötig sein.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Wie lauten die Schlüsselwörter für die private und die öffentliche Vereinbarung einer Methode beziehungsweise eines Feldes?

- 1.2 Sie wollen eine Klasse `Wein` erstellen. Die Klasse soll die Felder `alter`, `menge` und `wert` enthalten, jeweils vom Typ `int`. Außerdem soll die Klasse die folgenden Methoden enthalten:

- `Init()` zum Initialisieren der Werte für `alter` und `menge` einer neuen Instanz,
- `Altern()` zum Erhöhen des Werts von `alter`,
- `Trinken()` zum Verringern des Werts von `menge` und
- `WertBerechnen()` zum Berechnen des Wertes aus der Menge und dem Alter.

Die Methode `WertBerechnen()` soll nur innerhalb der Methoden `Trinken()`, `Altern()` und `Init()` der Klasse `Wein` aufgerufen werden.

Schreiben Sie die Klassenvereinbarung für die Klasse `Wein`. Arbeiten Sie dabei so weit wie möglich mit privaten Feldern und Methoden. Die Rümpfe für die Methoden können Sie leer lassen.

2 Konstruktoren und Destruktoren

In diesem Kapitel lernen Sie spezielle Methoden kennen, die beim Erzeugen und Zerstören von Instanzen ausgeführt werden – die Konstruktoren und den Destruktor.

2.1 Konstruktoren

Bisher haben wir in unseren Programmen beim Einsatz von Instanzen immer zwei Schritte durchgeführt: Zuerst haben wir die Instanz erzeugt und danach die Felder mit einer eigenen Methode initialisiert.

Diese Technik findet sich auch in dem folgenden Beispiel wieder:

```
#####
Initialisierung mit der Methode Init()
#####

using System;

namespace Cshp06d_02_01
{
    //die Klasse Sherry
    class Sherry
    {
        //die Felder
        int alter;
        int liter;

        //die Methoden
        //zur Initialisierung
        //die Felder werden über this angesprochen
        public void Init(int alter, int liter)
        {
            this.alter = alter;
            this.liter = liter;
        }

        //zum Ansehen
        public void Ansehen()
        {
            Console.WriteLine("Der Sherry ist {0} Jahre alt.", alter);
            Console.WriteLine("Die Flasche enthält {0} Liter.",
                liter);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Instanz flaschel erzeugen
            Sherry flaschel = new Sherry();
```

```

    //die Felder initialisieren
    flasche1.Init(10, 1);

    //die Werte ausgeben
    flasche1.Ansehen();
}

}

```

Code 2.1: Initialisierung mit der Methode `Init()`

Vereinbart wird eine Klasse `Sherry` mit den beiden Feldern `alter` und `liter`. Der Zugriff auf diese beiden Felder erfolgt über die Methoden der Klasse. Die Methode `Init()` setzt dabei die Werte der beiden Felder, und die Methode `Ansehen()` dient zum Abfragen der Werte.



Zur Auffrischung:

In der Methode `Init()` greifen wir über die `this`-Referenz auf die Felder `alter` und `liter` zu. Andernfalls würden die beiden Felder ja durch die gleichnamigen lokalen Variablen aus der Parameterliste der Methode überdeckt. Sie können dieses Problem der Überdeckung aber auch lösen, indem Sie für die lokalen Variablen andere Bezeichner benutzen.

In der Methode `Main()` erzeugen wir wie gewohnt zuerst eine Instanz `flasche1` für die Klasse `Sherry` und initialisieren diese Instanz dann durch den Aufruf der Methode `Init()`.

Das klappt zwar alles, aber sehr viel einfacher und auch realistischer wäre es, wenn wir die Werte der Felder direkt beim Erzeugen der Instanz setzen könnten. Denn auch in der realen Welt werden ja sehr viele Eigenschaften eines Objekts direkt beim Erstellen festgelegt und nicht erst in einem zweiten separaten Schritt nach dem Erstellen. In der objektorientierten Programmierung erfolgt diese Initialisierung direkt beim Erzeugen über einen **Konstruktor**.



Ein Konstruktor ist eine spezielle Methode, die automatisch beim Erzeugen einer Instanz einer Klasse ausgeführt wird.

Die Vereinbarung eines Konstruktors ist sehr einfach. Sie müssen lediglich eine Methode mit exakt demselben Namen wie die Klasse erstellen.

Unsere Klasse `Sherry` würde mit einem Konstruktor zum Beispiel so aussehen:

```

class Sherry
{
    //die Felder
    int alter;
    int liter;

    //die Methoden
    //der Konstruktor
    public Sherry(int alter, int liter)
}

```

```

{
    this.alter = alter;
    this.liter = liter;
}

//zum Ansehen
public void Ansehen()
{
    //bitte jeweils in einer Zeile eingeben
    Console.WriteLine("Der Sherry ist {0} Jahre
alt.", alter);
    Console.WriteLine("Die Flasche enthält {0}
Liter.", liter);
}
}

```

Im Vergleich zur Methode `Init()`, die wir vorher benutzt haben, haben sich nur zwei Dinge geändert:

1. Statt `Init` benutzen wir für die Methode den Namen der Klasse selbst – also `Sherry`.
2. Der Konstruktor hat keine Typangabe mehr.

Bitte beachten Sie:



Ein Konstruktor darf keine Typangabe haben. Sie dürfen hier auch nicht das Schlüsselwort `void` benutzen, um anzugeben, dass keine Rückgabe erfolgt.

Denken Sie daran, dass auch Methoden einer Klasse ohne weitere Angaben nur privat sichtbar sind. Achten Sie daher sorgfältig darauf, den Konstruktor über `public` öffentlich zu vereinbaren. Andernfalls ist kein Zugriff von außen möglich und das Erzeugen einer Instanz scheitert.

Der Aufruf eines Konstruktors erfolgt automatisch, sobald Sie eine Instanz für die Klasse anlegen. Dabei können Sie auch wie gewohnt Argumente übergeben.

Die Anweisung

```
Sherry flasche1 = new Sherry(10, 1);
```

erzeugt also eine Instanz `flasche1` der Klasse `Sherry` und initialisiert diese Instanz mit den Werten, die in den Klammern angegeben sind – in unserem Beispiel also mit 10 für `alter` und 1 für `liter`. Die Initialisierung erfolgt dabei automatisch durch den Konstruktor.

Sehen wir uns jetzt den Einsatz eines Konstruktors in einem kompletten Code an. Dazu ersetzen wir den Aufruf der Methode `Init()` aus dem vorigen Code durch einen Konstruktor.

```
#####
Initialisierung mit einem Konstruktor
#####
using System;
```

```

namespace Cshp06d_02_02
{
    //die Klasse Sherry
    class Sherry
    {
        //die Felder
        int alter;
        int liter;

        //die Methoden
        //der Konstruktor
        public Sherry(int alter, int liter)
        {
            this.alter = alter;
            this.liter = liter;
        }

        //zum Ansehen
        public void Ansehen()
        {
            Console.WriteLine("Der Sherry ist {0} Jahre alt.", alter);
            Console.WriteLine("Die Flasche enthält {0} Liter.", liter);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Instanz flaschel erzeugen
            //die Werte werden über den Konstruktor gesetzt
            Sherry flaschel = new Sherry(10, 1);

            //die Werte ausgeben
            flaschel.Ansehen();
        }
    }
}

```

Code 2.2: Initialisierung mit einem Konstruktor

Bitte beachten Sie, dass Sie die Initialisierung einer Instanz über einen Konstruktor nur ein einziges Mal durchführen können – nämlich beim Erstellen der Instanz. Anders als im Code 2.1, in dem Sie die Methode `Init()` auch mehrfach aufrufen können, um die Werte der Instanz zu ändern, ist die Änderung der Werte einer Instanz über den mehrfachen Aufruf des Konstruktors im vorigen Code nicht möglich. Denn jeder Aufruf des Konstruktors würde eine neue Instanz der Klasse erzeugen.

Auch die folgende Erweiterung der Methode `Main()` im vorigen Code führt zu einer Fehlermeldung:

```

static void Main(string[] args)
{
    //Instanz flaschel erzeugen
    //die Werte werden über den Konstruktor gesetzt

```

```
Sherry flaschel = new Sherry(10, 1);

//die Werte ausgeben
flaschel.Ansehen();

Sherry flaschel = new Sherry(20, 5);
...
}
```

Mit der Anweisung

```
Sherry flaschel = new Sherry(20, 5);
```

versuchen Sie, eine neue Instanz der Klasse `Sherry` mit demselben Namen wie eine bereits vorhandene Instanz zu erzeugen. Und darüber beschwert sich der Compiler völlig zu Recht.

Auch Änderungen bei der Anzahl der Argumente sind bei einem Konstruktor nicht möglich. In unserem Beispiel müssen Sie an den Konstruktor immer zwei Argumente übergeben. Der Aufruf mit einem Argument wie in der Anweisung

```
Sherry flaschel = new Sherry(20);
```

führt zu einem Fehler.



Bitte beachten Sie:

C# erstellt für jede Klasse, die Sie selbst vereinbaren, automatisch einen **Standardkonstruktor** mit einer leeren Parameterliste und einem leeren Anweisungsblock. Er wird intern angelegt und ist daher im Quelltext nicht sichtbar.

Den Standardkonstruktor müssen Sie beim Anlegen einer Instanz benutzen, wenn Sie selbst keinen eigenen Konstruktor erstellt haben. Dazu setzen Sie einfach hinter den Namen der Klasse auf der rechten Seite des Ausdrucks die Klammern () .

Sobald Sie allerdings einen eigenen Konstruktor erstellen, wird der Standardkonstruktor nicht mehr automatisch angelegt. Damit wäre dann im vorigen Code auch das Erzeugen einer Instanz über die Anweisung

```
Sherry flaschel = new Sherry();
```

nicht mehr möglich. Denn hier rufen Sie ja den nicht mehr vorhandenen Standardkonstruktor auf.

Im weiteren Verlauf des Studienhefts werden wir Ihnen aber noch eine Möglichkeit zeigen, wie Sie solche Probleme sehr elegant lösen können – das Überladen von Konstruktoren.

Kommen wir nun zum Destruktor.

2.2 Destruktor

Der Destruktor ist das Gegenstück zum Konstruktor. Er wird bei der Zerstörung einer Instanz aufgerufen. Der etwas martialische Begriff „Zerstörung“ meint einfach, dass eine Instanz nicht mehr existiert – zum Beispiel, wenn der Speicher durch die Garbage Collection freigegeben wird.



Der Destruktor ist eine spezielle Methode, die beim Zerstören einer Instanz aufgerufen wird.

Zur Auffrischung:

Da C# die Speicherverwaltung automatisch erledigt, müssen Sie sich um das Zerstören nur in seltenen Ausnahmefällen selbst kümmern. Normalerweise werden alle Aufräumarbeiten automatisch durch die Garbage Collection durchgeführt.

Ein Destruktor hat – genau wie der Konstruktor – denselben Namen wie die Klasse und zusätzlich eine vorangestellte Tilde ~. Er hat keinen Rückgabetyp und kann auch keine Parameter verarbeiten.

Der Destruktor für die Klasse `Sherry` müsste also die folgende Form haben:

```
~Sherry()
{
    Anweisungen;
}
```

Schauen wir uns den Einsatz eines Destruktors an einem Beispiel an. Im folgenden Code erzeugen wir eine Instanz der Klasse `Sherry`, der wir nach der Ausgabe der Werte `null` zuweisen. Anschließend rufen wir den Garbage Collector per Hand auf. Was genau bei den einzelnen Anweisungen geschieht, erklären wir Ihnen nach dem Code.

```
/*#####
Ein Destruktor
#####*/
using System;

namespace Cshp06d_02_03
{
    //die Klasse Sherry
    class Sherry
    {
        //die Felder
        int alter;
        int liter;

        //die Methoden
        //der Konstruktor
        public Sherry(int alter, int liter)
        {
            this.alter = alter;
        }
    }
}
```

```

        this.liter = liter;
    }

    //zum Ansehen
    public void Ansehen()
    {
        Console.WriteLine("Der Sherry ist {0} Jahre alt.", alter);
        Console.WriteLine("Die Flasche enthält {0} Liter.",
        liter);
    }

    //der Destruktor
    ~Sherry()
    {
        //bitte in einer Zeile eingeben
        Console.WriteLine("Eine Instanz von {0} wurde
        zerstört.", this.GetType());
    }
}

class Program
{
    static void Main(string[] args)
    {
        //Instanz flaschel erzeugen
        //die Werte werden über den Konstruktor gesetzt
        Sherry flaschel = new Sherry(10, 1);

        //die Werte ausgeben
        flaschel.Ansehen();
        //flaschel auf null setzen, damit sie vom
        //Garbage Collector aufgeräumt wird
        flaschel = null;
        //den Garbage Collector per Hand aufrufen
        GC.Collect();
    }
}
}

```

Code 2.3: Ein Destruktor

In dem Code erzeugen wir zunächst eine Instanz der Klasse `Sherry` und lassen die Werte der Felder über die Methode `Ansehen()` ausgeben. Dabei gibt es keinerlei Besonderheiten.

Danach weisen wir mit der Anweisung

```
flaschel = null;
```

der Instanz `flaschel` den Wert `null` zu und überschreiben so die Referenz auf das ursprüngliche Objekt. Nach dieser Zuweisung weiß der Garbage Collector, dass der Speicher für diese Instanz freigegeben werden kann.

Anschließend starten wir den Garbage Collector mit der Anweisung

```
GC.Collect();
```

Im Destruktor lassen wir dann über die Methode `this.GetType()` den Namen der Klasse ausgeben, für die der Destruktor ausgeführt wurde.

Hinweis:

Die Ausgabe des Klassennamens erfolgt im vorigen Code durch die Methode `GetType()` in der ersten Ausgabeanweisung. Diese Methode kennen Sie ja bereits von der Ausgabe der Liste mit Datentypen. In dem vorigen Code beschaffen wir uns über die Methode den Namen der Klasse, zu der die Instanz gehört.

In der Regel sollten Sie die komplette Speicherverwaltung aber C# selber überlassen und auch, wann immer möglich, auf Destruktoren verzichten. Denn so komfortabel die Garbage Collection ist, sie hat auch einige Tücken. So wissen Sie zum Beispiel nie genau, wann sie vom System ausgeführt wird. Damit ist auch nicht klar, wann ein Destruktor ausgeführt wird. Das können Sie in dem vorigen Code ganz einfach selbst ausprobieren, indem Sie nach dem Starten des Garbage Collectors noch eine Ausgabeanweisung einfügen. Diese Ausgabe wird unter Umständen noch vor dem Destruktor ausgeführt.

Auch auf den manuellen Start der Garbage Collection sollten Sie nach Möglichkeit verzichten. Denn Sie können ja auch nicht vorhersagen, welche Operationen genau ausgeführt werden. Wenn zum Beispiel sehr viele Aufräumarbeiten erforderlich sind, kann der Prozess ein wenig dauern und unter Umständen zu Wartezeiten in Ihrem Programm führen.

Das Beispiel im vorigen Code dient daher auch nur zur Demonstration.

Zusammenfassung

Ein Konstruktor ist eine Methode, die automatisch beim Erzeugen einer Instanz einer Klasse aufgerufen wird.

Ein Destruktor ist eine Methode, die automatisch beim Zerstören einer Instanz ausgeführt wird.

Konstruktor und Destruktor haben keine Rückgabetypen.

Ein Konstruktor kann Parameter verarbeiten, ein Destruktor nicht.

Aufgaben zur Selbstüberprüfung

- 2.1 Was ist der wichtigste Unterschied zwischen einem Konstruktor und einem Destruktor?

- 2.2 Was ist ein Standardkonstruktor?

- 2.3 Was geschieht mit dem Standardkonstruktor, wenn Sie einen eigenen Konstruktor erstellen?

- 2.4 Warum sollten Sie beim Einsatz eines Destruktors vorsichtig sein?

3 Überladen

In diesem Kapitel werden wir uns mit einigen Feinheiten im Umgang mit Konstruktoren beschäftigen. Sie erfahren unter anderem, wie Sie mehrere Konstruktoren mit demselben Namen und unterschiedlichen Parameterlisten einsetzen.

3.1 Überladen von Methoden

Beim Programmieren stehen Sie häufiger vor dem Problem, dass Sie sehr ähnliche Methoden benötigen, die sich nur in einigen Teilen unterscheiden. Nehmen wir ein recht einfaches Beispiel:

Sie haben eine Methode erstellt, die zwei Zahlen miteinander multipliziert und dann das Ergebnis auf dem Bildschirm ausgibt. Diese Methode nennen Sie `Rechnung()`. Nun wollen Sie eine sehr ähnliche Methode erstellen, die aber drei Zahlen miteinander multiplizieren soll. Den Namen `Rechnung()` könnten Sie für diese Methode nicht mehr verwenden, da er ja bereits vergeben ist. Also nennen Sie die Methode mit den drei Zahlen `Rechnung3()`. Damit Sie die beiden Methoden noch besser unterscheiden können, taufen Sie die erste Methode zur Multiplikation der beiden Zahlen in `Rechnung2()` um. So könnten Sie auf einen Blick sehen, welche Methode wie viele Zahlen multipliziert.

Nach diesem Muster könnten Sie dann auch weitere Methoden erstellen, die vier oder fünf Zahlen multiplizieren. Die Methode für vier Zahlen nennen Sie `Rechnung4()` und die Methode für fünf Zahlen `Rechnung5()`.

Spätestens aber, wenn Sie beschließen, noch weitere Methoden zu erstellen, die Werte mit verschiedenen Datentypen miteinander multiplizieren können, wird diese Form der Namensvergabe sehr unübersichtlich. Außerdem ist das Erstellen der verschiedenen Methoden ja sehr mühselig und auch aufwendig – zumal sich die Anweisungen nur in Details unterscheiden.

C# bietet Ihnen hier einen sehr eleganten Ausweg – das **Überladen**.



Beim Überladen können Sie in einer Klasse mehrere Methoden mit demselben Namen erstellen.

Jetzt fragen Sie sich: „Wie werden denn die Methoden unterschieden, wenn die Namen identisch sind?“ Die Antwort ist recht einfach: Die Methoden müssen unterschiedliche Parameterlisten haben. Anhand der Parameterlisten kann der Compiler dann erkennen, welche Methode aufgerufen werden soll.



Überladene Methoden müssen sich durch die Parameterlisten eindeutig voneinander unterscheiden.

Möglich wären zum Beispiel folgende Vereinbarungen:

```
public int Rechnung(int wert1, int wert2) ...
public int Rechnung(int wert1, int wert2, int wert3) ...
```

Bei der ersten Methode `Rechnung()` werden zwei Parameter vom Typ `int` übergeben, bei der zweiten Methode `Rechnung()` dagegen drei Parameter vom Typ `int`. Den Aufruf

```
Rechnung(1, 2);
```

könnte der Compiler dann eindeutig der Methode `Rechnung()` mit den zwei Parametern zuordnen und den Aufruf

```
Rechnung(1, 2, 3);
```

der Methode mit den drei Parametern.

Bitte beachten Sie, dass unterschiedliche Rückgabetypen für überladene Methoden nicht ausreichen. Die folgenden Vereinbarungen wären zum Beispiel nicht möglich:

```
public int Rechnung(int wert1, int wert2) ...
public float Rechnung(int wert1, int wert2) ...
```

Hier kann der Compiler beim Aufruf

```
Rechnung(1, 2);
```

nicht eindeutig erkennen, welche der beiden Methoden denn nun gemeint ist.

Wenn Sie beim Überladen die gleiche Anzahl Parameter mit unterschiedlichen Typen verwenden, muss durch die Argumente beim Aufruf eindeutig klar sein, welche Methode angesprochen werden soll. Dazu müssen Sie unter Umständen beim Aufruf der Methode explizit den Typ der Argumente angeben.

Bei den Vereinbarungen

```
public int Rechnung(int wert1, int wert2);
public float Rechnung(float wert1, float wert2);
```

führt der Aufruf

```
Rechnung(1.2, 2.1);
```

zu einer Beschwerde des Compilers, da er die Werte 1,2 und 2,1 als `double`-Typen betrachtet und keine entsprechende Methode vorhanden ist.

Zur Erinnerung:

Alle konstanten Gleitkommazahlen werden von C# als `double`-Typen interpretiert
– auch dann, wenn eigentlich der Typ `float` ausreichen würde.



Erst der Aufruf mit einem expliziten Typecasting der Argumente lässt diese Fehlermeldung verschwinden. Richtig wäre also folgender Aufruf der Methode:

```
Rechnung((float)1.2, (float)2.1);
```

oder etwas kompakter

```
Rechnung(1.2F, 2.1F);
```

**Noch einmal zur Erinnerung:**

Über das explizite Typecasting können Sie den Compiler anweisen, einen Wert als einen bestimmten Datentyp zu betrachten.

Die verschiedenen Möglichkeiten des Überladens für Methoden finden Sie noch einmal zusammengefasst im folgenden Code. Damit Sie besser nachvollziehen können, woher die verschiedenen Ausgaben stammen, haben wir in die Methoden entsprechende Meldungen eingefügt.

```
/*#####
/* Beispiele für überladene Methoden
#####
using System;

namespace Cshp06d_03_01
{
    //die Klasse Produkt
    //benutzt werden überladene Klassenmethoden
    class Produkt
    {
        //mit zwei int-Parametern
        public static int Rechnung(int wert1, int wert2)
        {
            Console.WriteLine("Rechnung mit zwei Parametern");
            return (wert1 * wert2);
        }

        //mit drei int-Parametern
        public static int Rechnung(int wert1, int wert2, int wert3)
        {
            Console.WriteLine("Rechnung mit drei Parametern");
            return (wert1 * wert2 * wert3);
        }

        //mit zwei float-Parametern
        public static float Rechnung(float wert1, float wert2)
        {
            Console.WriteLine("Rechnung mit zwei float-Parametern");
            return (wert1 * wert2);
        }

        //diese Methode wäre nicht zulässig, da sie sich
        //nur im Rückgabetyp von einer bereits vorhandenen
        //unterscheidet
        /*
        public static float Rechnung(int wert1, int wert2)
        {
            return (wert1 * wert2);
        }
        */
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        float wert1 = 10.02F;
        float wert2 = 10.09F;
        //Aufruf der überladenen Methoden
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Der Aufruf mit zwei int-
Argumenten: {0}", Produkt.Rechnung(10, 20));
        Console.WriteLine("Der Aufruf mit drei int-
Argumenten: {0}", Produkt.Rechnung(10, 20, 30));
        Console.WriteLine("Der Aufruf mit zwei float-
Argumenten: {0}", Produkt.
Rechnung((float)10.5,(float)20.5));
        //folgender Aufruf geht nicht, da es keine
        //Methode für die Verarbeitung von double-
        //Typen gibt
        //Console.WriteLine("Der Aufruf mit zwei
        //double-Argumenten: {0}", Produkt.Rechnung(10.5, 20.5));
        //so geht es, weil es sich um zwei float-
        //Argumente handelt
        Console.WriteLine("Der Aufruf mit zwei float-
Argumenten: {0}", Produkt.Rechnung(wert1, wert2));
    }
}
```

Code 3.1: Beispiele für überladene Methoden

Zur Auffrischung:

Klassenmethoden müssen Sie ohne eine Instanz direkt über den Namen der Klasse aufrufen.



3.2 Überladen von Konstruktoren

Das Überladen funktioniert nun nicht nur für Klassenmethoden, sondern auch für nahezu alle anderen Methoden. Besonders interessant ist es in Verbindung mit Konstruktoren. Wie Sie ja bereits wissen, wird der Name eines Konstruktors fest durch den Namen der Klasse vorgegeben. Einen alternativen Konstruktor mit einem abweichenden Namen können Sie also gar nicht vereinbaren.

Trotzdem können Sie auch bei Konstruktoren verschiedene Methoden für unterschiedliche Zwecke verwenden – eben durch das Überladen des Konstruktors. Die Technik entspricht dabei dem „normalen“ Überladen einer Methode: Sie geben den Konstruktor mehrfach mit verschiedenen eindeutigen Parameterlisten an. Anhand der Argumente, die Sie beim Erzeugen einer Instanz angegeben, kann der Compiler dann selbst feststellen, welchen Konstruktor er verwenden muss.

Schauen wir uns auch das Überladen von Konstruktoren an einem praktischen Beispiel an. In den folgenden Zeilen werden für unsere Klasse `Sherry` insgesamt drei Konstruktoren vereinbart:

```
public Sherry(int alter, int liter) ...
public Sherry(int alter) ...
public Sherry() ...
```

Der erste Konstruktor `Sherry(int alter, int liter)` arbeitet mit zwei Parametern für das Alter und die Liter. Beim zweiten Konstruktor `Sherry(int alter)` wird nur ein `int`-Wert für das Alter übergeben und mit dem dritten Konstruktor `Sherry()` schließlich erstellen wir einen neuen, eigenen Standardkonstruktor.



Zur Erinnerung:

Ein Standardkonstruktor ist ein Konstruktor mit leerer Parameterliste. Er wird automatisch für eine Klasse erstellt, aber wieder gelöscht, sobald Sie einen eigenen Konstruktor anlegen.

```
/*#####
Überladene Konstruktoren
#####*/
using System;

namespace Cshp06d_03_02
{
    //die Klasse Sherry
    class Sherry
    {
        //die Felder
        int alter;
        int liter;

        //die Methoden
        //der Konstruktor mit zwei Parametern
        public Sherry(int alter, int liter)
        {
            this.alter = alter;
            this.liter = liter;
        }

        //der Konstruktor mit einem Parameter
        public Sherry(int alter)
        {
            this.alter = alter;
            //hier könnte this auch entfallen, da es keine
            //gleichnamige lokale Variable mehr gibt
            this.liter = 1;
        }
    }
}
```

```
//der neue Standardkonstruktor
public Sherry()
{
    //this könnte hier ebenfalls entfallen
    this.alter = 10;
    this.liter = 1;
}

//zum Ansehen
public void Ansehen()
{
    Console.WriteLine("Der Sherry ist {0} Jahre alt.", alter);
    Console.WriteLine("Die Flasche enthält {0} Liter.",
                      liter);
}
}

class Program
{
    static void Main(string[] args)
    {
        //verschiedene Instanzen mit unterschiedlichen
        //Konstruktoren erzeugen
        Sherry flasche1 = new Sherry(20, 2);
        Sherry flasche2 = new Sherry(15);
        Sherry flasche3 = new Sherry();

        //die Werte ausgeben
        flasche1.Ansehen();
        flasche2.Ansehen();
        flasche3.Ansehen();
    }
}
```

Code 3.2: Ein Beispiel für überladene Konstruktoren

In den Methoden der Konstruktoren werden abhängig von der Anzahl der übergebenen Argumente die Werte für die Felder `alter` und `liter` entweder auf feste Werte gesetzt oder auf die Werte, die als Argumente übergeben wurden.

Beim Konstruktor `Sherry(int alter, int liter)` werden sowohl das Alter als auch die Liter über die Argumente gesetzt. Beim Konstruktor `Sherry(int alter)` wird der Wert für das Alter über das Argument gesetzt, die Liter dagegen fest auf 1. Der letzte Konstruktor `Sherry()` – der Standardkonstruktor – schließlich setzt beide Werte durch direkte Zuweisungen im Konstruktor selbst.

In der Methode `Main()` erzeugen wir dann drei Instanzen für die Klasse `Sherry`. Bei jeder Instanz wird eine andere Argumentliste benutzt und damit auch jedes Mal ein anderer Konstruktor aufgerufen.

**Bitte denken Sie daran:**

Der Standardkonstruktor für eine Klasse wird zwar automatisch angelegt, aber wieder entfernt, wenn Sie selbst einen Konstruktor erstellen. Wenn Sie bei eigenen Konstruktoren einen Standardkonstruktor einsetzen wollen, müssen Sie ihn also wieder neu programmieren.

So viel erst einmal zum Überladen von Methoden und Konstruktoren. Im nächsten Kapitel werden wir uns mit der Vererbung beschäftigen und dabei auch noch einmal auf die Konstruktoren zurückkommen.

Zusammenfassung

Durch das Überladen können Sie mehrere Methoden mit demselben Namen und demselben Bezugsrahmen erstellen.

Konstruktoren können ebenfalls überladen werden.

Aufgaben zur Selbstüberprüfung

- 3.1 Wodurch müssen sich überladene Methoden eindeutig voneinander unterscheiden?

- 3.2 Wie legen Sie unterschiedliche Konstruktoren für ein und dieselbe Klasse an?

- 3.3 Sehen Sie sich bitte die beiden folgenden Methodenköpfe an. Die Methode `MethodeA()` soll überladen werden. Ist das in dieser Form möglich? Begründen Sie bitte kurz Ihre Antwort.

```
public int MethodeA(int a) ...
public double MethodeA(int a) ...
```

- 3.4 Sie haben eine Methode `MethodeB()` wie folgt überladen:

```
public int MethodeB(int b) ...
public float MethodeB(float b) ...
```

Welche der beiden Methoden wird durch die Anweisung

```
MethodeB(1.2);
```

aufgerufen? Falls kein Aufruf möglich ist: Wie müssen Sie den Aufruf ändern?

4 Vererbung

In diesem Kapitel beschäftigen wir uns mit der Vererbung. Zunächst stellen wir Ihnen das Konzept der Vererbung vor. Dann lernen Sie, wie Sie die Vererbung in C# umsetzen und welche Besonderheiten Sie dabei beachten müssen.

4.1 Das Konzept der Vererbung

Mit der Vererbung haben wir uns schon kurz beim Thema „Klassenhierarchie und Ableitung“ beschäftigt – wahrscheinlich, ohne dass es Ihnen bewusst war.

Wenn Sie noch einmal zurückdenken, können Klassen in der Klassenhierarchie Attribute und auch Methoden von anderen Klassen übernehmen und erweitern. Dieser Vorgang wird **Vererbung** genannt.



Die Vererbung ist ein wesentliches Konzept der Objektorientierung.

Statt ähnliche Attribute oder ähnliche Methoden immer wieder neu abbilden zu müssen, werden die Attribute und Methoden aus einer übergeordneten Klasse an eine untergeordnete Klasse vererbt und können dort erweitert oder angepasst werden. Damit lassen sich einmal erstellte Klassen sehr gut wiederverwenden, indem sie als Grundlage für neue Klassen dienen.

Die Vererbung erfolgt nun nicht willkürlich, sondern nach festen Regeln:

- Eine Klasse kann alle Attribute und Methoden von einer Klasse erben, die in der Klassenhierarchie unmittelbar über ihr liegt. Die Klasse, die in der Klassenhierarchie unmittelbar über einer Klasse liegt, wird auch **unmittelbare Oberklasse** oder **superclass³** genannt.

Oder anders herum ausgedrückt:

- Die Weitergabe von Attributen und Methoden ist nur an die Klasse möglich, die in der Klassenhierarchie unmittelbar unter der Klasse liegt. Diese Klasse wird auch **unmittelbare Unterklasse** oder **subclass⁴** genannt.



Für die Vererbungsbeziehung werden auch die Begriffe **Generalisierung** und **Spezialisierung** verwendet. Eine übergeordnete Klasse **generalisiert** eine untergeordnete Klasse – sie verallgemeinert die untergeordnete Klasse. Eine untergeordnete Klasse **spezialisiert** eine übergeordnete Klasse – sie erweitert die übergeordnete Klasse.

Schauen wir uns die Vererbung noch einmal am Beispiel der Autos an.

Sie erstellen zuerst eine Klasse „Auto“, die ganz oben in der Klassenhierarchie steht. In dieser Klasse werden die Eigenschaften beschrieben, die für alle Autos gelten – also zum Beispiel die Geschwindigkeit und die Position.

3. *Superclass* stammt aus dem Englischen und bedeutet frei übersetzt „Oberklasse“.

4. *Subclass* stammt ebenfalls aus dem Englischen und bedeutet frei übersetzt „Unterklasse“.

Unterhalb dieser Klasse legen Sie dann zwei neue Klassen an: eine Klasse für Sportwagen und eine Klasse für Geländewagen. In diesen Klassen werden dann nur noch die typischen Eigenschaften der jeweiligen Klassen beschrieben – zum Beispiel die Motorisierung, die Bereifung oder die Form der Karosserie.

Die anderen allgemeingültigen Eigenschaften werden aus der übergeordneten Klasse **geerbt**. In unserem Beispiel bedeutet das: Die beiden Klassen „Sportwagen“ und „Geländewagen“ haben sowohl die Attribute als auch die Methoden der Klasse „Auto“ übernommen und zusätzlich um besondere Eigenschaften erweitert.

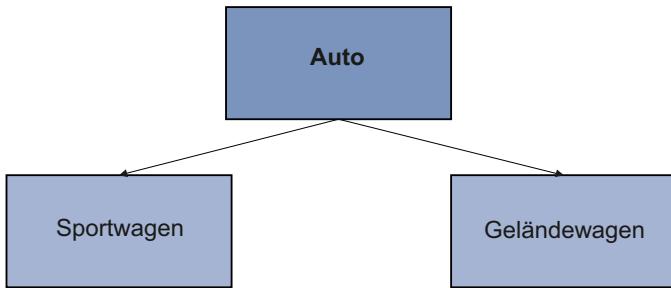


Abb. 4.1: Klassenhierarchie

Ohne die Vererbung müssten Sie für die beiden untergeordneten Klassen „Sportwagen“ und „Geländewagen“ immer auch die Eigenschaften aus der übergeordneten Klasse „Auto“ komplett neu erstellen. Sie hätten damit die dreifache Arbeit.

Bitte beachten Sie:

Konstruktoren aus einer übergeordneten Klasse werden nicht an eine abgeleitete Klasse weitergegeben. Sie können die Konstruktoren aber trotzdem einsetzen. Wie das geht, erfahren Sie später.



Nun bringt das einfache Weitergeben von Attributen und Methoden noch keinen besonderen Vorteil. Sie könnten hier auch einfach Kopien im Quelltext erstellen, um dieselbe Wirkung zu erzielen. Ein wesentlicher Bestandteil der Vererbung ist daher die Veränderung und die Erweiterung der Attribute und Methoden, die eine Unterkelassee erhält. Das heißt, Sie können allgemeine Eigenschaften in einer Oberklasse vereinbaren und dann in den Unterklassen weitere spezifische Eigenschaften ergänzen. Auf diese Weise erstellen Sie einmal Quellcode in den Oberklassen, den Sie dann in den untergeordneten Klassen wiederverwenden und gleichzeitig anpassen beziehungsweise erweitern können.

Dieses Vorgehen findet sich ja auch bei der Klassenhierarchie der Autos wieder. Die Oberklasse „Autos“ legt allgemeine Eigenschaften fest, die dann in den Unterklassen „Sportwagen“ und „Geländewagen“ jeweils spezifisch für diese beiden Unterklassen erweitert werden.

Schauen wir uns noch ein etwas komplexeres Beispiel für eine Vererbung an.

Sie wollen einen Ausschnitt der Tierwelt über Klassen abbilden. Der Ausschnitt soll Säugetiere, Beuteltiere, Kängurus und Bären darstellen. Die Klassen könnten Sie völlig isoliert voneinander anlegen und mit entsprechenden Attributen und Methoden versehen.

Sehr viel eleganter und flexibler ist es aber, wenn Sie die vier Klassen in eine hierarchische Beziehung setzen und dann Attribute und Methoden vererben. Die Klassenhierarchie für unser Beispiel könnte dann so aussehen:

An der Spitze stehen die Säugetiere. Unter dieser Klasse lassen sich sowohl Beuteltiere als auch Kängurus und Bären zusammenfassen. Beim Känguru handelt es sich um ein Beuteltier – also ordnen wir die Klasse „Kängurus“ der Klasse „Beuteltiere“ unter. Insgesamt hat unsere Klassenhierarchie dann drei Ebenen, die sich grafisch so darstellen lassen:

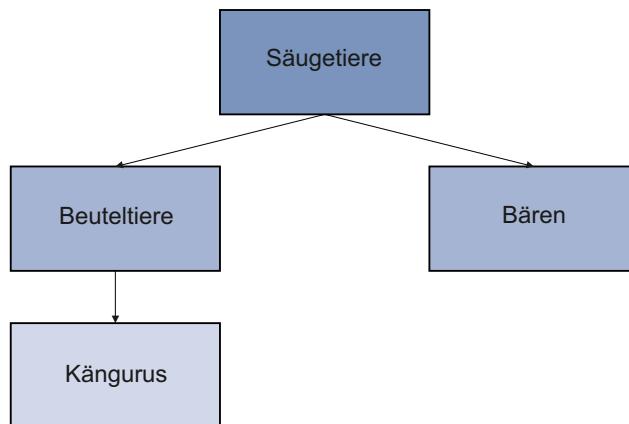


Abb. 4.2: Beispielhafte Klassenhierarchie für Säugetiere
(die Darstellung ist stark vereinfacht)

Die Vererbung der Attribute und Methoden erfolgt dann streng hierarchisch. Die Klasse „Kängurus“ kann von „Beuteltiere“ erben. „Beuteltiere“ wiederum erbt von „Säugetiere“. Damit stehen der Klasse „Kängurus“ auch alle Attribute und Methoden der Klasse „Säugetiere“ zur Verfügung.

Die Attribute und Methoden können dann schrittweise bei der Vererbung verfeinert werden. So kann die Klasse „Beuteltiere“ zum Beispiel die Attribute von „Säugetiere“ um „hat einen Beutel“ erweitern. In der Klasse „Kängurus“ können Sie dann zum Beispiel das Attribut „hat lange Beine“ vereinbaren. Wenn Sie die Attribute und Methoden für unser Känguru durch die gesamte Klassenhierarchie verfolgen, ist ein Känguru ein Säugetier, das einen Beutel hat und lange Beine.

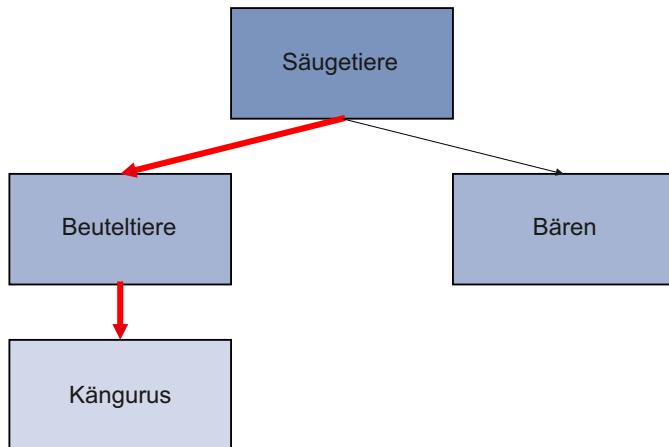


Abb. 4.3: Beispielhafte Vererbung von „Säugetiere“ auf „Kängurus“
(der „Weg“ der Vererbung ist durch rote Pfeile dargestellt)

Die Vererbung lässt sich umgangssprachlich ausdrücken als „ist ein“ – also zum Beispiel „Ein Känguru ist ein Beuteltier“ oder „Kängurus sind Beuteltiere“.



Wenn Sie Attribute oder Methoden einer übergeordneten Klasse verändern oder ergänzen, ändern sich damit automatisch auch die geerbten Attribute und Methoden der untergeordneten Klassen. Wenn Sie zum Beispiel für die Klasse „Säugetiere“ das Attribut „säugt den Nachwuchs“ ergänzen, wird dieses Attribut automatisch an die Klassen „Beuteltiere“ und „Bären“ vererbt. Da die Klasse „Beuteltiere“ an „Kängurus“ weitervererbt, hätte auch die Klasse „Kängurus“ automatisch das Attribut „säugt den Nachwuchs“.

Bitte beachten Sie, dass eine Vererbung immer streng in der Hierarchie erfolgt. In unserem Beispiel könnte die Klasse „Säugetiere“ also nicht direkt an die Klasse „Kängurus“ vererben. Die Vererbung erfolgt immer über die Klasse „Beuteltiere“.

Auch eine Vererbung zwischen Klassen auf gleicher Ebene ist nicht möglich. So kann die Klasse „Bären“ zum Beispiel nicht von der Klasse „Beuteltiere“ erben.

Für die Beschreibung von Klassen und Beziehungen zwischen Klassen können Sie zum Beispiel das Klassendiagramm der **Unified Modeling Language** (UML) verwenden. Hier werden die Klassen durch ein Rechteck dargestellt. Oben in dem Rechteck steht der Name der Klasse, darunter folgen die Attribute und Methoden. Ein stark vereinfachtes UML-Klassendiagramm für unsere Säugetiere könnte zum Beispiel so aussehen:

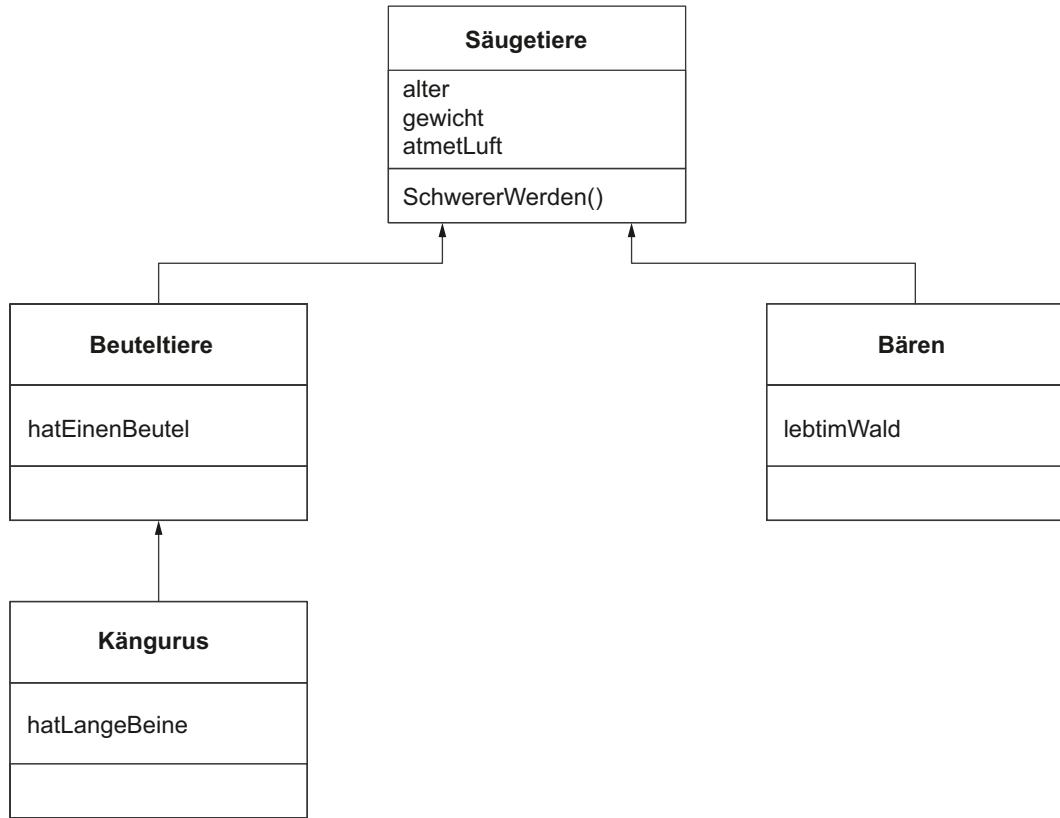


Abb. 4.4: Die Klassenhierarchie der Säugetiere als UML-Klassendiagramm

Die Linien mit dem nicht ausgefüllten Dreieck an der Spitze markieren dabei jeweils eine Vererbung. Die Spitze des Dreiecks zeigt dabei auf die Klasse, von der geerbt wird.



Mit der Modellierung von Klassen und den UML-Klassendiagrammen werden wir uns noch sehr intensiv beschäftigen, wenn wir unsere Beispielanwendung entwerfen.

Neben der einfachen Vererbung, bei der eine Klasse genau eine Oberklasse hat, kann eine Klasse auch mehrere Oberklassen haben, von denen sie erbt. Diese Form der Vererbung wird **mehrfache Vererbung** oder **Mehrfachvererbung** genannt. In unserem Beispiel könnte die Klasse „Kängurus“ zum Beispiel sowohl von der Klasse „Beuteltiere“ als auch von der Klasse „Bären“ erben. Allerdings ist die mehrfache Vererbung in unserem Beispiel unsinnig, da die Gemeinsamkeiten von Beuteltieren und Bären ja bereits durch die Klasse „Säugetiere“ abgebildet werden.



Die Mehrfachvererbung wird sehr schnell unübersichtlich und kann zu sehr unangenehmen Fehlern führen. Daher wird sie von C# **nicht** unterstützt.

So viel zum Konzept der Vererbung.

4.2 Vererbung in C#

Schauen wir uns nun an, wie die Vererbung in C# umgesetzt wird. Nehmen wir dazu wieder ein Beispiel aus der Tierwelt.

Wir vereinbaren zunächst eine Basisklasse `Baer`, die die beiden Felder `alter` und `gewicht` besitzt. Als Methoden vereinbaren wir einen Standardkonstruktor, der die beiden Felder auf Initialwerte setzt, und jeweils eine Methode für die Ausgabe des Alters und des Gewichts. Diese beiden Methoden setzen wir ein, um nicht direkt von außen auf die Felder der Klasse zugreifen zu müssen.

Der Quelltext für unsere Klasse `Baer` mit den Methoden sieht dann so aus:

```
//die Basisklasse Baer
class Baer
{
    int gewicht;
    int alter;

    //der Standardkonstruktor für Baer
    public Baer()
    {
        gewicht = 100;
        alter = 5;
    }

    //die Methode liefert das Gewicht
    public int GetGewicht()
    {
        return gewicht;
    }

    //die Methode liefert das Alter
    public int GetAlter()
    {
        return alter;
    }
}
```

Besonderheiten gibt es bei der Klasse `Baer` eigentlich nicht. Der selbst erstellte Standardkonstruktor setzt die Felder `gewicht` und `alter` auf feste Werte. Die beiden Methoden `GetGewicht()` und `GetAlter()` geben jeweils den aktuellen Wert der Felder zurück.

Von der Basisklasse `Baer` wollen wir jetzt eine Klasse `ElternBaer` ableiten. Diese Klasse soll die Felder und Methoden der Klasse `Baer` erben und zusätzlich ein Feld `anzahlKinder` besitzen, das die Anzahl der Kinder speichert.

Die Vereinbarung der Klasse `ElternBaer` sieht dann so aus:

```
//die Klasse ElternBaer erbt von der Klasse Baer
class ElternBaer : Baer
{
    int anzahlKinder;
```

```
//der Standardkonstruktor für ElternBaer
public ElternBaer()
{
    anzahlKinder = 2;
}

//die Methode liefert die Anzahl der Kinder
public int GetAnzahlKinder()
{
    return anzahlKinder;
}
```

Neu ist die erste Zeile der Vereinbarung

```
class ElternBaer : Baer
```

Hier legen wir fest, dass die Klasse `ElternBaer` von der Klasse `Baer` erben soll. Dazu geben Sie einen Doppelpunkt hinter dem Namen der Klasse an und anschließend den Namen der Klasse, von der geerbt werden soll.

In unserem Beispiel erbt also die Klasse `ElternBaer` die Methoden und Felder der Klasse `Baer` – also sowohl die Methoden `GetGewicht()` und `GetAlter()` als auch die Felder `gewicht` und `alter`. Diese Methoden und Felder müssen bei der Klassenvereinbarung der erbenden Klasse nicht noch einmal angegeben werden.



Zur Erinnerung:

Konstruktoren werden **nicht** vererbt.

Alle Felder und Methoden, die bei der Vereinbarung der Klasse `ElternBaer` aufgeführt werden, sind neue Felder und Methoden, die nur für die Klasse `ElternBaer` gelten. In unserem Beispiel sind das das Feld `anzahlKinder`, der Konstruktor und die Methode `GetAnzahlKinder()`. Bitte beachten Sie, dass diese neuen Felder und Methoden in der übergeordneten Klasse **nicht** bekannt sind. Sie könnten also in unserem Beispiel nicht direkt aus einer Methode der Klasse `Baer` auf das Feld `anzahlKinder` der Klasse `ElternBaer` zugreifen. Die Vererbung ist also – wenn Sie so wollen – eine Einbahnstraße, die nur von oben nach unten funktioniert. Im wahren Leben können ja Kinder auch keine Eigenschaften an ihre Eltern vererben, sondern nur Eigenschaften der Eltern erben.



Merken Sie sich:

Felder und Methoden, die von einer Klasse vererbt werden, müssen bei der Vereinbarung der erbenden Klasse nicht noch einmal aufgeführt werden. Bei der Vereinbarung einer abgeleiteten Klasse geben Sie nur die Felder und Methoden an, die zusätzlich für die abgeleitete Klasse gelten sollen.

Die neuen Felder und Methoden einer untergeordneten Klasse sind in der übergeordneten Klasse **nicht** bekannt.

Schauen wir uns die Vererbung jetzt an einem vollständigen Beispiel an. Im folgenden Code 4.1 erbt die Klasse ElternBaer von der Klasse Baer.

```
#####
Vererbung
#####
using System;

namespace Cshp06d_04_01
{
    //die Basisklasse Baer
    class Baer
    {
        int gewicht;
        int alter;

        //der Standardkonstruktor für Baer
        public Baer()
        {
            gewicht = 100;
            alter = 5;
        }

        //die Methode liefert das Gewicht
        public int GetGewicht()
        {
            return gewicht;
        }

        //die Methode liefert das Alter
        public int GetAlter()
        {
            return alter;
        }
    }

    //die Klasse ElternBaer erbt von der Klasse Baer
    class ElternBaer : Baer
    {
        int anzahlKinder;

        //der Standardkonstruktor für ElternBaer
        public ElternBaer()
        {
            anzahlKinder = 2;
        }

        //die Methode liefert die Anzahl der Kinder
        public int GetAnzahlKinder()
        {
            return anzahlKinder;
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        //einen Bären mit dem Standardkonstruktor erzeugen
        Baer alterBaer = new Baer();
        //einen Papabären mit dem Standardkonstruktor erzeugen
        ElternBaer papaBaer = new ElternBaer();
        //Werte über die jeweiligen Methoden anzeigen
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Der alte Bär ist {0} Jahre
alt und wiegt {1} Kilo.",
        alterBaer.GetAlter(), alterBaer.GetGewicht());
        Console.WriteLine("Der Papabär ist {0} Jahre
alt, wiegt {1} Kilo und hat {2} Kinder.",
        papaBaer.GetAlter(), papaBaer.GetGewicht(),
        papaBaer.GetAnzahlKinder());
    }
}

```

Code 4.1: Eine Vererbung

Wenn Sie den Code ausführen lassen, sollten die folgenden Ausgaben erscheinen:

Der alte Bär ist 5 Jahre alt und wiegt 100 Kilo.

Der Papabär ist 5 Jahre alt, wiegt 100 Kilo und hat 2 Kinder.

Schauen wir uns an, wie diese Werte zustande kommen.

Beim alten Bären ist die Erklärung recht einfach. Wir haben ja sowohl das Alter als auch das Gewicht im Standardkonstruktor fest zugewiesen. Auch die Anzahl der Kinder von Papabär lässt sich ohne Schwierigkeiten nachvollziehen, da wir den entsprechenden Wert ebenfalls über den Standardkonstruktor der Klasse `ElternBaer` gesetzt haben.

Woher aber hat der Papabär sein Gewicht und sein Alter? Hier ist die Erklärung ein wenig komplizierter. Beim Erzeugen einer Instanz für die Klasse `ElternBaer` muss zunächst einmal – quasi intern – eine Instanz der Klasse `Baer` erzeugt werden. Und diese Klasse `Baer` hat ja einen Standardkonstruktor, der die Werte für das Gewicht und das Alter setzt und dann – wenn Sie so wollen – an eine Instanz der abgeleiteten Klasse `ElternBaer` zurückgibt. Das heißt, beim Erzeugen einer Instanz der abgeleiteten Klasse `ElternBaer` wird auch der Konstruktor der übergeordneten Klasse `Baer` ausgeführt.



Beim Erzeugen einer Instanz einer abgeleiteten Klasse wird auch ein Konstruktor der übergeordneten Klasse ausgeführt.

Da wir in unserem Beispiel die Werte der Felder fest über die jeweiligen Standardkonstruktoren setzen, würden alle unsere Bären und auch Elternbären immer dasselbe Gewicht, dasselbe Alter und auch dieselbe Anzahl Kinder haben. Besonders brauchbar ist das natürlich nicht.

Etwas flexibler wäre das Programm, wenn Sie im Standardkonstruktor der Klasse ElternBaer neue Werte für die Felder gewicht und alter zuweisen. Damit würden Sie die Zuweisungen aus dem Standardkonstruktor der übergeordneten Klasse Baer also direkt wieder überschreiben. Das sollte problemlos gehen, da die Felder ja in der abgeleiteten Klasse ebenfalls verfügbar sind und der Standardkonstruktor der übergeordneten Klasse Baer auch vor dem Standardkonstruktor der abgeleiteten Klasse ElternBaer ausgeführt wird.

Der geänderte Konstruktor der Klasse ElternBaer könnte dann zum Beispiel so aussehen:

```
//der geänderte Standardkonstruktor
public ElternBaer()
{
    gewicht = 300;
    alter = 5;
    anzahlKinder = 2;
}
```

Nun würde jede Instanz der Klasse ElternBaer 300 Kilo wiegen, 5 Jahre alt sein und 2 Kinder haben.

Hinweis:

Ohne weitere Änderungen funktioniert der direkte Zugriff auf die Felder aus der Klasse ElternBaer nicht. Denn die Felder gewicht und alter sind in der Klasse Baer privat vereinbart und stehen damit auch nur der Klasse Baer zur Verfügung. Eine direkte Zuweisung im Konstruktor der Klasse ElternBaer ist damit nicht möglich. Warum das so ist, erfahren Sie im nächsten Kapitel.

Sehr viel eleganter und auch flexibler als die feste Zuweisung von Werten für die Felder wäre es aber, wenn wir eigene Konstruktoren erstellen und die Werte für Alter, Gewicht und Kinder als Argumente übergeben.

Das könnte dann zum Beispiel so aussehen:

```
#####
Vererbung ohne Standardkonstruktoren
Der Code lässt sich nicht übersetzen
#####

using System;

namespace Cshp06d_04_02
{
    //die Basisklasse Baer
    class Baer
    {
        int gewicht;
        int alter;
```

```
public Baer(int gewicht, int alter)
{
    this.gewicht = gewicht;
    this.alter = alter;
}

//die Methode liefert das Gewicht
public int GetGewicht()
{
    return gewicht;
}

//die Methode liefert das Alter
public int GetAlter()
{
    return alter;
}

//die Klasse ElternBaer erbt von der Klasse Baer
class ElternBaer : Baer
{
    int anzahlKinder;

    //ein eigener Konstruktor
    public ElternBaer(int gewicht, int alter, int anzahlKinder)
    {
        this.gewicht = gewicht;
        this.alter = alter;
        this.anzahlKinder = anzahlKinder;
    }

    //die Methode liefert die Anzahl der Kinder
    public int GetAnzahlKinder()
    {
        return anzahlKinder;
    }
}

class Program
{
    static void Main(string[] args)
    {
        //einen Bären erzeugen
        Baer alterBaer = new Baer(200, 3);
        //einen Papabären erzeugen
        ElternBaer papaBaer = new ElternBaer(500, 5, 20);

        //Werte über die jeweiligen Methoden anzeigen
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Der alte Bär ist {0} Jahre
alt und wiegt {1} Kilo.",
alterBaer.GetAlter(), alterBaer.GetGewicht());
```

```
        Console.WriteLine("Der Papabär ist {0} Jahre  
        alt, wiegt {1} Kilo und hat {2} Kinder.",  
        papaBaer.GetAlter(), papaBaer.GetGewicht(),  
        papaBaer.GetAnzahlKinder());  
    }  
}  
}
```

Code 4.2: Vererbung ohne Standardkonstruktoren (der Code lässt sich nicht übersetzen)

Der Code sieht auf den ersten Blick brauchbar aus, führt aber unter anderem zu einer Beschwerde des Compilers, dass kein Argument angegeben wurde, das dem formalen Parameter `gewicht` für den Konstruktor der Klasse `Baer` entspricht.

Wenn Sie noch einmal an die Aufrufe der Konstruktoren beim Erstellen einer Instanz einer abgeleiteten Klasse zurückdenken, wird auch schnell klar, was hinter dieser etwas geheimnisvollen Meldung steckt. Unsere übergeordnete Klasse `Baer` hat jetzt nur noch einen Konstruktor, der mit zwei Argumenten für das Alter und das Gewicht versorgt werden muss. Dieser Konstruktor wird aber beim Erzeugen einer Instanz der abgeleiteten Klasse `ElternBaer` nicht automatisch aufgerufen, da der Compiler ja gar nicht wissen kann, welche Werte er für die Argumente benutzen soll. Also sucht er nach einem Standardkonstruktor für die Klasse `Baer`, den es nach den Änderungen aber nicht mehr gibt.

Eine Möglichkeit wäre es nun, kurzerhand einen neuen leeren Standardkonstruktor für die Klasse `Baer` zu erstellen. Es geht aber auch sehr viel bequemer: Sie können nämlich einen Konstruktor einer übergeordneten Klasse selbst über das Schlüsselwort `base`⁵ aufrufen. Dabei können Sie auch wie gewohnt Argumente übergeben, über die dann der passende Konstruktor gefunden wird.

Der geänderte Konstruktor der Klasse `ElternBaer` mit dem Aufruf des Konstruktors der übergeordneten Klasse `Baer` würde dann so aussehen:

```
public ElternBaer(int gewicht, int alter, int anzahlKinder) :  
base(gewicht, alter)  
{  
    this.anzahlKinder = anzahlKinder;  
}
```

Code 4.3: Der Aufruf des Konstruktors der übergeordneten Klasse

Hinweis:

Da sich an dem Code nur wenige Zeilen ändern, drucken wir ihn nicht noch einmal komplett ab. Sie finden den vollständigen Code im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt `Cshp06d_04_03`.

5. `base` bedeutet übersetzt so viel wie „Basis“ oder „Grundlage“.

Das Schlüsselwort `base` wird hinter der Parameterliste des eigentlichen Konstruktors angegeben und durch einen Doppelpunkt getrennt. Damit Sie es schneller wiederfinden, haben wir den Aufruf im vorigen Code fett markiert.

**Bitte beachten Sie:**

Der Aufruf des Konstruktors der Basisklasse über `base` muss in der „Kopfzeile“ des Konstruktors der abgeleiteten Klasse erfolgen. Sie können den Aufruf nicht im Anweisungsblock des Konstruktors durchführen.

Sie dürfen die beiden Parameter `gewicht` und `alter` in der Parameterliste des Konstruktors der Klasse `ElternBaer` nicht löschen. Denn diese beiden Parameter werden über `base` an den Konstruktor der Klasse `Baer` durchgereicht. Deshalb dürfen Sie bei der Angabe der Argumente für `base` auch keinen Typ verwenden. Andernfalls versuchen Sie ja, die Argumente neu zu vereinbaren.

Ersetzen Sie jetzt einmal den Konstruktor für die Klasse `ElternBaer` aus dem Code 4.2 durch den Konstruktor mit dem Aufruf des Konstruktors der Basisklasse aus dem Code 4.3. Sie werden sehen, es funktioniert.

Wenn Sie der Aufruf des Konstruktor der Basisklasse verwirrt: Eigentlich ist das Prinzip ganz einfach. Wenn die übergeordnete Klasse einer abgeleiteten Klasse einen Konstruktor mit Parametern hat, müssen Sie diesen Konstruktor ausdrücklich über `base` im Konstruktor der abgeleiteten Klasse aufrufen und dabei auch die erforderlichen Argumente übergeben. Das ist auch dann erforderlich, wenn die übergeordnete Klasse mehrere Konuktoren hat. Hier müssen Sie ebenfalls den gewünschten Konstruktor über `base` im Konstruktor der abgeleiteten Klasse aufrufen.

Es gibt nur einen Fall, in dem der Aufruf des Konstruktors über `base` in der abgeleiteten Klasse nicht zwingend erforderlich ist – nämlich dann, wenn die übergeordnete Klasse einen Standardkonstruktor hat. Den Aufruf dieses Konstruktors ergänzt der Compiler selbst.

Merken Sie sich:

Der Compiler versucht beim Erzeugen einer Instanz einer abgeleiteten Klasse immer, den Standardkonstruktor der übergeordneten Klasse aufzurufen.



Wenn die übergeordnete Klasse keinen Standardkonstruktor hat, müssen Sie den Konstruktor der übergeordneten Klasse über `base` im Konstruktor der abgeleiteten Klasse selbst aufrufen.

Wenn eine übergeordnete Klasse neben dem Standardkonstruktor über weitere Konstruktoren verfügt, wird beim Erzeugen einer Instanz einer abgeleiteten Klasse ohne weitere Angaben immer der Standardkonstruktor aufgerufen. Wenn Sie einen anderen Konstruktor aus der übergeordneten Klasse benutzen möchten, müssen Sie ihn wieder mit `base` im Konstruktor der abgeleiteten Klasse aufrufen.

Oder etwas kompakter:

Wenn die übergeordnete Klasse einen Standardkonstruktor hat und Sie diesen Standardkonstruktor benutzen möchten, brauchen Sie im Konstruktor der abgeleiteten Klasse den Konstruktor der übergeordneten Klasse nicht selbst aufzurufen.

Hat die übergeordnete Klasse dagegen keinen Standardkonstruktor oder wollen Sie den Standardkonstruktor nicht verwenden, müssen Sie im Konstruktor der abgeleiteten Klasse den Konstruktor der übergeordneten Klasse über `base` aufrufen.

4.3 Zugriff auf geerbte Attribute

Schauen wir uns nun den Zugriff auf geerbte Attribute etwas genauer an. Dazu erweitern wir das Beispiel mit den Bären noch einmal ein wenig. Wir ergänzen in der Klasse `ElternBaer` eine Methode, die das Gewicht des Bären verändert – zum Beispiel, weil die Eltern sich Sorgen um den Nachwuchs machen und daher nicht mehr zum Essen kommen.

Die Methode an sich ist ausgesprochen einfach:

```
public void Schock()
{
    gewicht--;
}
```

Code 4.4: Die neue Methode `Schock()` für die Klasse `ElternBaer`

Hinweis:

Das vollständige Beispiel finden Sie im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **Cshp06d_04_04**.

Bei jedem Schock, den ein Elternbär erleidet, verliert er ein Kilo Gewicht.

Wenn Sie diese Methode aber jetzt so in unserem Beispiel verwenden wollen, beschwert sich der Compiler, er könne in der Methode `Schock()` nicht auf das Feld `gewicht` der Klasse `Baer` zugreifen. Und das, obwohl die Klasse `ElternBaer` das Feld `gewicht` doch eigentlich von der Klasse `Baer` geerbt hat. Die Erklärung für das Problem ist recht einfach.

Bei `gewicht` handelt es sich um ein **privates** Feld der Basisklasse `Baer`. Wie Sie wissen, stehen private Felder nur der Klasse selbst zur Verfügung. Damit ist auch der Zugriff aus abgeleiteten Klassen nicht möglich. Diese vermeintliche Einschränkung hat auch durchaus ihren Sinn, da der Zugriff aus abgeleiteten Klassen auf private Felder und Methoden der Basisklasse das Prinzip der Datenkapselung verletzen würde.

Wie lässt sich das Problem nun lösen?

Eine Möglichkeit ist es, das Feld `gewicht` kurzerhand als `public` – also als öffentlich – zu vereinbaren. Damit steht es dann den abgeleiteten Klassen zur Verfügung, aber auch sämtlichen anderen Klassen, die nichts mit der Basisklasse gemeinsam haben. Da die Datenkapselung dabei komplett verloren geht, ist diese Lösung unbrauchbar – auch wenn sie vom Compiler ohne Fehlermeldung verarbeitet wird. Lassen Sie sich aber trotzdem nicht dazu verleiten, alle Felder der Einfachheit halber als `public` zu vereinbaren.

Eine weitere Alternative wäre es, das Feld `gewicht` als Eigenschaft zu vereinbaren. Damit könnte dann der Zugriff über die `set()`- und `get()`-Methoden erfolgen. Dazu müssten wir aber die Vereinbarung der Klasse `Baer` in großen Teilen neu schreiben.

Es gibt noch eine dritte Möglichkeit: Sie vereinbaren das Feld `gewicht` so, dass es der Klasse selbst und allen abgeleiteten Klassen zur Verfügung steht. Dazu verwenden Sie das Schlüsselwort `protected`⁶.



Mit `protected` vereinbaren Sie geschützte Felder und Methoden, auf die nur von der Klasse selbst und von abgeleiteten Klassen aus zugegriffen werden kann. Alle anderen Klassen haben keinen Zugriff auf geschützte Elemente. Die Vereinbarung als `protected` ist damit ein Mittelweg zwischen den Sichtbarkeiten `private` und `public`.

Damit auch die abgeleitete Klasse `ElternBaer` auf das Feld `gewicht` der Basisklasse `Baer` zugreifen kann, müssen Sie lediglich die Vereinbarung der Klasse `Baer` ein wenig überarbeiten. Sie könnte dann zum Beispiel so aussehen:

```
//die Basisklasse Baer
class Baer
{
    protected int gewicht;
    int alter;

    ...
}
```

Code 4.5: Ein geschütztes Feld (es handelt sich um ein Fragment)

6. `Protected` bedeutet übersetzt so viel wie „geschützt“.

Nehmen Sie die entsprechenden Änderungen jetzt bitte vor und testen Sie dann das Programm noch einmal. Nach dem Eintreten des Schocks verliert der Papabär jetzt auch wie gewünscht ein Kilogramm Gewicht.

Hinweis:

Sie finden das vollständige Beispiel im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **Cshp06d_04_05**.

Neben den Vereinbarungen als `private`, `protected` und `public` kennt C# noch weitere Sichtbarkeiten. Sie finden sie zusammengefasst in der Tab. 4.1.

Tab. 4.1: Die Sichtbarkeiten für Felder und Methoden

Vereinbarung	Zugriff durch die Klasse selbst?	Zugriff durch eine abgeleitete Klasse?	Zugriff durch eine fremde Klasse?
<code>private</code>	ja	nein	nein
<code>protected</code>	ja	ja	nein
<code>private protected</code>	ja	ja, wenn sie sich im selben Projekt befindet	ja, wenn sie sich im selben Projekt befindet
<code>internal</code>	ja	ja, wenn sie sich im selben Projekt befindet	ja, wenn sie sich im selben Projekt befindet
<code>protected internal</code>	ja	ja	ja, wenn sie sich im selben Projekt befindet
<code>public</code>	ja	ja	ja

So viel an dieser Stelle zur Vererbung. Im nächsten Kapitel werden wir uns noch mit einigen Besonderheiten wie dem Polymorphismus beschäftigen.

Vielleicht sollten Sie sich aber nach dem Lösen der Aufgaben zur Selbstüberprüfung für dieses Kapitel erst einmal eine kurze Pause gönnen, damit sich das neue Wissen setzen kann.

Zusammenfassung

Mit der Vererbung können aus bereits existierenden Klassen neue Klassen abgeleitet werden. Die abgeleiteten Klassen erben dabei die Methoden, Eigenschaften und Felder der Basisklasse.

Für die Vererbung gibt es feste Regeln. So kann eine Klasse zum Beispiel nur von der Klasse erben, die sich in der Klassenhierarchie unmittelbar über ihr befindet.

Um dem Compiler mitzuteilen, dass eine Klasse von einer anderen Klasse abgeleitet werden soll, geben Sie hinter dem Namen der Klasse einen Doppelpunkt und den Namen der Basisklasse an.

Wenn die Basisklasse keinen Standardkonstruktor hat, müssen Sie den Konstruktor der Basisklasse ausdrücklich selbst über den Konstruktor der abgeleiteten Klasse aufrufen.

Über das Schlüsselwort `protected` vereinbaren Sie Felder und Methoden einer Klasse als geschützt.

Aufgaben zur Selbstüberprüfung

- 4.1 Was ist eine Generalisierung? Was ist eine Spezialisierung?

- 4.2 Welche Methoden werden bei der Vererbung **nicht** an eine abgeleitete Klasse weitergegeben?

- 4.3 Vereinbaren Sie eine Klasse `Hund`, die von der Klasse `Haustiere` abgeleitet ist. Sie müssen lediglich die erste Zeile der Vereinbarung angeben.

5 Polymorphismus und überschriebene Methoden

Sie kennen jetzt ein wesentliches Grundprinzip der objektorientierten Programmierung: die Vererbung. In diesem Kapitel wollen wir uns mit einem weiteren Grundprinzip beschäftigen – dem Polymorphismus.

Schauen wir uns zuerst an, was sich hinter diesem geheimnisvollen Begriff verbirgt.

5.1 Ein wenig Theorie

Denken Sie noch einmal an die Einführung in die Objektorientierung zurück. Ein wesentlicher Anspruch der Objektorientierung ist es, die Trennung zwischen Daten und Verhalten aufzuheben und so menschliche Sichtweisen besser abbilden zu können.

Neben der Verknüpfung von Daten und Verhalten eines Objekts weist das menschliche Denken noch eine weitere besondere Leistung auf: Begriffe können kontextabhängig angewendet werden. Das bedeutet, ein und derselbe Begriff kann in zwei verschiedenen Situationen völlig unterschiedliche Bedeutungen haben und muss korrekt interpretiert werden.

Einige Beispiele:

Im Restaurant meint das Wort „Bedienung“ zum Beispiel in der Regel den Service, bei einem Computerprogramm oder einem Gerät aber die Handhabung. Die Aussage „Die Bedienung ist schlecht“ hat damit – abhängig vom Kontext, in dem sie getroffen wird – ebenfalls völlig unterschiedliche Bedeutung.

Wenn Sie diese Aussage in einem Restaurant treffen, geht ein Gesprächspartner davon aus, dass Sie den schlechten Service meinen.

Unterhalten Sie sich dagegen zum Beispiel über ein bestimmtes Computerprogramm, ist durch das Umfeld klar, dass der Umgang mit diesem Programm gemeint ist.

Das Wort „sitzt“ kann ebenfalls abhängig vom Umfeld ganz andere Bedeutungen haben – zum Beispiel

- „Der Anzug sitzt.“
- „Meine Frau sitzt auf dem Stuhl.“
- „Der Gefangene sitzt.“

In der Objektorientierung – die ja versucht, menschliche Sichtweisen nachzuvollziehen – wird die Möglichkeit, gleiche Begriffe in unterschiedlichen Situationen auf verschiedene Arten einzusetzen, durch den **Polymorphismus** abgebildet.

Polymorph stammt aus dem Griechischen und bedeutet so viel wie „vielgestaltig“.



Ein Beispiel für Polymorphismus wäre eine Methode, die Zahlen oder Zeichenketten „addieren“ kann. Wenn an diese Methode ganze Zahlen als Eingabedaten übergeben werden, werden die Zahlen wie gewohnt addiert. Die Zahlen 5 und 4 würden das Ergebnis 9 liefern.

Werden dagegen Zeichenketten an die Methode übergeben, werden die Zeichenketten aneinandergehängt. Aus den Zeichenketten „Auto“ und „haus“ würde dann zum Beispiel „Autohaus“.

Ein anderes Beispiel für Polymorphismus ist das Ausführen einer Methode in Abhängigkeit von einer Klasse. Schauen wir uns dazu noch einmal das UML-Klassendiagramm für die Säugetiere an. Wir haben die Klasse „Beuteltiere“ weggelassen und eine weitere Klasse „Wale“ ergänzt.

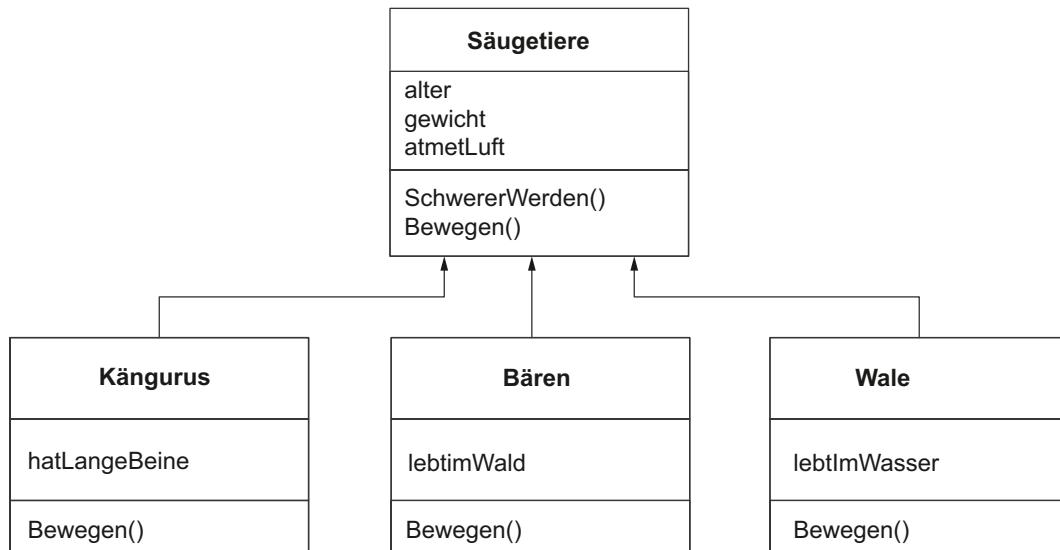


Abb. 5.1: Polymorphe Methoden

Wenn Sie die Abb. 5.1 genau betrachten, sollte Ihnen schon etwas auffallen: Alle Klassen besitzen eine eigene Methode `Bewegen()` – auch die abgeleiteten Klassen. Eigentlich wäre das ja nicht nötig, da die Methode vererbt wird – also auch in den abgeleiteten Klassen vorhanden sein sollte.

Nun bewegen sich aber Kängurus, Bären und Wale ganz unterschiedlich. Deshalb wird die Methode `Bewegen()` in der Oberklasse „Säugetiere“ nur sehr allgemein dargestellt. In den abgeleiteten Klassen „Kängurus“, „Bären“ und „Wale“ wird diese Methode dann detaillierter für die jeweilige Klasse festgelegt. Dazu wird die Methode in den abgeleiteten Klassen **überschrieben**. Wie das funktioniert, zeigen wir Ihnen gleich.

Alle Klassen unserer Säugetiere hätten dann eine eigene Methode `Bewegen()`, die aber völlig unterschiedliche Aktionen auslösen kann. Das Känguru bewegt sich hüpfend fort, der Bär in der Regel auf vier Beinen und der Wal schwimmt. Abhängig von der Klasse, über die die Methode `Bewegen()` aufgerufen wird, ändert sich damit auch das Ergebnis der Methode `Bewegen()`.

So viel zur Theorie. Schauen wir uns nun den Polymorphismus in der Praxis an.

5.2 Die praktische Umsetzung

In unseren bisherigen Codes für die Bären haben wir die Werte der Felder immer über Ausgabeanweisungen in der Methode `Main()` anzeigen lassen. Wir wollen jetzt für jede Klasse eine eigene Methode erstellen, die bei der Klasse `Baer` das Gewicht und das Alter anzeigt und bei der Klasse `ElternBaer` zusätzlich die Anzahl der Kinder. Die Methode soll in beiden Fällen `Ausgeben()` heißen. Das heißt, wir überschreiben in der abgeleiteten Klasse `ElternBaer` die Methode aus der Basisklasse `Baer`.

Damit Sie die Änderungen im Code 5.1 schneller wiederfinden, haben wir sie fett markiert. Bitte beachten Sie auch, dass Sie beide Felder der Klasse `Baer` jetzt als `protected` vereinbaren müssen. Denn wir wollen ja auch das Feld `alter` in der abgeleiteten Klasse ausgeben.

```
/*#####
Eine "überschriebene" Methode
#####
using System;

namespace Cshp06d_05_01
{
    //die Basisklasse Baer
    class Baer
    {
        //beide Felder müssen jetzt als protected
        //vereinbart werden
        protected int gewicht;
        protected int alter;

        public Baer(int gewicht, int alter)
        {
            this.gewicht = gewicht;
            this.alter = alter;
        }

        //die Methode liefert das Gewicht
        public int GetGewicht()
        {
            return gewicht;
        }

        //die Methode liefert das Alter
        public int GetAlter()
        {
            return alter;
        }

        //die Methode zur Ausgabe der Werte
        public void Ausgeben()
        {
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("Die Ausgabe erfolgt aus
{0}", this.GetType());
            Console.WriteLine("Der Bär ist {0} Jahre alt
{1} Kinder", gewicht, Kinder);
        }
}
```

```

        und wiegt {1} Kilo.", alter, gewicht);
    }
}

//die Klasse ElternBaer erbt von der Klasse Baer
class ElternBaer : Baer
{
    int anzahlKinder;

    //ein eigener Konstruktor
    //er ruft den Konstruktor der Basisklasse über base auf
    public ElternBaer(int gewicht, int alter, int
anzahlKinder) : base(gewicht, alter)
    {
        this.anzahlKinder = anzahlKinder;
    }

    //die Methode liefert die Anzahl der Kinder
    public int GetAnzahlKinder()
    {
        return anzahlKinder;
    }

    //die Methode zur Ausgabe der Werte
    //Sie "überschreibt" die gleichnamige Methode der
    //Basisklasse
    public void Ausgeben()
    {
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Die Ausgabe erfolgt aus
{0}", this.GetType());
        Console.WriteLine("Der Bär ist {0} Jahre alt,
wiegt {1} Kilo und hat {2} Kinder.", alter,
gewicht, anzahlKinder);
    }
}

class Program
{
    static void Main(string[] args)
    {
        //einen Bären erzeugen
        Baer alterBaer = new Baer(200, 3);
        //einen Papabären erzeugen
        ElternBaer papaBaer = new ElternBaer(500, 5, 20);

        //Werte über die jeweiligen Methoden anzeigen
        alterBaer.Ausgeben();
        papaBaer.Ausgeben();
    }
}
}

```

Code 5.1: Eine überschriebene Methode

Besonders spektakulär ist das Überschreiben der Methode `Ausgeben()` in der Klasse `ElternBaer` nicht. Wir vereinbaren die Methode einfach in der Klasse neu und ändern dann die Anweisungen, die ausgeführt werden sollen.

Wenn Sie den Code übersetzen und ausführen lassen, erscheinen auch wie gewünscht unterschiedliche Ausgaben – einmal aus der Methode `Ausgeben()` der Instanz `alterBaer` und einmal aus der Methode `Ausgeben()` der Instanz `papaBaer`.

Hinweis:

Beim Eingeben des Codes zeigt Ihnen Visual Studio eine Warnung an. Damit werden wir uns gleich noch ausführlicher beschäftigen.

Beim Überschreiben einer Methode wird die Methode, die ursprünglich vererbt wurde, komplett verdeckt. Sie können also nicht einfach „neue“ Anweisungen an eine geerbte Methode anhängen, sondern müssen gegebenenfalls identische Anweisungen wiederholen.

Wenn Sie den Code allerdings aufmerksam studieren und noch einmal an das Überladen von Methoden zurückdenken, sollten Sie jetzt stutzig werden: Beide Ausgabemethoden tragen denselben Namen und auch die Parameterlisten sind identisch – nämlich leer. Beim Überladen von Methoden hieß es aber, die Parameterlisten müssen unterschiedlich sein, damit der Compiler die Methoden unterscheiden kann. Trotzdem lässt sich der vorige Code übersetzen und liefert auch die korrekten Ausgaben.

Die Erklärung ist einfach: Die beiden Methoden gehören zu verschiedenen Klassen und sind damit für den Compiler eindeutig zu unterscheiden. Der vollständige Name der einen Methode ist nämlich `Baer.Ausgeben()` und der vollständige Name der anderen Methode `ElternBaer.Ausgeben()`. Die Methode wird daher auch nicht überladen, sondern überschrieben.

Beim **Überladen** von Methoden gibt es mehrere Methoden mit demselben Namen und unterschiedlichen Parameterlisten in **ein und derselben Klasse**.



Beim **Überschreiben** von Methoden gibt es mehrere Methoden mit demselben Namen und derselben Parameterliste in **unterschiedlichen Klassen**.

Allerdings erfolgt in unserem Beispiel kein „echtes“ Überschreiben der Methode, denn der Compiler ordnet die Methoden bereits bei der Übersetzung den Klassen zu – also **vor** der Ausführung des Programms. Damit ist **beim** Ausführen des Programms genau bekannt, welche Methode zu welcher Klasse gehört.

Die Zuordnung vor der Ausführung des Programms wird auch **statische Bindung** oder **frühe Bindung** genannt.



Die Methoden werden in unserem Beispiel also nicht überschrieben, sondern einfach nur verdeckt. Da dieses Verdecken in vielen Fällen versehentlich geschieht, weil ein Bezeichner mehrfach verwendet wird, erscheint auch die Warnung des Compilers. Damit sie nicht mehr erscheint, müssen Sie mit dem Schlüsselwort `new` ausdrücklich mitteilen, dass das Verdecken Absicht ist. Der Kopf der Methode `Ausgeben()` in der Klasse `ElternBaer` würde dann so aussehen:

```
public new void Ausgeben()
```

Die statische Bindung kann in einigen Fällen aber auch zu einem echten Problem werden. Schauen wir uns dazu einmal das folgende Beispiel an:

```
...
class Program
{
    static void Main(string[] args)
    {
        //einen Bären erzeugen
        Baer alterBaer = new Baer(200, 3);
        //einen Papabären erzeugen
        ElternBaer papaBaer = new ElternBaer(500, 5, 20);
        //ein Testbär
        Baer testBaer;

        //alterBaer wird testBaer zugewiesen
        testBaer = alterBaer;
        testBaer.Ausgeben();

        //jetzt wird papaBaer zugewiesen
        testBaer = papaBaer;
        testBaer.Ausgeben();
    }
}
```

Code 5.2: Probleme mit der statischen Bindung (es handelt sich um ein Fragment)

Hinweis:

Den vollständigen Code finden Sie im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **Cshp06d_05_02**.

An den beiden Klassen `Baer` und `ElternBaer` an sich haben wir nichts geändert. Neu sind lediglich einige Anweisungen in der Methode `Main()`. Hier vereinbaren wir eine Variable `testBaer` der Klasse `Baer`, der wir zuerst die Instanz `alterBaer` der Klasse `Baer` zuweisen. Anschließend rufen wir über die Variable `testBaer` die Ausgabemethode der Klasse auf. Danach weisen wir der Variablen die Instanz `papaBaer` der Klasse `ElternBaer` zu und rufen ebenfalls wieder die Ausgabemethode auf.

Hinweis:

Das Zuweisen einer Instanz der Klasse `ElternBaer` auf die Variable `testBaer` ist möglich – auch wenn die Typen der Variablen eigentlich nicht zusammenpassen. Denn bei `ElternBaer` handelt es sich ja um eine abgeleitete Klasse der Basisklasse

`Baer`. Und abgeleitete Klassen können Sie überall dort benutzen, wo Sie auch die übergeordnete Klasse verwenden können. Dieses Prinzip wird **Substitution**⁷ genannt.

Eigentlich sollte auch der vorige Code zwei unterschiedliche Ausgaben auf dem Bildschirm erzeugen. Denn vor der ersten Ausgabe wird der Variablen `testBaer` ja eine Instanz der Klasse `Baer` zugewiesen und vor der zweiten Ausgabe eine Instanz der Klasse `ElternBaer`. Wenn Sie den Code aber ausführen lassen, werden Sie eine unangenehme Überraschung erleben: Auf dem Bildschirm erscheint in beiden Fällen die Ausgabe der Methode `Ausgeben()` unserer Klasse `Baer` – und das, obwohl der Compiler die beiden Klassen scheinbar auseinanderhalten kann. Denn die Methode `GetType()` liefert bei der ersten Ausgabe die Klasse `Baer` und bei der zweiten Ausgabe die Klasse `ElternBaer`. Ausgegeben werden aber in beiden Fällen nur das Alter und das Gewicht.

Wenn Sie noch einmal an die statische Bindung zurückdenken, wird auch schnell klar, warum das Programm nicht so funktioniert wie gewollt. Denn der Compiler geht davon aus, dass über die Variable vom Typ `Baer` in beiden Fällen auch die Methode `Ausgeben()` der Klasse `Baer` aufgerufen werden soll. Ob nun über diese Variablen auch auf Instanzen einer abgeleiteten Klasse zugegriffen wird, interessiert ihn nicht weiter. Er ordnet die Methoden einmal vor der Übersetzung zu und überprüft dann den Typ beim Aufruf nicht weiter. Wenn Sie so wollen, bekommt das Programm zwar mit, dass beim zweiten Zugriff mit einem anderen Typ gearbeitet wird, nimmt aber durch die statische Bindung keine neue Zuordnung der Methoden vor.

Wir müssen also noch dafür sorgen, dass die Zuordnung der Methoden nicht mehr statisch bei der Übersetzung erfolgt, sondern dynamisch bei der Ausführung des Programms.

Die Zuordnung bei der Ausführung des Programms wird auch **dynamische Bindung** oder **späte Bindung** genannt.



5.3 Dynamische Bindung

Bevor Sie sich jetzt mit Grausen abwenden: Dynamische Bindung hört sich kompliziert an, ist aber ausgesprochen einfach. Sie müssen nichts weiter machen, als in der Vereinbarung der Basisklasse vor die entsprechende Methode das Schlüsselwort `virtual` zu setzen. Damit der Compiler weiß, dass die Methode in der abgeleiteten Klasse die Methode aus der Basisklasse überschreibt, geben Sie dort bei der Methode das Schlüsselwort `override`⁸ an.

Die geänderten Methoden in den beiden Klassen sehen dann so aus:

```
class Baer
{
    ...
    //die Methode zur Ausgabe der Werte
    //sie wird jetzt dynamisch gebunden
    public virtual void Ausgeben()
```

7. Substitution bedeutet so viel wie „Stellvertretung“ oder „Ersetzung“.

8. Wörtlich übersetzt bedeutet `override` „außer Kraft setzen“ oder „aufheben“.

```

    {
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Die Ausgabe erfolgt aus
        {0}", this.GetType());
        Console.WriteLine("Der Bär ist {0} Jahre alt und
        wiegt {1} Kilo.", alter, gewicht);
    }
}

class ElternBaer : Baer
{
    ...
    //die Methode zur Ausgabe der Werte
    //Sie überschreibt die gleichnamige Methode der
    //Basisklasse
    public override void Ausgeben()
    {
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Die Ausgabe erfolgt aus
        {0}", this.GetType());
        Console.WriteLine("Der Bär ist {0} Jahre alt,
        wiegt {1} Kilo und hat {2} Kinder.", alter,
        gewicht, anzahlKinder);
    }
}

```

Hinweis:

Den vollständigen geänderten Code finden Sie im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **Cshp06d_05_03**.

Probieren Sie diese Änderungen jetzt bitte aus. Sie werden sehen, es werden auch tatsächlich die richtigen Methoden ausgeführt.

**Bitte beachten Sie:**

Es reicht nicht aus, nur die Methode in der Basisklasse als `virtual` zu vereinbaren. Sie müssen auch die Methode in der abgeleiteten Klasse mit dem Schlüsselwort `override` markieren. Andernfalls überschreiben Sie die Methode nicht, sondern verdecken Sie lediglich. Achten Sie daher sehr genau auf mögliche Warnungen des Compilers.

Abschließend noch ein Hinweis:

Alle Klassen, die Sie selbst erstellen, erben automatisch von der Klasse `System.Object`. Sie können daher für Ihre selbst erstellten Klassen auch Methoden aus der Klasse `System.Object` verwenden oder diese Methoden in Ihren eigenen Klassen überschreiben. Diese Möglichkeit haben wir ja zum Beispiel bei der Methode `GetType()` eingesetzt.

Alternativ können Sie sich den Namen einer Klasse auch über die Methode `ToString()`⁹ aus der Klasse `System.Object` beschaffen. Die Anweisung

```
Console.WriteLine("Die Ausgabe erfolgt aus {0}",  
    this.GetType());
```

in unseren Methoden `Ausgeben()` könnte also auch so aussehen

```
Console.WriteLine("Die Ausgabe erfolgt aus {0}",  
    this.ToString());
```

Zusammenfassung

Polymorphismus bezeichnet die Fähigkeit, auf dieselben Begriffe in unterschiedlichen Situationen auf verschiedene Arten zu reagieren.

In der objektorientierten Programmierung wird der Polymorphismus unter anderem durch das Überschreiben von Methoden umgesetzt.

Um eine Methode in einer untergeordneten Klasse zu überschreiben, vereinbaren Sie sie neu. Dabei werden die Anweisungen der überschriebenen Methode verdeckt.

Bei der Zuordnung von Methoden zu Klassen wird zwischen der statischen und der dynamischen Bindung unterschieden.

Um Methoden tatsächlich zu überschreiben, vereinbaren Sie die Methoden in der Basisklasse als `virtual`. In der abgeleiteten Klasse müssen Sie durch das Schlüsselwort `override` angeben, dass die Methode eine andere Methode überschreibt.

Aufgaben zur Selbstüberprüfung

- 5.1 Was ist der Unterschied zwischen dem Überschreiben und dem Überladen einer Methode?

9. `ToString` steht für „zu Zeichenkette“.

- 5.2 Beschreiben Sie den Unterschied zwischen der statischen und der dynamischen Bindung.

- 5.3 Wie können Sie sicherstellen, dass Sie eine Methode in einer abgeleiteten Klasse auch tatsächlich überschreiben und diese nicht nur verdeckt wird?

6 Wiederverwendung von Quelltexten

In diesem Kapitel lernen Sie, wie Sie einmal erstellte Klassen in beliebigen anderen Projekten wiederverwenden.

Als Beispiel benutzen wir dabei eine sehr einfache Klasse mit zwei Klassenmethoden. Diese Klasse speichern wir in einem eigenen Projekt mit dem Namen **MeineKlassen**. Da unsere Klasse keine ausführbare Anwendung enthalten soll, benutzen wir als Vorlage aber nicht mehr eine Konsolenanwendung, sondern eine Klassenbibliothek für den .NET-Standard.

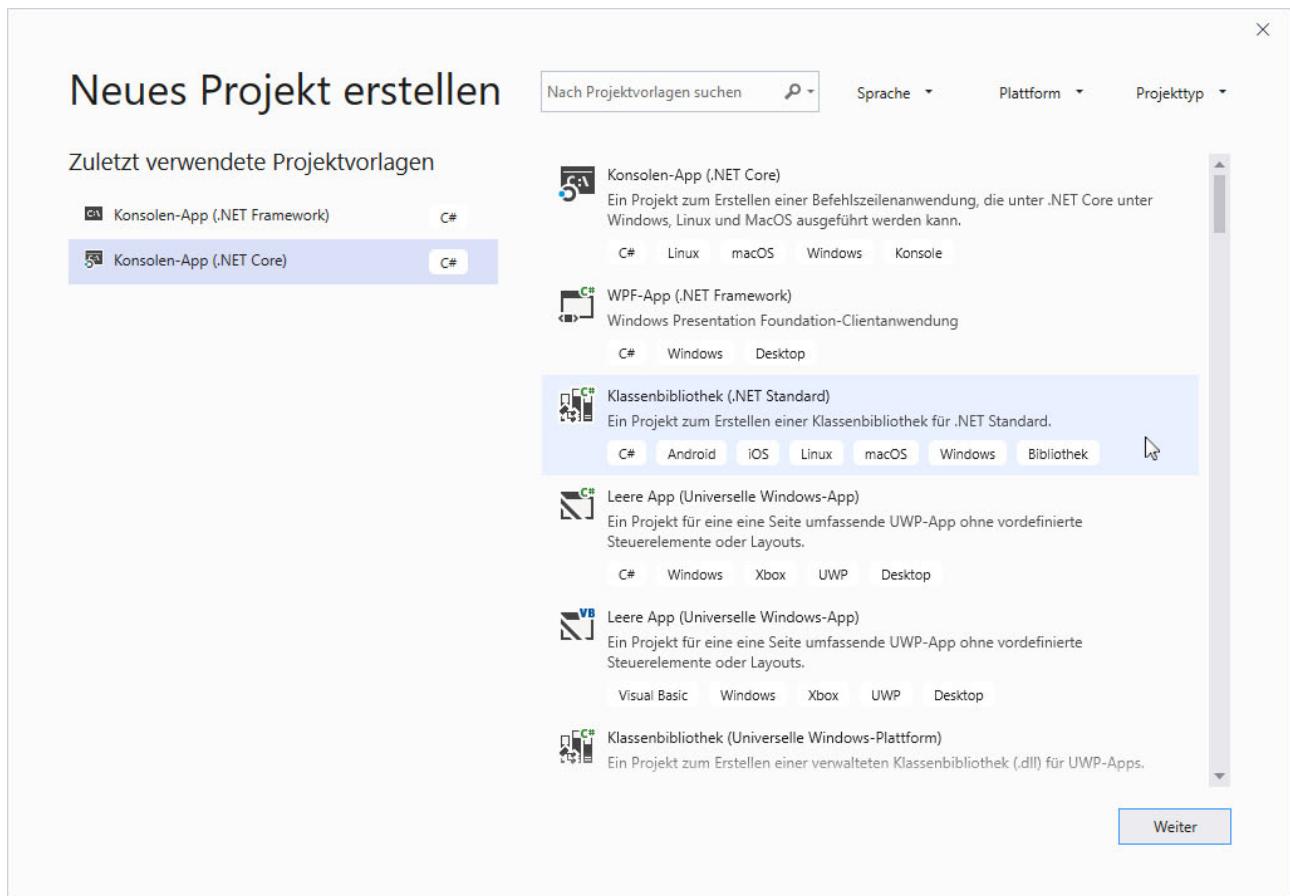


Abb. 6.1: Die Vorlage **Klassenbibliothek (.NET Standard)** im Fenster **Neues Projekt erstellen** (in der Mitte der Abbildung am Mauszeiger)

Legen Sie jetzt ein neues Projekt mit der Vorlage **Klassenbibliothek** für den .NET-Standard an. Als Namen verwenden Sie bitte **MeineKlassen**.

Bitte beachten Sie:

Es gibt mehrere Vorlagen für Klassenbibliotheken. Wir benötigen eine Klassenbibliothek für den .NET-Standard für die Programmiersprache C#.



Das Quelltextgerüst, das Visual Studio anlegt, sieht fast genauso aus wie bei einer Konsoleanwendung. Es fehlt lediglich die Methode `Main()`, außerdem ist die Klasse als `public` vereinbart. So wird sichergestellt, dass der Zugriff auf die Klasse von jeder beliebigen Stelle aus möglich ist.

Damit Sie die Klasse später schneller zuordnen können, sollten Sie ihr einen eindeutigen Namen geben – zum Beispiel `EinUndAusgabe` – und auch die Quelltextdatei unter diesem Namen ablegen. Das geht am schnellsten, wenn Sie den Eintrag der Quelltextdatei mit der Erweiterung `.cs` im Projektmappen-Explorer markieren und die Datei anschließend mit der Funktion **Umbenennen** im Kontextmenü umbenennen. Ändern Sie danach auch den Namen der Klasse im Code.

Übernehmen Sie anschließend die beiden Klassenmethoden aus dem folgenden Code 6.1 und speichern Sie das komplette Projekt.

```
/*#####
Eine wiederverwendbare Klasse
#####*/
using System;
namespace MeineKlassen
{
    public class EinUndAusgabe
    {
        static public string Eingabe()
        {
            string zeichenkette;
            Console.Write("Bitte geben Sie einen Wert ein: ");
            zeichenkette = Console.ReadLine();
            return zeichenkette;
        }

        static public void Ausgabe(string wert)
        {
            Console.WriteLine("Ihre Eingabe war: {0}", wert);
        }
    }
}
```

Code 6.1: Die wiederverwendbare Klasse

Hinweis:

Sie finden die Klasse im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **MeineKlassen** unter dem Namen **EinUndAusgabe.cs**.



Bitte beachten Sie:

Unsere Klasse enthält keine Methode `Main()`, sondern nur die beiden Klassenmethoden `Eingabe()` und `Ausgabe()`. Der Code enthält also kein ausführbares Konsolenprogramm.

Um die Klasse in einem anderen Projekt einzusetzen, rufen Sie die Funktion **Projekt/Vorhandenes Element hinzufügen ...** auf und wechseln in den Ordner, in dem sich die .cs-Datei mit der Klasse befindet. Markieren Sie diese Datei und klicken Sie dann auf **Hinzufügen**.

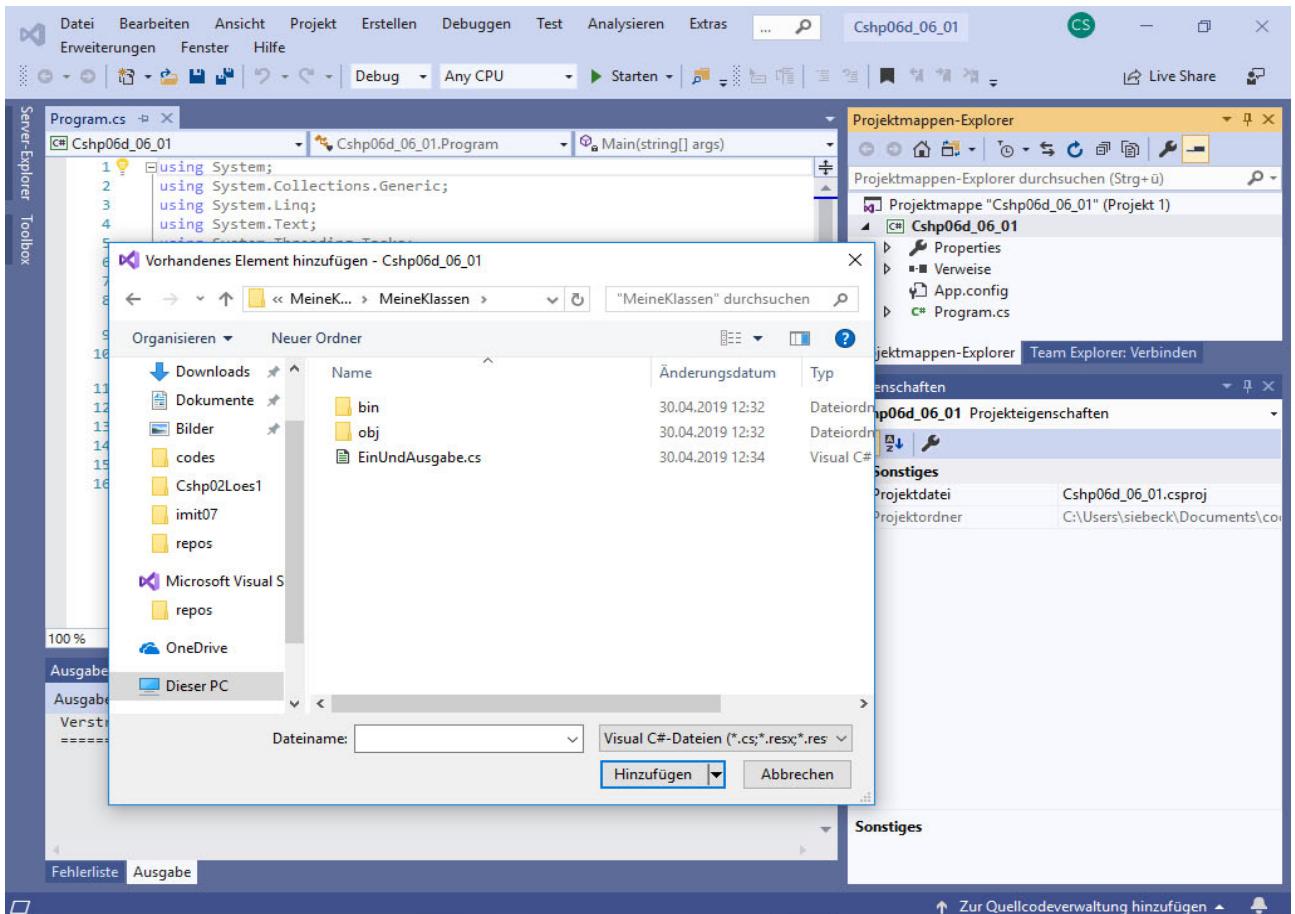


Abb. 6.2: Der Dialog **Vorhandenes Element hinzufügen** mit der Datei **EinUndAusgabe.cs**

Nach dem Einfügen der Datei können Sie dann auf die Klasse zugreifen. Dabei müssen Sie allerdings den kompletten Namen mitsamt eventuell vereinbarten Namensräumen angeben. Für unser Beispiel müsste der Aufruf der Methode `Eingabe()` aus der Klasse `EinUndAusgabe` im Namensraum `MeineKlassen` also so aussehen:

```
string eingabe = MeineKlassen.EinUndAusgabe.Eingabe();
```

Vor dem Namen der Klassenmethode `Eingabe()` stehen der Bezeichner des Namensraums und der Bezeichner der Klasse – jeweils durch Punkte getrennt.

Hinweis:

Ein Beispiel, das auf die Methoden `Eingabe()` und `Ausgabe()` zugreift, finden Sie im heftbezogenen Download-Bereich der Online-Lernplattform im Projekt **Cshp06d_06_01**.

Beim Einfügen von Dateien über die Funktion **Projekt/Vorhandenes Element hinzufügen ...** erstellt Visual Studio in der Standardeinstellung Kopien der Dateien und legt diese Kopien im Projektordner ab. Änderungen, die Sie an einer eingefügten Datei vor-

nehmen, wirken sich daher nicht auf das Original aus. Wenn Sie die Änderungen automatisch in allen beteiligten Projekten durchführen wollen, klicken Sie im Dialog **Vorhandenes Element hinzufügen** auf den Pfeil rechts in der Schaltfläche **Hinzufügen** und wählen Sie die Funktion **Als Link hinzufügen**. Dann setzt Visual Studio lediglich einen Verweis auf die Datei in das Projekt.

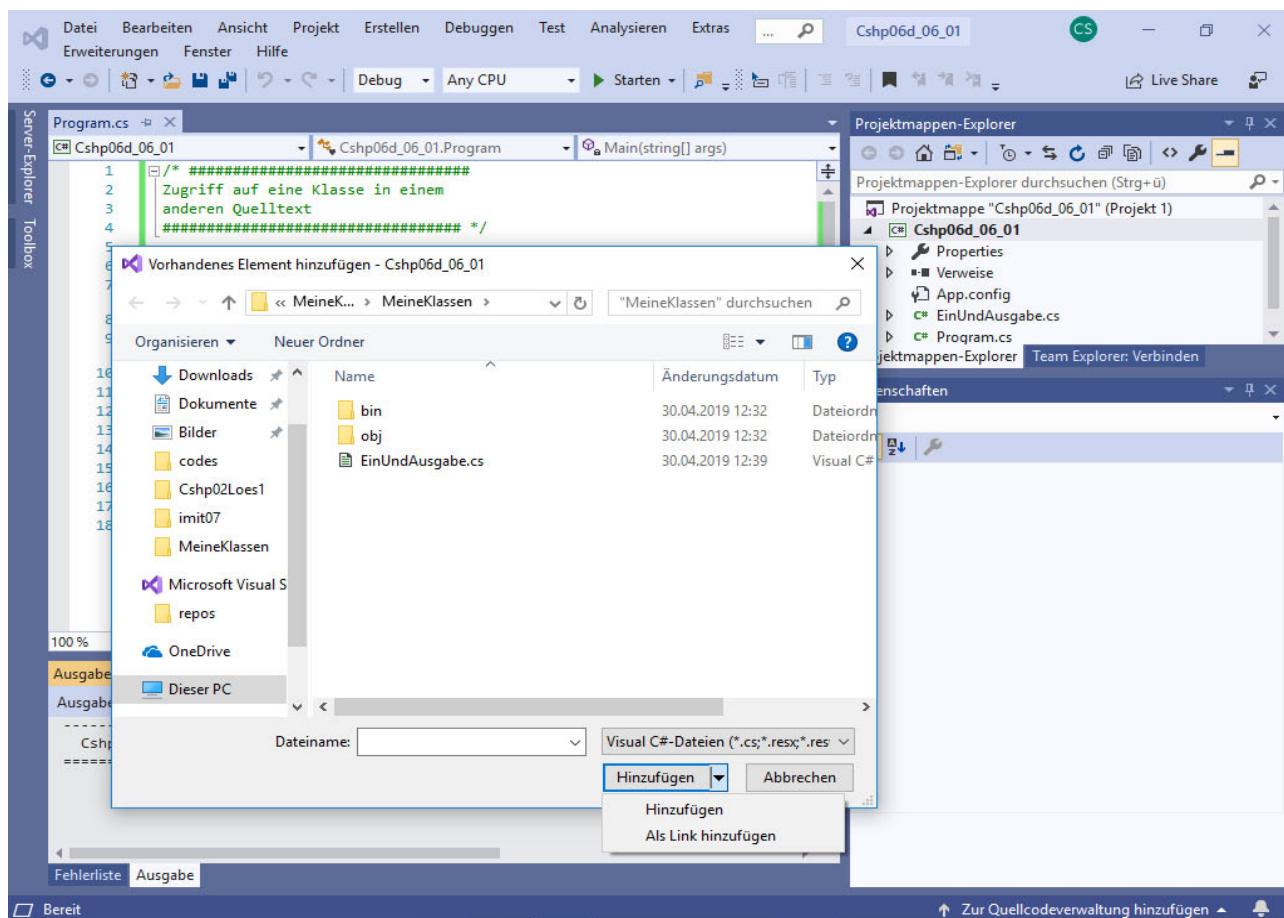


Abb. 6.3: Die Funktion **Als Link hinzufügen** der Schaltfläche **Hinzufügen**

Das ist allerdings bei den ersten Gehversuchen auch nicht ganz ungefährlich. Wenn Sie sich nämlich eine Datei „zerschießen“, funktionieren unter Umständen gleich mehrere Projekte nicht mehr. Arbeiten Sie daher zu Beginn lieber mit Kopien und lassen Sie das Original unverändert. Im Fall der Fälle können Sie so immer noch auf eine funktionierende Version zurückgreifen.

Zusammenfassung

Einmal erstellte Klassen können Sie auch in anderen Projekten wiederverwenden. Dazu fügen Sie den Quelltext mit der Klasse zum Projekt hinzu.

Damit der Zugriff auf die Klasse von allen Stellen möglich ist, verwenden Sie die Sichtbarkeit `public` für die Klasse.

Aufgaben zur Selbstüberprüfung

- 6.1 Welche Vorlage von Visual Studio können Sie sehr gut für eine Klasse benutzen, die Sie wiederverwenden wollen?

- 6.2 Sie fügen eine Datei mit der Funktion **Projekt/Vorhandenes Element hinzufügen ...** in ein Projekt ein. Befindet sich in dem Projekt danach eine Kopie dieser Datei oder ein Verweis zum Original?

- 6.3 Sie haben eine Klasse `Test` im Namensraum `Probe` erstellt und in einer eigenen Datei gespeichert. Wie können Sie aus einer anderen Klasse auf eine Klassenmethode `RufMichAuf()` in dieser Klasse `Test` zugreifen? Geben Sie bitte den kompletten Namen an.

Schlussbetrachtung

Herzlichen Glückwunsch! Sie dürfen jetzt wirklich stolz auf sich sein und auch erst einmal ein wenig verschnaufen.

In diesem Studienheft haben Sie sich mit komplexen und teilweise sehr abstrakten Themen wie der Datenkapselung, der Vererbung und dem Polymorphismus auseinanderge setzt. Sie kennen jetzt wesentliche Prinzipien der objektorientierten Programmierung und können Sie mit C# umsetzen.

Wenn Sie bei dem einen oder anderen Thema noch etwas unsicher sind, lassen Sie sich nicht entmutigen. Themen wie Polymorphismus, Überladen und Überschreiben sind nicht ganz einfach nachzuvollziehen und erschließen sich häufig erst bei intensiver praktischer Anwendung.

Nehmen Sie sich daher Zeit und entwickeln Sie eigene Programme, die die Techniken aus diesem Studienheft umsetzen. Sollten Ihre ersten eigenen Gehversuche dabei nicht sofort von Erfolg gekrönt sein, arbeiten Sie bitte die entsprechenden Kapitel in diesem Studienheft noch einmal in Ruhe durch.

Mit diesem Studienheft werden wir uns auch von den Konsolenprogrammen verabschieden. Denn Sie haben sich jetzt alle notwendigen Grundlagen erarbeitet, um mit der Programmierung von Anwendungen mit grafischer Oberfläche beginnen zu können.

Falls Sie jetzt glauben, das Erstellen solcher Programme sei schwierig und kompliziert: Sie werden eine angenehme Überraschung erleben. Visual Studio nimmt Ihnen sehr viel Arbeit ab. Nur so viel schon vorweg: Sie werden in vergleichsweise kurzer Zeit mit relativ wenig Aufwand auch anspruchsvollere Programme erstellen können.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Das Schlüsselwort für die private Vereinbarung heißt `private`. Das Schlüsselwort für die öffentliche Vereinbarung heißt `public`.
- 1.2 Die Klassenvereinbarung könnte zum Beispiel so aussehen:

```
class Wein
{
    private int alter;
    private int menge;
    private int wert;

    //die private Methode
    private int WertBerechnen()
    {

    }
    //die öffentlichen Methoden
    public void Init(int, int)
    {

    }

    public void Trinken()
    {

    }

    public void Altern()
    {

    }
}
```

Die Methode `WertBerechnen()` kann als `private` vereinbart werden, da sie nur innerhalb der Klasse selbst eingesetzt wird.

Die ausdrückliche Angabe von `private` kann auch wegfallen.

Kapitel 2

- 2.1 Der Konstruktor wird beim Erzeugen einer Instanz ausgeführt, der Destruktor beim Zerstören.
- 2.2 Der Standardkonstruktor ist ein Konstruktor mit leerer Parameterliste. Er wird automatisch angelegt.
- 2.3 Beim Erstellen eines eigenen Konstruktors wird der Standardkonstruktor gelöscht.
- 2.4 Es ist nicht klar, wann die Garbage Collection durchgeführt wird. Damit ist auch nicht absehbar, wann der Destruktor genau ausgeführt wird.

Kapitel 3

- 3.1 Überladene Methoden müssen sich durch die Parameterliste eindeutig voneinander unterscheiden.
- 3.2 Sie überladen die Konstruktoren.
- 3.3 Nein, das Überladen in dieser Form ist nicht möglich. Unterschiedliche Rückgabetypen reichen für eine eindeutige Zuordnung beim Aufruf nicht aus.
- 3.4 Durch die Anweisung

```
MethodeB(1.2);
```

wird keine der beiden Methoden aufgerufen, da der Wert 1.2 als double interpretiert wird. Sie müssen den Wert 1.2 ausdrücklich als float kennzeichnen – zum Beispiel durch

```
MethodeB(1.2F);
```

oder durch

```
MethodeB((float)1.2);
```

Kapitel 4

- 4.1 Eine Generalisierung verallgemeinert eine untergeordnete Klasse. Eine Spezialisierung erweitert eine übergeordnete Klasse.
- 4.2 Konstruktoren werden bei der Vererbung nicht weitergegeben.
- 4.3 Die erste Zeile der Klassenvereinbarung könnte so aussehen:

```
class Hund : Haustiere
```

Kapitel 5

- 5.1 Beim Überladen von Methoden gibt es mehrere Methoden mit demselben Namen und unterschiedlichen Parameterlisten in einer und derselben Klasse. Beim Überschreiben von Methoden gibt es mehrere Methoden mit demselben Namen und derselben Parameterliste in unterschiedlichen Klassen.

- 5.2 Bei der statischen Bindung erfolgt die Zuordnung von Methoden zu Klassen vor der Ausführung des Programms. Bei der dynamischen Bindung erfolgt die Zuordnung erst bei der Ausführung des Programms. Dabei wird auch der Typ ausdrücklich überprüft.
- 5.3 Sie vereinbaren die Methode in der Basisklasse als `virtual`. Bei der Methode in der abgeleiteten Klasse geben Sie das Schlüsselwort `override` an.

Kapitel 6

- 6.1 Für eine Klasse, die Sie wiederverwenden wollen, können Sie die Vorlage **Klassenbibliothek** benutzen.
- 6.2 Die Funktion **Projekt/Vorhandenes Element hinzufügen ...** fügt in der Standardeinstellung eine Kopie der Datei ein.
- 6.3 Für den Zugriff müssen der Namensraum und der Name der Klasse vor den Namen der Klassenmethode gestellt werden. Der Zugriff erfolgt also mit `Probe.Test.RufMichAuf()`.

B. Glossar

Abgeleitete Klasse	Eine abgeleitete Klasse ist eine Klasse, deren Eigenschaften zum Teil von einer anderen Klasse stammen. In der abgeleiteten Klasse können zusätzliche Eigenschaften definiert werden.
Attribut	Attribute beschreiben den Zustand eines Objekts. Der Begriff wird auch im relationalen Datenmodell für die Spalten einer Tabelle benutzt. Bei C# werden die Attribute Felder genannt.
Attributwert	Der Attributwert ist der Wert, den ein Attribut aktuell hat.
Destruktor	Ein Destruktor ist eine besondere Methode, die automatisch beim Zerstören einer Instanz aufgerufen wird.
Dynamische Bindung	Die dynamische Bindung ordnet Klassen und Methoden bei der Ausführung des Programms zu.
Encapsulation	<i>Encapsulation</i> ist der englische Begriff für Datenkapselung.
Feld	Als Feld wird in der Programmiersprache C# ein Attribut einer Klasse bezeichnet.
Frühe Bindung	Siehe statische Bindung.
Garbage Collection	Die <i>Garbage Collection</i> gibt Speicher frei und reorganisiert den Speicher. Sie wird automatisch ausgeführt.
Garbage Collector	Der <i>Garbage Collector</i> ist der Prozess, der die <i>Garbage Collection</i> ausführt.
Geheimnisprinzip	Das Geheimnisprinzip besagt, dass vor allem das Was einer Klasse nach außen sichtbar sein muss, nicht das Wie.
Generalisierung	Die Generalisierung ist eine Vererbungsbeziehung. Sie drückt aus, dass eine übergeordnete Klasse eine untergeordnete Klasse verallgemeinert.
Geschützte Vereinbarung	Mit der geschützten Vereinbarung steht eine Methode oder ein Feld einer Klasse nur der Klasse selbst oder einer abgeleiteten Klasse zur Verfügung. Die geschützte Vereinbarung erfolgt über das Schlüsselwort <code>protected</code> .
Information hiding	<i>Information hiding</i> ist der englische Begriff für Geheimnisprinzip.
Klasse	In einer Klasse werden Objekte mit ähnlichen Eigenschaften und ähnlichem Verhalten zusammengefasst. Klassen stehen in einem hierarchischen Verhältnis zueinander. Eine Klasse bildet den Bauplan für ein Objekt.

Klassendiagramm	Im Klassendiagramm werden Klassen und ihre Beziehungen untereinander beschrieben. Die Klassen werden dabei durch ein Rechteck dargestellt. Die Beziehungen werden durch verschiedene Linien dargestellt.
Klassenhierarchie	Die Klassenhierarchie beschreibt die Abhängigkeiten der Klassen. Aus den übergeordneten Klassen werden die Eigenschaften der untergeordneten Klassen abgeleitet.
Klassenmethode	Eine Klassenmethode ist eine Methode, die aufgerufen werden kann, ohne eine Instanz der Klasse erzeugen zu müssen. Die Vereinbarung einer Klassenmethode muss mit dem Schlüsselwort <code>static</code> erfolgen.
Klassenvariable	Eine Klassenvariable ist eine Variable, die für die gesamte Klasse gilt.
Konstruktor	Ein Konstruktor ist eine besondere Methode, die automatisch beim Erzeugen einer Instanz aufgerufen wird.
Methoden	Methoden beschreiben das Verhalten eines Objekts.
Objekt	Ein Objekt in der objektorientierten Programmierung ist jede in sich geschlossene Einheit. Ein Objekt besteht aus dem Zustand und dem Verhalten.
Objektbezeichner	Der Objektbezeichner ist ein eindeutiger Name eines Objekts. Er wird einmal vergeben und kann nicht verändert werden.
Objektdiagramm	Im Objektdiagramm werden Objekte und ihre Beziehungen untereinander beschrieben. Die Objekte werden dabei durch ein Rechteck dargestellt.
Objektgleichheit	Objektgleichheit bezeichnet zwei oder mehr identische Objekte.
Objektidentität	Über die Objektidentität wird ein Objekt eindeutig identifiziert.
Objektorientierte Programmierung	Bei der objektorientierten Programmierung werden Daten- und Verhaltensaspekte gemeinsam betrachtet. Das wesentliche Element der objektorientierten Programmierung ist das Objekt.
Öffentliche Vereinbarung	Mit der öffentlichen Vereinbarung steht eine Methode oder ein Attribut einer Klasse allen anderen Klassen zur Verfügung. Die öffentliche Vereinbarung sollte nur dann eingesetzt werden, wenn es unbedingt erforderlich ist, da das Prinzip der Datenkapselung verletzt wird. Die öffentliche Vereinbarung erfolgt über das Schlüsselwort <code>public</code> .

Polymorphismus	Polymorphismus bezeichnet die Möglichkeit, gleiche Begriffe in unterschiedlichen Situationen auf verschiedene Arten einzusetzen.
Private Vereinbarung	Mit der privaten Vereinbarung steht eine Methode oder ein Feld einer Klasse nur der Klasse selbst zur Verfügung. Die private Vereinbarung erfolgt über das Schlüsselwort <code>private</code> .
Referenz	Eine Referenz ist ein Verweis auf ein Objekt.
Sichtbarkeit	Die Sichtbarkeit legt fest, wie streng die Datenkapselung in einer Klasse erfolgt, beziehungsweise regelt, wann auf ein Datenobjekt zugegriffen werden kann.
Späte Bindung	Siehe dynamische Bindung
Spezialisierung	Die Spezialisierung ist eine Vererbungsbeziehung. Sie drückt aus, dass eine untergeordnete Klasse eine übergeordnete Klasse erweitert.
Standardkonstruktor	Ein Standardkonstruktor ist ein Konstruktor ohne Parameter.
Statische Bindung	Bei der statischen Bindung erfolgt die Zuordnung der Methoden zu den Klassen beim Übersetzen – also vor dem eigentlichen Ausführen des Programms.
Subclass	Siehe unmittelbare Unterklasse
Substitution	Bei der Substitution kann eine abgeleitete Klasse überall dort benutzt werden, wo auch die übergeordnete Klasse verwendet werden kann.
Superclass	Siehe unmittelbare Oberklasse
this-Referenz	Die <code>this</code> -Referenz bezieht sich auf die Instanz einer Klasse, die eine Methode aufgerufen hat.
Überladen	Beim Überladen können Sie mehrere Methoden mit demselben Namen in derselben Klasse erstellen. Die Unterscheidung erfolgt durch die Parameterlisten.
Überschreiben	Beim Überschreiben erstellen Sie mehrere Methoden mit demselben Namen und derselben Parameterliste in unterschiedlichen Klassen.
UML	UML steht für <i>Unified Modeling Language</i> . UML ist eine objektorientierte Standardmodellierungssprache zur Beschreibung der Struktur und des Verhaltens von Objekten in Anwendungsbereichen und Datenverarbeitungssystemen.

Unified Modeling Language	Siehe UML
Unmittelbare Oberklasse	Eine unmittelbare Oberklasse liegt in der Klassenhierarchie direkt über einer Klasse.
Unmittelbare UnterkLASSE	Eine unmittelbare UnterkLASSE liegt in der Klassenhierarchie direkt unter einer Klasse.
Vererbung	Über die Vererbung können Sie Attribute und Methoden aus übergeordneten Klassen an eine untergeordnete Klasse weitergeben. Die ererbten Methoden und Attribute können verändert und erweitert werden.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl. Bonn: Rheinwerk.

Theis, T. (2015). *Einstieg in Visual C# mit Visual Studio 2015. Ideal für Programmieranfänger geeignet.* 6. Auflage. Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Die Klassenansicht für die Klasse Esel	6
Abb. 4.1	Klassenhierarchie	35
Abb. 4.2	Beispielhafte Klassenhierarchie für Säugetiere	36
Abb. 4.3	Beispielhafte Vererbung von „Säugetiere“ auf „Kängurus“	37
Abb. 4.4	Die Klassenhierarchie der Säugetiere als UML-Klassendiagramm	38
Abb. 5.1	Polymorphe Methoden	52
Abb. 6.1	Die Vorlage Klassenbibliothek (.NET Standard) im Fenster Neues Projekt erstellen (in der Mitte der Abbildung am Mauszeiger) ..	61
Abb. 6.2	Der Dialog Vorhandenes Element hinzufügen mit der Datei EinUndAusgabe.cs	63
Abb. 6.3	Die Funktion Als Link hinzufügen der Schaltfläche Hinzufügen	64

E. Tabellenverzeichnis

Tab. 4.1 Die Sichtbarkeiten für Felder und Methoden..... 49

F. Codeverzeichnis

Code 1.1	Zugriff auf ein Feld von außen (das Programm lässt sich so nicht übersetzen)	4
Code 1.2	Beispiel für eine öffentliche Feldvereinbarung	8
Code 1.3	Gemischte Verwendung von öffentlicher und privater Vereinbarung ..	10
Code 1.4	Eine Eigenschaft im praktischen Einsatz	14
Code 2.1	Initialisierung mit der Methode Init()	18
Code 2.2	Initialisierung mit einem Konstruktor	20
Code 2.3	Ein Destruktor	23
Code 3.1	Beispiele für überladene Methoden	29
Code 3.2	Ein Beispiel für überladene Konstruktoren	31
Code 4.1	Eine Vererbung	42
Code 4.2	Vererbung ohne Standardkonstruktoren (der Code lässt sich nicht übersetzen)	45
Code 4.3	Der Aufruf des Konstruktors der übergeordneten Klasse	45
Code 4.4	Die neue Methode Schock() für die Klasse ElternBaer	47
Code 4.5	Ein geschütztes Feld (es handelt sich um ein Fragment)	48
Code 5.1	Eine überschriebene Methode	54
Code 5.2	Probleme mit der statischen Bindung (es handelt sich um ein Fragment)	56
Code 6.1	Die wiederverwendbare Klasse	62

G. Sachwortverzeichnis

B

- Bindung
 dynamische 57
 frühe 55
 späte 57
 statische 55

D

- Datenkapselung 3
 Destruktor 22

E

- Eigenschaft 11
 encapsulation 3

G

- Geheimnisprinzip 3
 Generalisierung 34

I

- information hiding 3

K

- Konstruktor 18

M

- Mehrfachvererbung 38

O

- Oberklasse
 unmittelbare 34

P

- Polymorphismus 51

S

- Sichtbarkeit 3
 für Felder und Methoden 49
 Spezialisierung 34
 Standardkonstruktor 21
 subclass 34
 Substitution 57
 superclass 34

U

- Überladen 26
 Überschreiben 55
 Unified Modeling Language 37
 Unterklasse
 unmittelbare 34

V

- Vererbung 34

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP06D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

1. Sie haben folgende Klassenvereinbarung:

```
class Hund
{
    int gewicht;
    int alter;
}
```

Kann eine Klasse `KleinerHund`, die von der Klasse `Hund` abgeleitet wird, auf die Felder `gewicht` und `alter` der Klasse `Hund` zugreifen? Begründen Sie bitte Ihre Antwort.

5 Pkt.

2. Sie haben folgende Vereinbarung einer Klasse `Katze`:

```
class Katze
{
    int groesse;

    public Katze(int groesse)
    {
        this.groesse = groesse;
    }
    ...
}
```

Was geschieht, wenn Sie eine neue Instanz der Klasse `Katze` mit der Anweisung

```
Katze kleineKatze = new Katze();
```

erzeugen wollen? Begründen Sie bitte Ihre Antwort.

5 Pkt.

3. Sie haben folgende Klassenvereinbarungen:

```
class Hund
{
    int gewicht;
    int alter;
}
```

```
class KleinerHund : Hund
{
    int groesse;
}
```

Die Klasse `KleinerHund` soll auf das Feld `alter` der Klasse `Hund` zugreifen können. Welche Möglichkeiten kennen Sie, diesen Zugriff zu ermöglichen? Bewerten Sie diese Möglichkeiten bitte auch. Achten Sie dabei besonders auf die Datenkapselung.

Ein Tipp: Sie sollten mindestens drei verschiedene Varianten nennen können. Zwei hängen direkt mit der Vereinbarung des Feldes zusammen, die dritte dagegen nicht.

15 Pkt.

- Betrachten Sie bitte folgendes Quelltextfragment. Es erzeugt ein Array für Instanzen der Klasse `Baer` und ruft für jedes Element im Array die Methode `Ausgeben()` auf.

```
Baer[] Baerenliste = new Baer[2];
Bierenliste[0] = new Baer(200, 3);
Bierenliste[1] = new ElternBaer(500, 5, 20);
foreach (Baer testBaer in Baerenliste)
    testBaer.Ausgeben();
```

Die Klasse `ElternBaer` ist von der Klasse `Baer` abgeleitet und verfügt über eine eigene Methode `Ausgeben()`, die die Methode `Ausgeben()` der Basisklasse überschreiben soll. Die Methode `Ausgeben()` der abgeleiteten Klasse ist **nicht** mit `override` markiert.

Wird die Methode der Basisklasse tatsächlich überschrieben? Wie lauten die vollständigen Namen der Methoden, die in der Schleife aufgerufen werden? Geben Sie die Namen bitte im Format `<Klassenname.Methodename>` an.

Ein Tipp zur Lösung: Wenn Sie mit dem Fragment auf dem Trockenen nicht zurecht kommen, bauen Sie es einfach in eines der „Bären“-Programme aus dem Studienheft ein.

10 Pkt.

- Sie wollen eine Klasse `Vogel` von der Klasse `Haustiere` ableiten. Die Klasse `Haustiere` verfügt lediglich über einen Konstruktor, der zwei `int`-Typen als Parameter verarbeitet. Was müssen Sie beim Konstruktor der Klasse `Vogel` in jedem Fall berücksichtigen?

10 Pkt.

- Sie haben folgende Basisklasse `Insekt`:

```
class Insekt
{
    protected int laenge;

    protected int gewicht;

    //der Konstruktor
    public Insekt(int laenge, int gewicht)
    {
        this.laenge = laenge;
        this.gewicht = gewicht;
```

```
}
```

```
//die Methode zum Essen
public virtual void Essen()
{
    laenge = laenge + 1;

    gewicht = gewicht + 1;
}

//die Methode zur Ausgabe
public virtual void Ausgabe()
{
    Console.WriteLine("Das Insekt ist {0} cm lang
        und wiegt {1} Gramm.", laenge, gewicht);
}
```

Leiten Sie aus dieser Basisklasse eine Klasse `Libelle` ab. Die Klasse `Libelle` soll zusätzlich ein Feld `fluegellaenge` enthalten, das über einen Konstruktor mit einem beliebigen Wert initialisiert werden kann.

Die Methode `Essen()` der Klasse `Libelle` soll nicht nur das Gewicht und die Länge erhöhen, sondern auch die Flügellänge.

Ändern Sie auch die Methode `Ausgabe()` der Klasse `Libelle` so, dass zusätzlich die Flügellänge ausgegeben wird.

Stellen Sie in Ihrem Programm sicher, dass die Methoden in der Klasse `Libelle` die Methoden in der Klasse `Insekt` in jedem Fall überschreiben.

Testen Sie Ihre neue Klasse, indem Sie eine `Main()`-Methode schreiben und in dieser je ein Objekt der Basisklasse und der abgeleiteten Klasse erzeugen und jeweils beide Methoden mindestens einmal aufrufen.

55 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Erste Schritte mit Windows Forms

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP07D

Objektorientierte Software-Entwicklung mit C#

Erste Schritte mit Windows Forms

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Erste Schritte mit Windows Forms

Inhaltsverzeichnis

Einleitung	1
1 Windows-Programme und Visual Studio	3
1.1 Das Erstellen von Windows-Programmen	3
1.2 Aufbau eines typischen Windows-Programms	4
1.3 Projekt für eine Windows Forms-Anwendung erstellen	5
Zusammenfassung	9
2 Das Programm „Hallo Welt“	10
2.1 Eigenschaften einstellen	10
2.2 Steuerelemente einfügen und positionieren	17
2.3 Verarbeitung von Ereignissen	20
Zusammenfassung	25
3 Einige kleine Spielereien	27
3.1 Texte zwischen Labels kopieren	28
3.2 Texte aus Kombinations- und Listenfeldern kopieren	32
3.3 Steuerelemente ein- und ausblenden	34
3.4 Der letzte Schliff	36
Zusammenfassung	37
4 Ein einfacher Taschenrechner	40
4.1 Vorüberlegungen und Vorbereitungen	40
4.2 Die Eingabefelder	41
4.3 Die Optionsfelder	42
4.4 Die Berechnungen	43
4.5 Der Feinschliff	46
Zusammenfassung	52
Schlussbetrachtung	54

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	55
B.	Glossar	57
C.	Literaturverzeichnis	59
D.	Abbildungsverzeichnis	60
E.	Codeverzeichnis	61
F.	Medienverzeichnis	62
G.	Sachwortverzeichnis	63
H.	Einsendeaufgabe	65

Einleitung

Alle Programme, die Sie bisher erstellt haben, liegen in der Eingabeaufforderung von Windows – also nicht unter der grafischen Oberfläche von Windows, sondern als rein textbasierte Anwendung.

Grafische Oberflächen werden auch als GUI bezeichnet. GUI steht für *Graphical User Interface* (frei übersetzt: „grafische Benutzeroberfläche“).



In diesem Studienheft beginnen wir mit dem Erstellen echter Windows-Programme. Anders als bei den Konsolenprogrammen, die Sie Zeile für Zeile selbst programmieren mussten, wird das Erstellen von Windows-Programmen direkt von Visual Studio unterstützt. So können Sie zum Beispiel mit wenig Aufwand Fenster und viele andere Objekte erstellen und die Eigenschaften dieser Objekte mit einigen wenigen Mausklicks verändern.

Im Einzelnen lernen Sie in diesem Studienheft,

- welche grundsätzlichen Möglichkeiten es für das Erstellen von Windows-Anwendungen gibt,
- wie ein typisches Windows-Programm aufgebaut ist,
- wie Sie ein Projekt für eine Windows Forms-Anwendung erstellen,
- was sich hinter dem Eigenschaftenfenster verbirgt,
- wie Sie Eigenschaften für Steuerelemente setzen,
- wie Sie Steuerelemente in ein Formular einfügen und positionieren,
- was Ereignisse sind,
- wie Sie Ereignisse verarbeiten,
- wie Sie Texte zwischen Labels kopieren,
- wie Sie Listen- und Kombinationsfelder einsetzen,
- wie Sie Kontrollkästchen verwenden und auswerten,
- welche Möglichkeiten Ihnen der Designer von Visual Studio für die Gestaltung von Formularen anbietet,
- wie Sie Eingabefelder verwenden,
- wie Sie Gruppen mit Optionsfeldern erstellen und auswerten,
- wie Sie Größenänderungen an einem Formular verhindern und
- wie Sie die Aktivierreihenfolge der Steuerelemente verändern.

Nach dem Durcharbeiten dieses Studienhefts haben Sie Ihre ersten echten Windows-Anwendungen erstellt – unter anderem einen einfachen Taschenrechner.

Ein wichtiger Hinweis:

Nach dem Öffnen eines Projekts für eine Windows Forms-Anwendung erscheint unter Umständen nicht direkt das Formular. Öffnen Sie es dann über einen Doppelklick auf den Eintrag der Datei **Form1.cs** im Projektmappen-Explorer.

Christoph Siebeck

1 Windows-Programme und Visual Studio

In diesem Kapitel beschäftigen wir uns zunächst mit den verschiedenen Möglichkeiten zum Erstellen von Windows-Programmen und stellen Ihnen den Aufbau von Windows-Programmen vor. Anschließend zeigen wir Ihnen, wie Sie mit Visual Studio ein neues Projekt für ein Windows-Programm anlegen.

Beginnen wir mit den verschiedenen Möglichkeiten zum Erstellen von Windows-Programmen.

1.1 Das Erstellen von Windows-Programmen

Grundsätzlich gibt es für das Erstellen von Windows-Programmen mehrere Möglichkeiten:

- 1) Sie programmieren die gesamte Anwendung „per Hand“ und greifen dabei auf Funktionen und Klassen aus der **Windows API**¹ zurück. Diese Variante ist allerdings recht kompliziert, sehr aufwendig und auch fehlerträchtig.
- 2) Sie benutzen die **Microsoft Foundation Classes**² – kurz MFC genannt. Diese Klassen kapseln zahlreiche Funktionen der Windows API und ermöglichen so einen leichteren und gleichzeitig auch komfortableren Zugriff. Trotzdem müssen Sie auch beim Einsatz der Microsoft Foundation Classes noch sehr viel Arbeit selbst erledigen.
- 3) Sie benutzen **Windows Forms**³. Bei dieser Variante erstellen Sie die Anwendungen sehr einfach über Formulare und fertige Steuerelemente. Die eigentliche Programmierarbeit beschränkt sich dabei in der Regel auf vergleichsweise wenige Anweisungen.
- 4) Sie arbeiten mit der **Windows Presentation Foundation**⁴ – kurz WPF genannt. Hier können Sie ebenfalls zahlreiche vorgefertigte Elemente einsetzen.
- 5) Sie erstellen **Universal Windows Platform Apps** – die universellen Apps.

Wir konzentrieren uns in diesem Lehrgang auf das Erstellen von Windows-Anwendungen über Windows Forms und die Windows Presentation Foundation. Außerdem beschäftigen wir uns mit dem Erstellen von Universal Windows Platform Apps. Der Einsatz der MFC ist für den Einstieg recht komplex und wird außerdem von Visual Studio nicht direkt unterstützt.



Sehen wir uns nun den Aufbau eines typischen Windows-Programms an.

1. Allgemein ausgedrückt handelt es sich bei einer API (*Application Programming Interface*) um eine Schnittstelle, über die Sie aus eigenen Programmen die Funktionen eines anderen Programms aufrufen können. Übersetzt bedeutet *Application Programming Interface* etwa „Schnittstelle für die Anwendungsprogrammierung“.
2. Frei übersetzt bedeutet *Microsoft Foundation Classes* so viel wie „grundlegende Microsoft Klassen“ oder „Microsoft Basisklassen“.
3. Übersetzt bedeutet *Windows Forms* so viel wie „Windows Schablonen“ oder „Windows Formulare“.
4. Frei übersetzt bedeutet *Windows Presentation Foundation* so viel wie „Windows Präsentationsbasis“.

1.2 Aufbau eines typischen Windows-Programms

Normale Windows-Programme laufen in Fenstern, die bestimmte Eigenschaften haben. So können Sie ein Anwendungsfenster zum Beispiel verschieben, minimieren und maximieren und auch beliebig in der Größe verändern. Jedes Anwendungsfenster hat eine Titelleiste, die den Namen der Anwendung und Symbole für Standardfunktionen wie das Minimieren, Maximieren und Schließen der Anwendung enthalten sollte.

In einem Anwendungsfenster selbst finden Sie in der Regel bestimmte Standardelemente zur Bedienung der Anwendung. Dazu gehören zum Beispiel:

- Menüleisten zum Auswählen von Befehlen,
- Symbolleisten zum Auswählen von Standardfunktionen mit der Maus,
- Menübänder zum Auswählen von Befehlen und Standardfunktionen,
- Listen- und Kombinationsfelder zum Auswählen von Einträgen,
- Kontrollkästchen zum Setzen von Einstellungen,
- Schaltflächen zum Auslösen bestimmter Aktionen,
- Schieberegler zum Einstellen von Werten und
- noch vieles mehr.

Ein typisches Anwendungsfenster könnte zum Beispiel so aussehen:

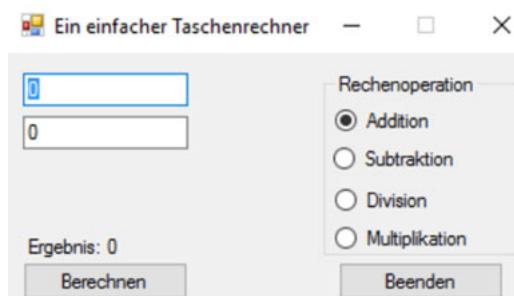


Abb. 1.1: Ein typisches Anwendungsfenster unter Windows

All diese Standardbedienelemente können Sie in Ihren eigenen Windows-Programmen natürlich ebenfalls einsetzen. Jetzt fragen Sie sich vielleicht etwas erschrocken: „Wie soll ich das denn alles programmieren?“

Keine Sorge: Richtig programmieren müssen Sie die Standardelemente kaum noch. Die Anwendung aus der vorigen Abbildung werden Sie zum Beispiel am Ende dieses Studienhefts ohne Schwierigkeiten selbst erstellen können.

Denn für die meisten Standardelemente und noch zahlreiche Funktionen mehr stellt Ihnen Visual Studio beziehungsweise das .NET Framework vorgefertigte Klassen und Steuerelemente zur Verfügung. Diese Klassen und Steuerelemente können Sie für eigene Objekte benutzen und anpassen. Um viele Standardfunktionen wie das Verschieben, Minimieren und Maximieren eines Anwendungsfensters müssen Sie sich dabei überhaupt nicht kümmern. Diese Funktionen werden von Visual Studio automatisch mit in Ihre Programme eingebaut, wenn Sie eine neue Anwendung mit grafischer Oberfläche erstellen. Auch die Eigenschaften eines Objekts – zum Beispiel den Titel eines Fensters – können Sie komfortabel mit einigen Mausklicks ändern.

1.3 Projekt für eine Windows Forms-Anwendung erstellen

Schauen wir uns nun an, wie Sie ein Projekt für eine Windows Forms-Anwendung erstellen.

Hinweis:

Wir konzentrieren uns hier vor allem auf die Besonderheiten beim Anlegen eines Projekts für eine Windows Forms-Anwendung.

Starten Sie bitte Visual Studio. Rufen Sie dann die Funktion zum Anlegen eines neuen Projekts auf.

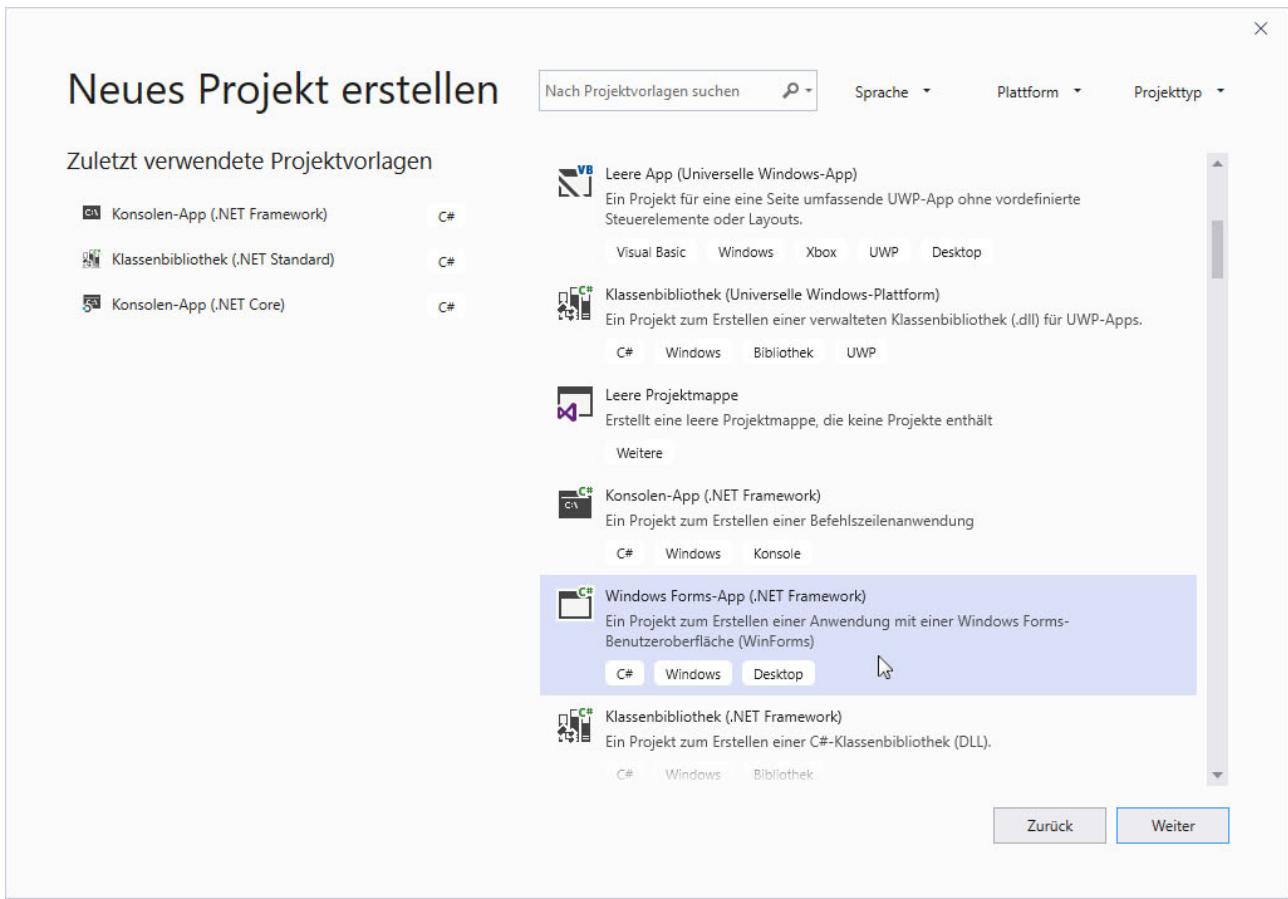


Abb. 1.2: Das Fenster Neues Projekt erstellen

Im Fenster **Neues Projekt erstellen** markieren Sie anschließend den Eintrag **Windows Forms-App (.NET Framework)** für C# in der Liste rechts und klicken auf die Schaltfläche **Weiter**.

Bitte beachten Sie:

Sie können auch Windows Forms-Apps mit unterschiedlichen Programmiersprachen erstellen. Bitte achten Sie daher darauf, dass Sie die Vorlage für C# auswählen.



Geben Sie dann wie gewohnt einen Namen für das Projekt ein. Unsere Anwendung soll den Namen **Cshp07d_p01** erhalten. Tragen Sie diesen Namen bitte in das Feld **Projektname** ein und klicken Sie auf die Schaltfläche **Erstellen**.

Visual Studio erzeugt die Dateien für das neue Projekt und zeigt Ihnen das Formular im Designer des Editors an.

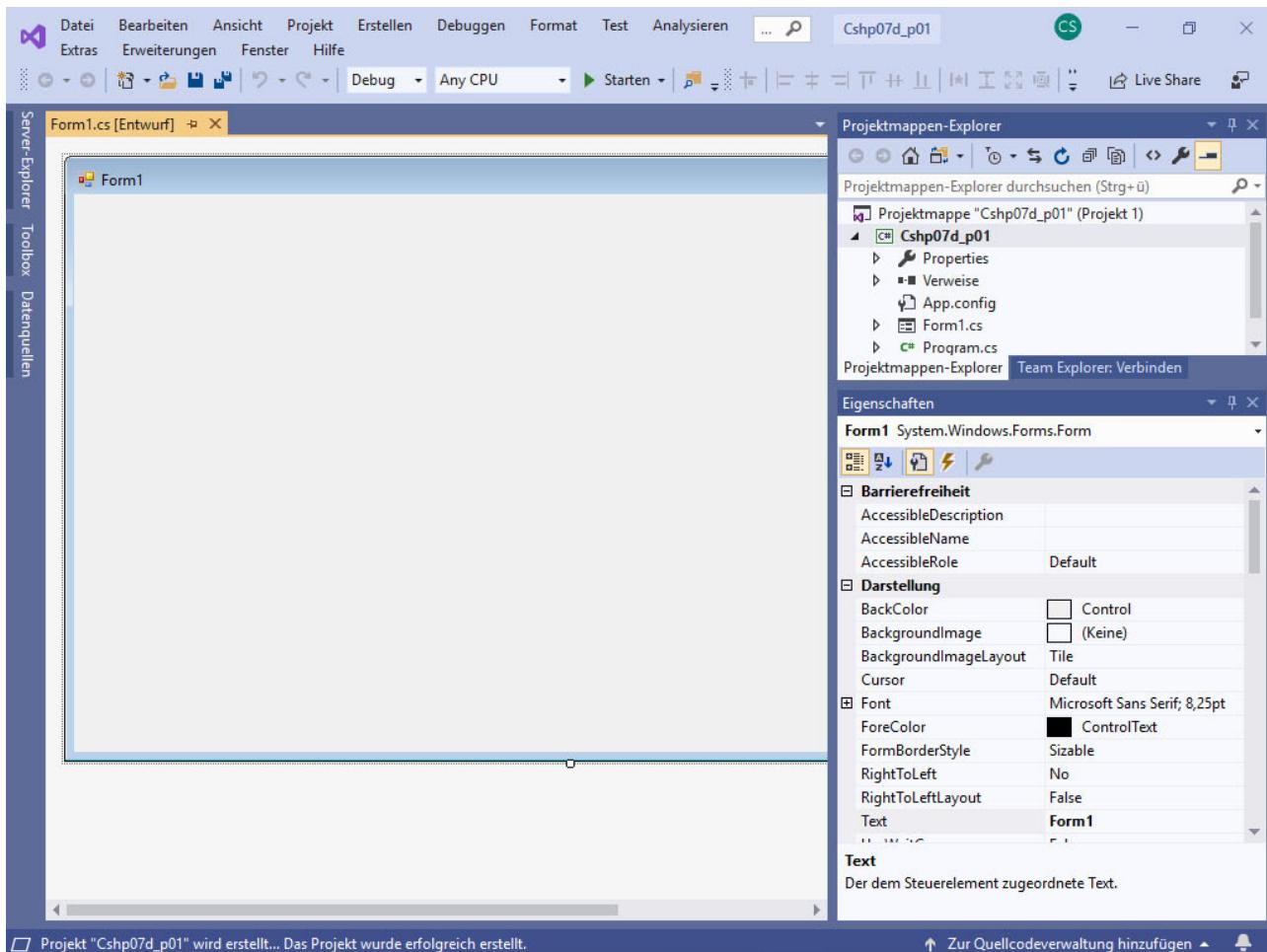


Abb. 1.3: Das fertige Projekt für eine Windows Forms-Anwendung

Dieses Formular mit dem Namen **Form1⁵** ist das Hauptfenster unserer späteren Anwendung. In das Fenster können Sie Steuerelemente wie zum Beispiel Listenfelder und Schaltflächen einfügen und mit der Maus anordnen. Das Einfügen dieser Elemente erfolgt im Wesentlichen über die **Toolbox** – den Werkzeugkasten. Sie finden ihn links im Fenster von Visual Studio – allerdings nur als Registerkarte. Die eigentliche Toolbox wird erst dann angezeigt, wenn Sie mit der Maus auf die Registerkarte klicken.

5. *Form* steht als Kurzform für Formular.

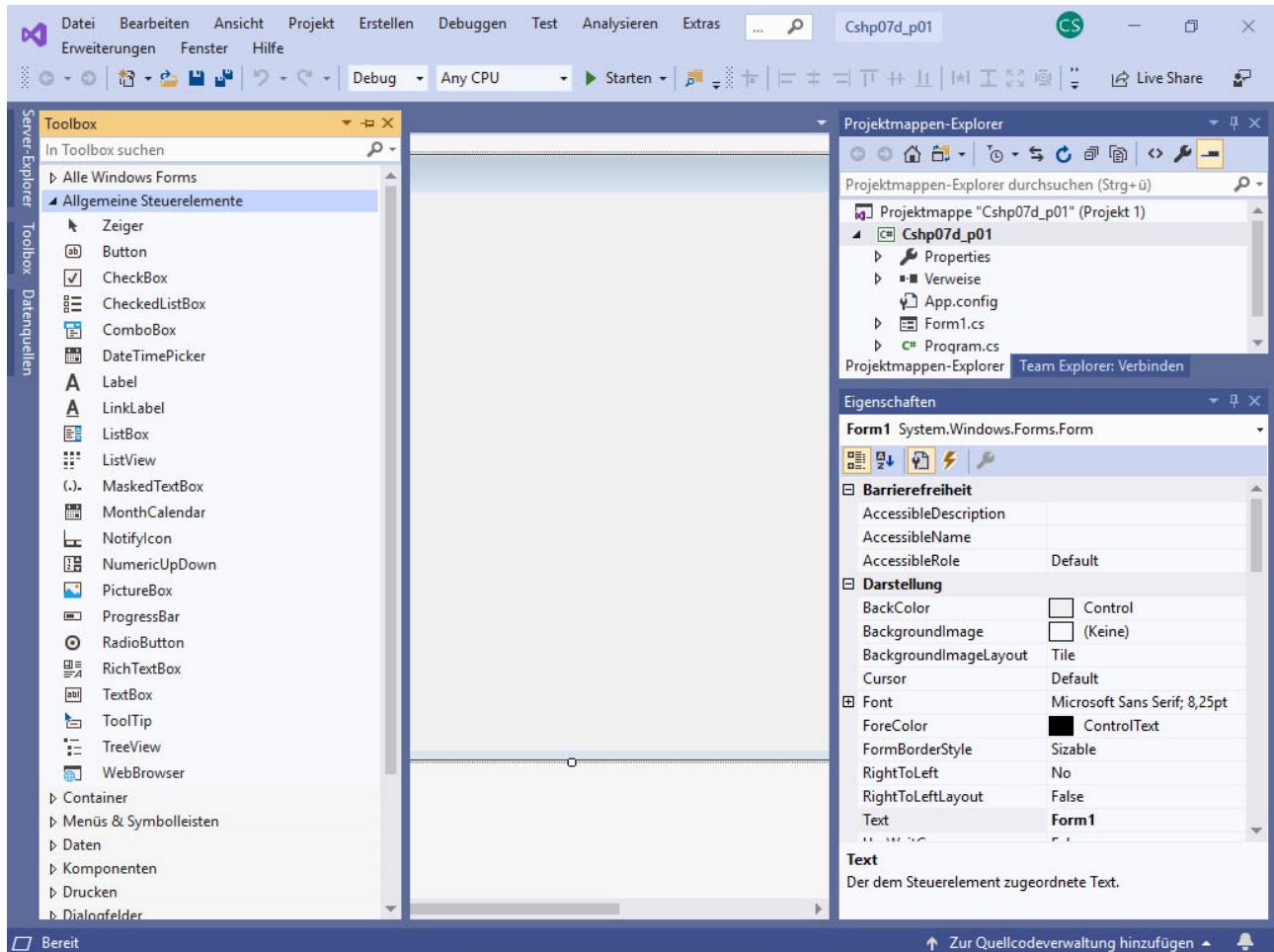


Abb. 1.4: Die Toolbox von Visual Studio
(links in der Abbildung)

Tipp:

Verschaffen Sie sich ruhig schon einmal einen kurzen Eindruck von der Vielfalt der Steuerelemente. Sehen Sie sich an, welche Einträge Sie in der Toolbox finden.

Falls bei Ihnen in der Toolbox nur übergeordnete Einträge wie **Alle Windows Forms** oder **Allgemeine Steuerelemente** angezeigt werden, klicken Sie einmal mit der Maus auf das Dreieck ▾ vor einem Eintrag beziehungsweise auf den Text des Eintrags. Dann werden auch die Steuerelemente der Gruppe angezeigt.

Die Vorlage, die Visual Studio erstellt hat, können Sie auch ausführen lassen. Probieren Sie das aus. Lassen Sie das Programm mit der Funktion **Debuggen/Starten ohne Debugging** oder mit der Tastenkombination **Strg + F5** übersetzen und ausführen.

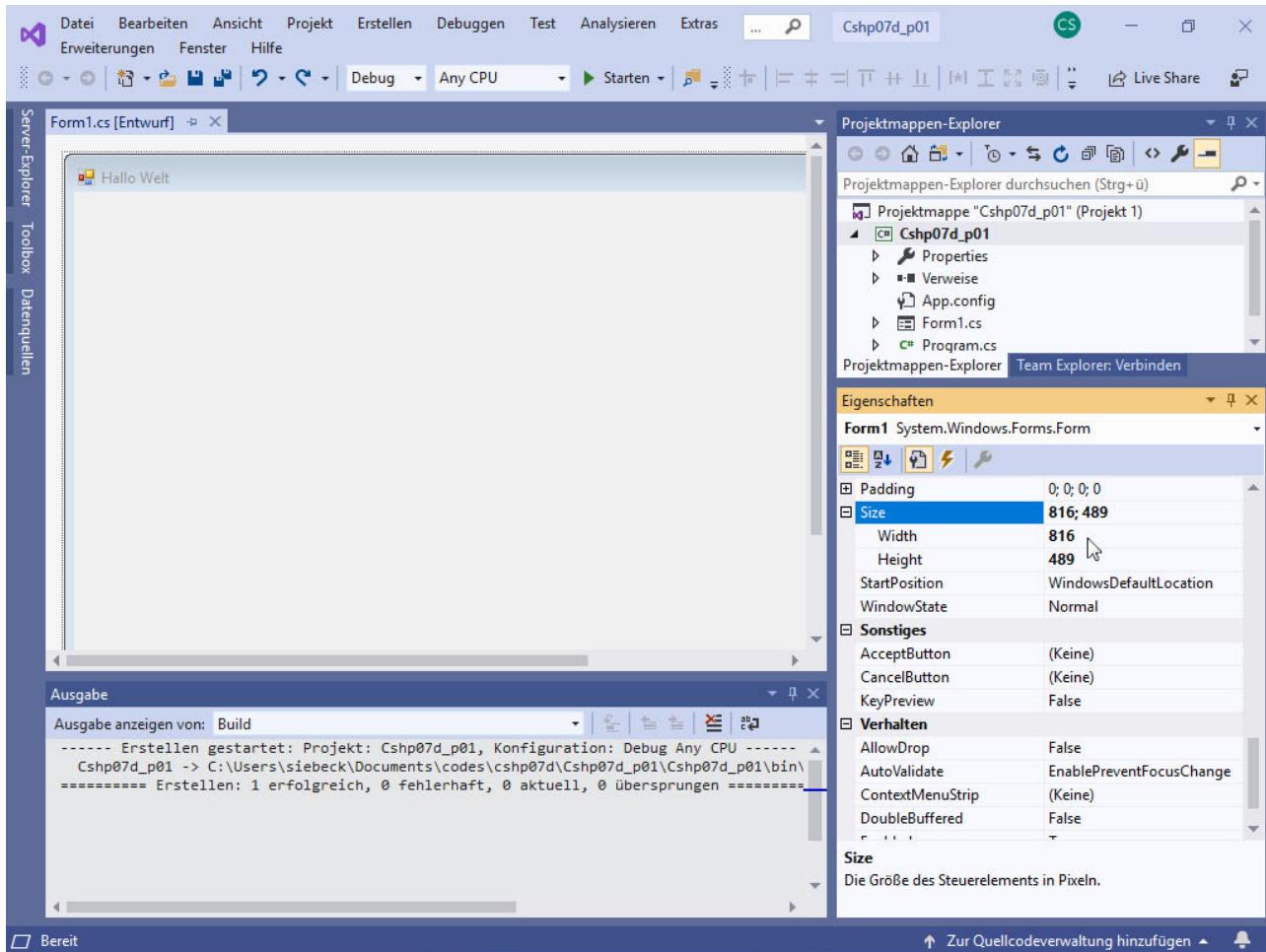


Abb. 1.5: Die erste Windows-Anwendung

Nach kurzer Zeit erscheint ein leeres Fenster, das Sie verschieben, minimieren und maximieren können. Auch das Systemmenü und die Standardschaltflächen rechts in der Titelleiste des Fensters funktionieren bereits – ohne dass Sie eine einzige Zeile programmiert haben. Das können Sie ganz einfach testen, indem Sie das Fenster über die entsprechende Schaltfläche ganz rechts in der Titelleiste oder über das Systemmenü wieder schließen.

Sehr sinnvoll ist solch eine Anwendung ohne Funktionalität natürlich nicht. In den nächsten Kapiteln werden wir daher das Anwendungsfenster auch mit Leben füllen. Speichern Sie dazu bitte das Projekt, das Sie gerade angelegt haben. Wir werden es gleich für die erste echte Windows-Anwendung benutzen.

Sie sehen bereits hier, wie komfortabel das Arbeiten mit Windows Forms ist. Sie müssen weder eine Klasse für das Fenster selbst vereinbaren noch sich mit der Vereinbarung von Feldern und Methoden herumschlagen. Alle Teile des Quelltextes, der für die Ausführung der Anwendung erforderlich ist, werden automatisch im Hintergrund von Visual Studio erzeugt.

Zusammenfassung

Normale Windows-Anwendungen laufen in Fenstern. Diese Fenster verfügen über Standardelemente wie die Titelleiste oder Schaltflächen zur Bedienung.

Über Windows Forms können Sie sehr einfach Windows-Anwendungen mit Standardsteuerelementen erstellen.

Für den Einsatz der Windows Forms legen Sie ein entsprechendes Projekt an. Dieses Projekt enthält bereits ein Gerüst für eine lauffähige Anwendung.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Welche grundsätzlichen Möglichkeiten zum Erstellen von Windows-Anwendungen kennen Sie? Nennen Sie bitte mindestens drei.

- 1.2 Beschreiben Sie kurz, wie Sie ein Projekt für eine Windows Forms-Anwendung anlegen.

- 1.3 Was verbirgt sich hinter dem Formular **Form1**, das für eine Windows Forms-Anwendung automatisch angelegt wird?

2 Das Programm „Hallo Welt“

In diesem Kapitel lernen Sie, wie Sie das Grundgerüst einer Windows Forms-Anwendung mit Leben füllen.

Als Beispiel überarbeiten wir dazu das Programm aus dem letzten Kapitel so, dass wir den Text „Hallo Welt“ in dem Fenster ausgeben und zusätzlich noch eine Schaltfläche zum Beenden der Anwendung anzeigen lassen.

Öffnen Sie jetzt bitte das Projekt **Cshp07d_p01**, falls Sie es nicht mehr geladen haben. Denken Sie dabei daran, die Funktion **Projekt öffnen** zu benutzen. Sie starten diese Funktion entweder über den Eintrag **Öffnen/Projekt/Projektmappe ...** im Menü **Datei** oder über den Eintrag des Projekts auf der Startseite.

2.1 Eigenschaften einstellen

Im ersten Schritt wollen wir zunächst einmal dafür sorgen, dass in der Titelleiste des Formulars nicht mehr der Text „Form1“ angezeigt wird, sondern der Text „Hallo Welt“. Dazu müssen wir die **Eigenschaften** des Formulars verändern. Das erfolgt über das **Eigenschaftenfenster**.

Falls dieses Fenster bei Ihnen nicht bereits unten rechts eingeblendet wird, lassen Sie es mit der entsprechenden Funktion im Menü **Ansicht** anzeigen. Alternativ können Sie auch im Kontextmenü des Formulars im Designer die Funktion **Eigenschaften** benutzen.

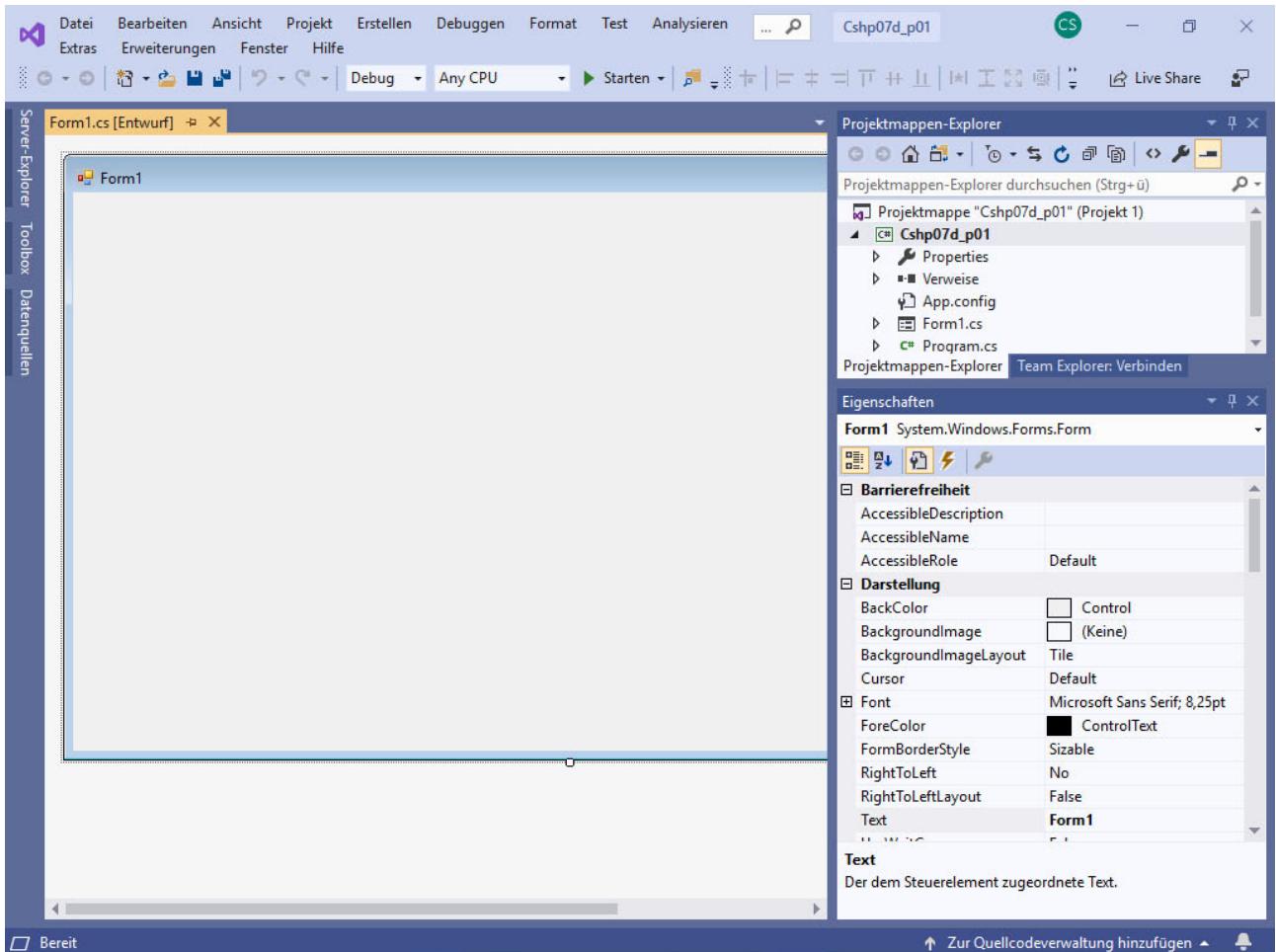


Abb. 2.1: Das eingeblendete Fenster **Eigenschaften**
(rechts unten in der Abbildung)

Über das Eigenschaftenfenster setzen Sie die Eigenschaften für das aktuell markierte Steuerelement – in unserem Beispiel also für das Formular **Form1**. Die einzelnen Eigenschaften werden dabei in mehreren Gruppen alphabetisch sortiert als Tabelle angezeigt. Dabei werden zum Beispiel alle Eigenschaften, die die Darstellung eines Steuerelements betreffen, in einer Gruppe **Darstellung** zusammengefasst.

Links in der Tabelle finden Sie den Namen der Eigenschaft und rechts den aktuellen Wert. Sowohl die Eigenschaften als auch die aktuellen Werte werden allerdings in vielen Fällen als verkürzte englische Begriffe angezeigt. So steht die Eigenschaft **BackColor** zum Beispiel für *background color* – die Hintergrundfarbe.

Neben den Eigenschaften können Sie über das Eigenschaftenfenster auch die Reaktion auf Ereignisse wie einen Mausklick festlegen. Dazu klicken Sie einmal auf das Symbol **Ereignisse** in der Symbolleiste des Fensters.

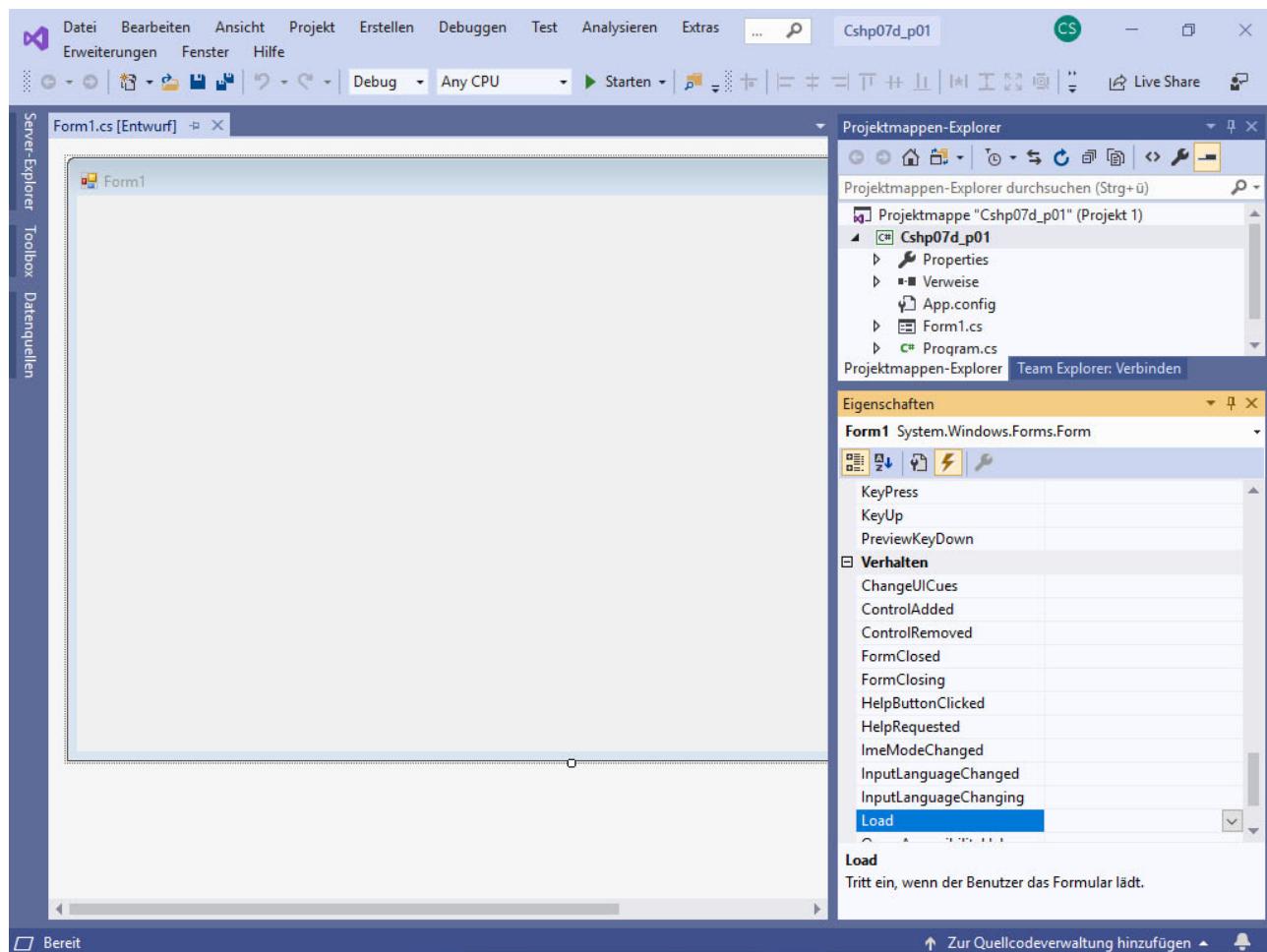


Abb. 2.2: Das Fenster **Eigenschaften** mit der Anzeige der Ereignisse

Probieren Sie das einmal aus. Lassen Sie die Ereignisse im Eigenschaftenfenster anzeigen. Wechseln Sie dann wieder zurück in die Anzeige der Eigenschaften. Klicken Sie dazu auf das Symbol **Eigenschaften** in der Symbolleiste des Eigenschaftenfensters.

Hinweis:

Was sich genau hinter den Ereignissen verbirgt, erfahren Sie gleich.

Neben der gruppierten Darstellung können Sie die Eigenschaften und Ereignisse im Eigenschaftenfenster auch alphabetisch sortiert anzeigen lassen. Das ist zum Beispiel dann interessant, wenn Sie den Namen einer Eigenschaft oder eines Ereignisses bereits kennen und direkt darauf zugreifen möchten. Die alphabetische Darstellung können Sie über das Symbol **Alphabetisch** oben links im Eigenschaftenfenster aktivieren.



Das Eigenschaftenfenster ist ein zentrales Element beim Erstellen von Windows Forms-Anwendungen. Hier legen Sie sowohl die Eigenschaften als auch die Reaktion auf Ereignisse fest.

Ändern wir nun den Text in der Titelleiste des Formulars. Klicken Sie bitte mit der linken Maustaste in das Feld rechts neben der Eigenschaft **Text**. Diese Eigenschaft befindet sich unten in der Gruppe **Darstellung** und sollte bereits durch eine blaue Hinterlegung markiert sein. Löschen Sie dann den Eintrag **Form1** und geben Sie den neuen Text **Hallo Welt** über die Tastatur ein. Drücken Sie anschließend die Eingabetaste.

Das Formular sollte jetzt so aussehen:

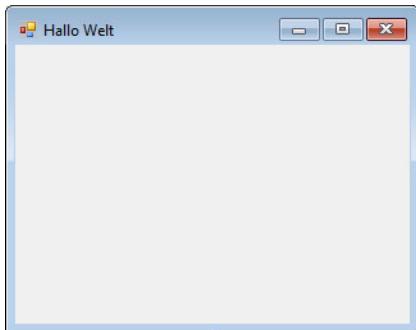


Abb. 2.3: Der geänderte Titel des Formulars

Wie Sie sehen, hat sich durch das Ändern der Eigenschaft automatisch der Text in der Titelleiste des Formulars verändert.

Mit dieser Technik können Sie auch noch weitere Eigenschaften des Formulars einstellen – zum Beispiel die Größe in Pixeln⁶. Die entsprechende Eigenschaft heißt **Size**⁷ und befindet sich in der Gruppe **Layout**. Verschieben Sie jetzt bitte die Anzeige im Eigenschaftenfenster, bis die entsprechende Eigenschaft angezeigt wird.

6. *Pixel* ist ein Kunstwort aus *picture* und *element*. Ein Pixel steht für einen Bildpunkt.

7. *Size* bedeutet übersetzt „Größe“.

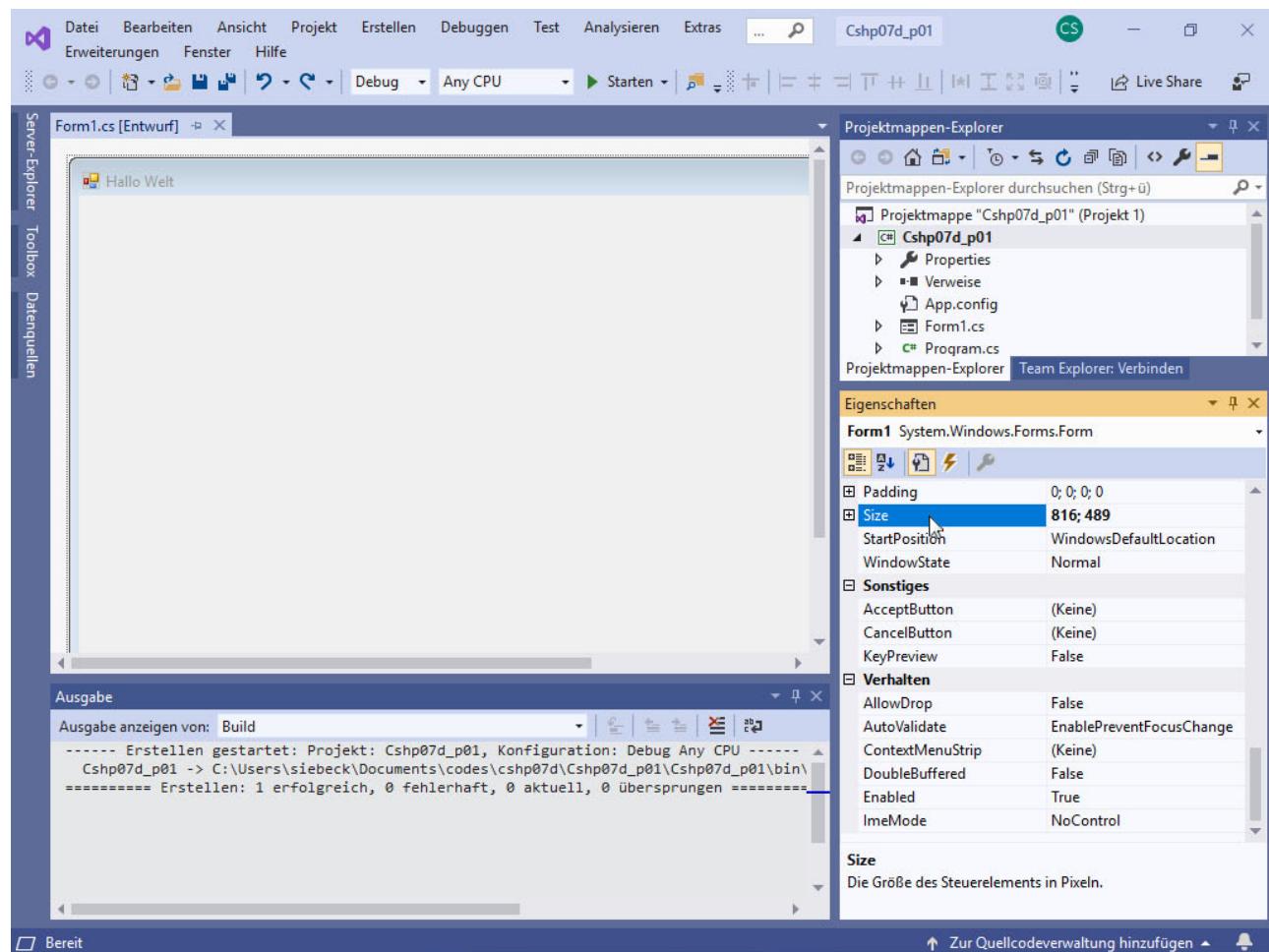


Abb. 2.4: Die Eigenschaft **Size**
(der Eintrag befindet sich rechts in der Abbildung am Mauszeiger)

Wie Sie sehen, befindet sich vor der Eigenschaft ein kleines Plussymbol. An diesem Symbol erkennen Sie, dass es für die Eigenschaft noch Untereigenschaften gibt. Diese Untereigenschaften werden angezeigt, wenn Sie einmal auf das Symbol klicken. Probieren Sie das bitte. Lassen Sie die Untereigenschaften für die Eigenschaft **Size** anzeigen.

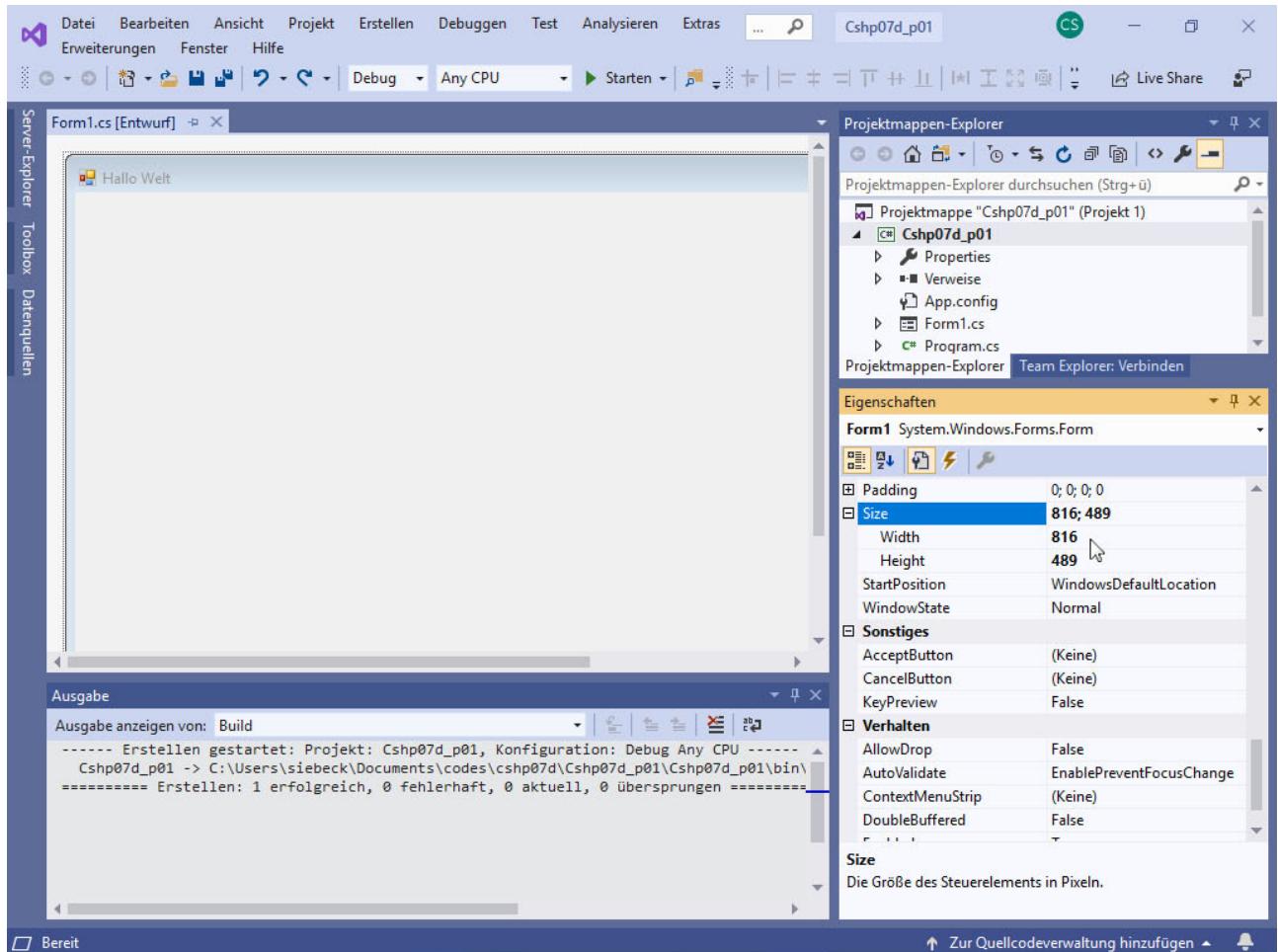


Abb. 2.5: Die Untereigenschaften der Eigenschaft **Size**

Jetzt können Sie über die Eigenschaften **Width** und **Height**⁸ die Breite und Höhe des Formulars einstellen. In unserem Beispiel soll das Formular 250 Pixel breit und 100 Pixel hoch sein. Setzen Sie die beiden Eigenschaften bitte auf die entsprechenden Werte.

Das Formular sollte nun so aussehen:

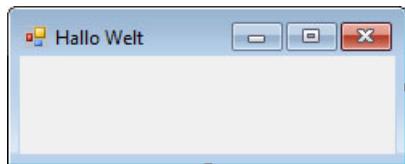


Abb. 2.6: Das Formular mit der neuen Breite und Höhe

Die Größe des Formulars können Sie übrigens auch im Designer mit der Maus festlegen. Ziehen Sie dazu einfach mit gedrückter linker Maustaste an einem der Anfasser am Rand des Formulars. Wenn das Formular die gewünschte Größe erreicht hat, lassen Sie die Maustaste wieder los. Die Eigenschaften **Width** und **Height** werden dann automatisch angepasst.

8. *Width* bedeutet übersetzt so viel wie „Breite“ und *height* „Höhe“.

Speichern Sie jetzt bitte die Änderungen an dem Projekt. Wenn Sie wollen, können Sie die Anwendung ja auch noch einmal zum Test ausführen lassen.

Tipps zum Arbeiten mit dem Eigenschaftenfenster

Tipp 1: Gruppen ausblenden

Die zahlreichen Einträge im Eigenschaftenfenster sorgen gerade zu Beginn schon einmal dafür, dass man etwas den Überblick verliert. Sie können die Anzeige aber gezielt auf bestimmte wichtige Gruppen wie zum Beispiel **Darstellung** oder **Layout** beschränken. Dazu blenden Sie die Gruppen, die Sie nicht vollständig anzeigen möchten, einfach durch einen Mausklick auf das Symbol im Gruppenkopf aus. Alternativ können Sie auch auf den Text im Gruppenkopf doppelklicken.

Visual Studio zeigt Ihnen dann nur noch den Kopf der Gruppe an. Durch einen weiteren Mausklick auf das Symbol oder einen weiteren Doppelklick auf den Text im Gruppenkopf können Sie jederzeit auch wieder die Einträge der Gruppe anzeigen lassen.

Tipp 2: Hilfe zu einer Eigenschaft anzeigen

Wenn Sie nicht genau wissen, was sich hinter einer Eigenschaft verbirgt, stellen Sie die Einfügemarkie in die Zeile der Eigenschaft. Visual Studio zeigt Ihnen dann erste Informationen im Bereich ganz unten im Eigenschaftenfenster an.

Tipp 3: Besondere Kennzeichen für die Eingabefelder

Für viele Eigenschaften – zum Beispiel bei **BackColor** für die Hintergrundfarbe – können Sie nur bestimmte Werte eintragen. Das erkennen Sie daran, dass Visual Studio ein Kombinationsfeld anzeigt, wenn Sie den Wert der Eigenschaft ändern möchten. Durch einen Mausklick auf das Symbol hinten im Kombinationsfeld können Sie dann die Liste öffnen und den gewünschten Eintrag auswählen.

Für einige Eigenschaften – wie zum Beispiel **Icon** für das Symbol links in der Titelleiste des Formulars – erfolgt die Auswahl über einen Dialog. Diesen Dialog öffnen Sie durch einen Mausklick auf das Symbol , das hinten im Eingabefeld der Eigenschaft eingebettet wird.

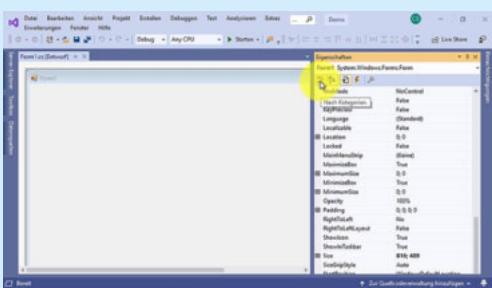
Tipp 4: Welches Steuerelement ist gerade ausgewählt?

Für welches Steuerelement Sie gerade die Eigenschaften einstellen, erkennen Sie zum einen an den Anfassern beziehungsweise Markierungen im Designer. Außerdem wird das aktuelle Steuerelement im Kombinationsfeld direkt unterhalb der Titelleiste des Eigenschaftenfensters angezeigt. Über dieses Kombinationsfeld können Sie auch sehr schnell das aktuelle Steuerelement wechseln. Das wird vor allem später interessant, wenn Sie mit sehr vielen Steuerelementen in einem Formular arbeiten.

Tipp 5: Werte von Untereigenschaften direkt setzen

Sie können die Werte von Untereigenschaften in vielen Fällen auch direkt über das Feld der übergeordneten Eigenschaft setzen. Dazu überschreiben Sie einfach die angezeigten Werte und trennen die einzelnen Einträge durch ein Semikolon.

Beim direkten Ändern müssen Sie allerdings wissen, welcher Eintrag zu welcher Untereigenschaft gehört. Daher ist es gerade zu Beginn häufig einfacher, die Untereigenschaften einzublenden und die Werte dann getrennt einzustellen.



In diesem Video stellen wir Ihnen das Eigenschaftenfenster im Einsatz vor.



www.dfz.media/3qd8a3

Video 2.1: Arbeiten mit dem Eigenschaftenfenster

2.2 Steuerelemente einfügen und positionieren

Nachdem wir jetzt den Titel und die Größe des Formulars geändert haben, wollen wir zwei Steuerelemente in das Formular einfügen: eine Schaltfläche zum Beenden der Anwendung und ein Feld für den Text „Hallo Welt“. Dazu verwenden wir die Toolbox.

Lassen Sie die Toolbox anzeigen. Klicken Sie dazu mit der Maus auf die entsprechende Registerzunge am linken Rand des Fensters von Visual Studio. Falls die Toolbox Ihr Formular im Designer zudeckt, klicken Sie auf das Symbol **Automatisch im Hintergrund** in der Titelleiste der Toolbox, damit sie dauerhaft eingeblendet wird.

Verschieben Sie dann gegebenenfalls die Anzeige in der Toolbox, bis die Gruppe **Allgemeine Steuerelemente** angezeigt wird, und blenden Sie die Einträge dieser Gruppe ein. Ziehen Sie anschließend das Steuerelement **Button**⁹ mit gedrückter linker Maustaste unten in die Mitte des Formulars und lassen Sie die Maustaste wieder los. Das Formular sollte nun ungefähr so aussehen:

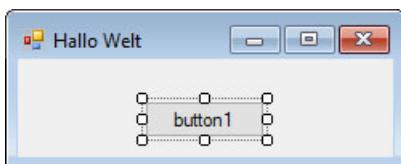


Abb. 2.7: Die Schaltfläche im Formular

Hinweis:

Wenn Sie die Toolbox nicht über das Symbol **Automatisch im Hintergrund** dauerhaft anzeigen lassen, wird sie nach dem Einfügen des Steuerelements automatisch wieder ausgeblendet.

Wenn Sie beim Positionieren eines Steuerelements einmal genau hinsehen, werden Sie feststellen, dass zum Teil blaue Linien erscheinen. Diese Linien sind eine Art Abstandshalter und helfen Ihnen ein wenig beim Layout des Formulars.

9. Wörtlich übersetzt bedeutet *button* „Knopf“.

Falls Ihnen die Position des Steuerelements nach dem Ablegen nicht gefällt, können Sie es auch jederzeit noch nachträglich verschieben. Stellen Sie dazu den Mauszeiger auf das Steuerelement und ziehen Sie es mit gedrückter linker Maustaste an die gewünschte Position im Formular.



Bitte beachten Sie:

Sie müssen die Maus mitten auf das Steuerelement stellen, bevor Sie es verschieben. Andernfalls ziehen Sie unter Umständen an einem der Anfasser und verändern so die Größe des Steuerelements. Vor allem bei kleineren Steuerelementen müssen Sie daher genau zielen, um das Element zu verschieben.

Einmal eingefügte Steuerelemente können Sie auch jederzeit wieder löschen. Dazu markieren Sie zunächst das gewünschte Element im Formular mit einem Mausklick und drücken dann die Taste **Entf**.

Fügen Sie jetzt bitte das Steuerelement für den Text ein. Es trägt den Namen **Label¹⁰** und befindet sich in der Gruppe **Allgemeine Steuerelemente** der Toolbox. Positionieren Sie das Label bitte mittig oberhalb der Schaltfläche. Ihr Formular sollte nun ungefähr so aussehen:

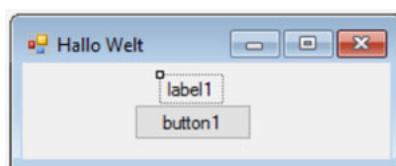


Abb. 2.8: Die beiden Steuerelemente im Formular

Nun müssen wir noch die Standardtexte auf den beiden Steuerelementen ändern. Das erfolgt jeweils über die Eigenschaft **Text**. Für das Label verwenden Sie bitte den Text „Hallo Welt“ und für die Schaltfläche den Text „Beenden“.

Hinweis:

Denken Sie bitte daran, dass Sie nach dem Ändern der Eigenschaft für das erste Steuerelement zunächst das zweite Steuerelement auswählen müssen. Dazu klicken Sie entweder mit der Maus auf das gewünschte Element im Formular oder Sie wählen das Steuerelement über das Kombinationsfeld oben im Eigenschaftenfenster aus.

Verschieben Sie nach den Änderungen das Label gegebenenfalls noch einmal ein wenig, damit es wieder mittig über der Schaltfläche steht. Das fertige Formular sollte dann so aussehen:



Abb. 2.9: Das fertige Formular

10. Wörtlich übersetzt bedeutet *label* „Schild“ oder „Etikett“.

Speichern Sie jetzt alle Änderungen an dem Projekt und lassen Sie das Programm ausführen. Sie werden sehen, dass Sie zum Beispiel die Schaltfläche anklicken können. Allerdings hat der Klick noch keinerlei Wirkung. Sie müssen das Programm also über das Schließensymbol ganz rechts in der Titelleiste oder über das Systemmenü beenden.

Tipps zum Einfügen von Steuerelementen

Tipp 1: Anzeige der Toolbox übersichtlicher gestalten

Auch die einzelnen Gruppen in der Toolbox können Sie über das Symbol oder den Text im Kopf einer Gruppe gezielt ein- und ausblenden. Die Gruppe **Alle Windows Forms** können Sie zum Beispiel zunächst einmal ausgeblendet lassen. Denn in dieser Gruppe finden sich im Wesentlichen die Steuerelemente aus den anderen Gruppen in alphabetischer Reihenfolge.

Tipp 2: Steuerelemente ausrichten lassen

Sie können die Steuerelemente auch von Visual Studio ausrichten lassen. Mit den Funktionen **Format/Auf Formular zentrieren/Horizontal** und **Format/Auf Formular zentrieren/Vertikal** lässt sich ein Steuerelement zum Beispiel exakt in der Mitte des Formulars positionieren.

Über die Symbole in der Symbolleiste **Layout** können Sie auch mehrere Steuerelemente untereinander ausrichten. Markieren Sie dazu zunächst die gewünschten Steuerelemente, indem Sie mit gedrückter Taste nacheinander auf die Elemente klicken. Wählen Sie dann die gewünschte Ausrichtung über das entsprechende Symbol aus. Da die Symbole weitgehend selbsterklärend sind, wollen wir Ihnen die einzelnen Funktionen hier nicht weiter vorstellen. Probieren Sie die Wirkung am besten einmal selber aus.

Tipp 3: Steuerelemente durch einen einfachen Mausklick einfügen

Sie können ein Steuerelement auch mit einem einfachen Mausklick aus der Toolbox in ein Formular einfügen. Klicken Sie dazu zunächst auf das gewünschte Steuerelement in der Toolbox und anschließend an die Position im Formular, an der das Steuerelement abgelegt werden soll.

Tipp 4: Steuerelemente mehrfach einfügen

Wenn Sie ein Steuerelement mehrfach einfügen wollen, können Sie sich mit einem kleinen Trick das ständige Markieren in der Toolbox sparen. Halten Sie einfach beim ersten Einfügen die Taste **Strg** gedrückt. Sie können dann so lange dasselbe Steuerelement noch einmal einfügen, bis Sie das Element **Zeiger** oben in der Liste auswählen.

2.3 Verarbeitung von Ereignissen

Bisher haben wir in unserem Programm lediglich die verschiedenen Steuerelemente eingefügt. Jetzt müssen wir noch dafür sorgen, dass ein Mausklick auf die Schaltfläche **Beenden** das Programm auch tatsächlich beendet. Wir müssen also die Verarbeitung für das Ereignis **Mausklick** der Schaltfläche programmieren. Bevor wir uns damit beschäftigen, wollen wir uns zuerst einmal etwas genauer ansehen, was sich überhaupt hinter den Ereignissen verbirgt.



Ereignisse sind eine Art Signal, das von bestimmten Aktionen ausgelöst wird. Zu den Aktionen, die ein Ereignis auslösen, gehören zum Beispiel das Bewegen der Maus, das Klicken, das Doppelklicken oder auch eine Tastatureingabe.

Ereignisse können in Objekten bestimmte Methoden aufrufen und so dazu führen, dass – abhängig von einem Ereignis – bestimmte Befehle ausgeführt werden. Ein typisches Beispiel ist ein Doppelklick auf eine Datei, der das Öffnen der Datei veranlasst. Hier führt das Ereignis **Doppelklicken** dazu, dass der Befehl „Datei öffnen“ ausgeführt wird. Ein anderes Beispiel wäre ein Mausklick auf eine Schaltfläche. Hier führt das Ereignis **Klicken** dazu, dass die Anweisungen ausgeführt werden, die für die Schaltfläche programmiert worden sind.

Exkurs: Vorteile der Ereignisverarbeitung

Die Verarbeitung von Ereignissen – im Fachjargon auch mit dem englischen Begriff **event handling** bezeichnet – ermöglicht eine völlig neue Form der Programmierung.

Ältere Programmiersprachen arbeiten strikt **sequenziell**; das heißt, die Anweisungen werden exakt in der Reihenfolge ausgeführt, in der sie im Quelltext vorhanden sind. Dabei können zwar unterschiedliche Wege durch den Quelltext durchlaufen werden – zum Beispiel durch den Aufruf verschiedener Methoden abhängig von einer Tastatureingabe, grundsätzlich ist der Ablauf des Programms aber fest vorgegeben. Diese sequenzielle Programmverarbeitung erfolgte zum Beispiel bei sämtlichen Konsolenprogrammen, die Sie bisher programmiert haben. Hier wurden immer die Anweisungen in der Methode `Main()` von oben nach unten ausgeführt.

Die Ereignisverarbeitung dagegen erlaubt eine flexible Reaktion eines Programms auf eintretende Ereignisse. Das heißt, der Programmablauf ist nicht von vornherein festgelegt, sondern kann von außen durch die Reaktion auf Ereignisse bestimmt werden.

Neben der flexiblen Reaktion kann die Ereignisverarbeitung auch zu einer höheren Betriebssicherheit von Programmen führen, da Aktionen erst dann ausgeführt werden, wenn ein Ereignis tatsächlich eingetreten ist. Diese Fähigkeit ist zum Beispiel bei der Programmierung von Internet-Anwendungen enorm wichtig, da hier nicht davon ausgegangen werden kann, dass ein begonnenes Programm auch komplett bis zum Ende durchläuft. Möglicherweise geht mitten im Ablauf des Programms die Verbindung zum Internet verloren oder der Anwender schließt den Browser, ohne das Ende des Programms abzuwarten.

Welche Ereignisse in einem Programm verarbeitet werden können, hängt von der eingesetzten Programmiersprache ab. So können die meisten modernen Programmiersprachen wie C#, Java oder Visual Basic Standardereignisse wie Mausaktionen oder Tastatureingaben verarbeiten. Andere hoch spezialisierte Programmiersprachen sind auch in der Lage, auf sehr spezielle Ereignisse wie das Verarbeiten von Datenbanktabellen zu reagieren.

So viel zu den Vorteilen der Ereignisverarbeitung.

In unserem Beispielprogramm soll das Ereignis **Klicken** für die Schaltfläche **Beenden** dazu führen, dass das Programm beendet wird. Wir müssen also für die Schaltfläche eine Reaktion auf das Ereignis **Klicken** definieren. Dazu benutzen wir wieder das Eigenschaftenfenster.

Markieren Sie die Schaltfläche im Formular. Klicken Sie dann im Eigenschaftenfenster auf das Symbol **Ereignisse** .

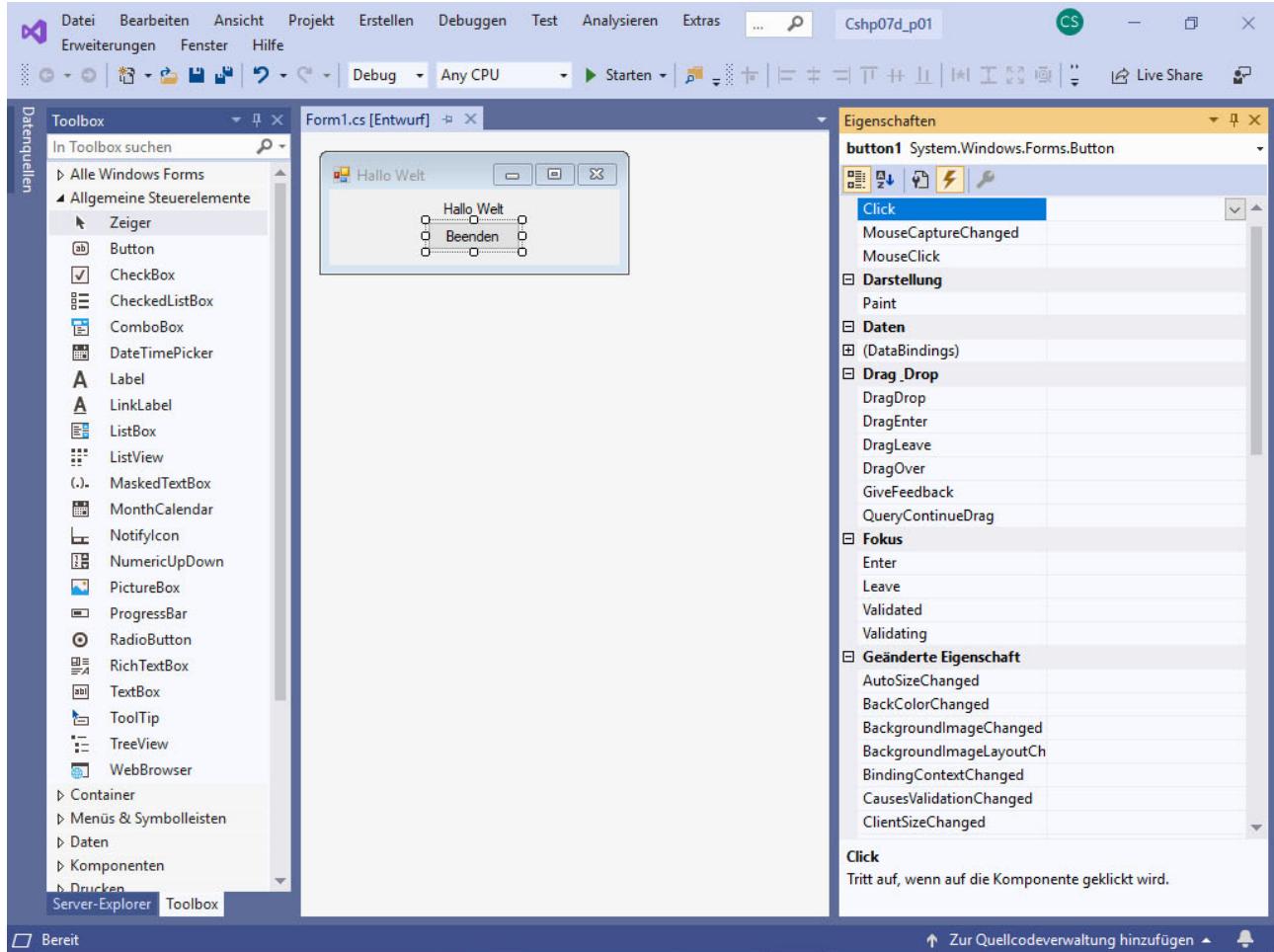


Abb. 2.10: Die Ereignisse für eine Schaltfläche im Eigenschaftenfenster
(der Projektmappen-Explorer ist zur besseren Übersicht ausgeblendet)

Im Eigenschaftenfenster finden Sie jetzt eine Liste der Ereignisse, auf die das Steuerelement reagieren kann. In der linken Spalte steht die Bezeichnung des Ereignisses – allerdings wieder in englischer Sprache und zum Teil auch verkürzt. In der rechten Spalte des jeweiligen Ereignisses können Sie festlegen, was beim Eintreten des Ereignisses geschehen soll.

In unserem Fall können wir zwischen zwei verschiedenen Ereignissen wählen – und zwar zwischen **Click** und **MouseClick**¹¹. Das Ereignis **Click** ist dabei etwas allgemeiner und wird zum Beispiel auch dann ausgelöst, wenn das Steuerelement markiert ist und Sie die Eingabetaste drücken. Das Ereignis **MouseClick** dagegen wird nur bei einem Mausklick ausgelöst.

Was genau beim Eintreten eines Ereignisses passieren soll, müssen Sie selbst programmieren. Dazu doppelklicken Sie mit der Maus in das Feld rechts neben dem Ereignis.

Visual Studio erstellt dann automatisch in der Datei **Form1.cs** eine Methode für die Ereignisverarbeitung und zeigt diese Methode im Editor an. Der Name der Methode hängt dabei vom Namen des Steuerelements und vom Namen des Ereignisses ab. In unserem Fall würde zum Beispiel eine Methode `Button1_Click()` für das Ereignis **Click** der Schaltfläche angelegt.

```

 6  using System.Linq;
 7  using System.Text;
 8  using System.Threading.Tasks;
 9  using System.Windows.Forms;
10
11  namespace Cshp07d_p01
12  {
13      public partial class Form1 : Form
14      {
15          public Form1()
16          {
17              InitializeComponent();
18          }
19
20          private void Button1_Click(object sender, EventArgs e)
21          {
22          }
23      }
24  }

```

Abb. 2.11: Die Methode `Button1_Click()` im Editor
(zur besseren Übersicht haben wir die anderen Bereiche im Fenster ausgeblendet)

Hinweis:

Eine Windows Forms-Anwendung besteht in der Regel aus mehreren Dateien. Die Datei **Form1.cs** enthält unter anderem die Anweisungen für die Reaktion auf Ereignisse. Die Datei **Form1.Designer.cs** enthält Anweisungen, die das Formular und die

11. *Click* bedeutet übersetzt „Klick“. *MouseClick* steht für „Mausklick“.

Steuerelemente erzeugen, und die Datei **Program.cs** Anweisungen für das Ausführen des Programms. An den Dateien **Form1.Designer.cs** und **Program.cs** müssen Sie normalerweise selbst keine Änderungen vornehmen.

Schauen wir die automatisch angelegte Methode `Button1_Click()` einmal genauer an. Sie besteht aus den folgenden Zeilen:

```
private void Button1_Click(object sender, EventArgs e)
{
}
```

Code 2.1: Der Quelltext der Methode `Button1_Click()`

Bei der ersten Zeile handelt es sich um den Kopf der Methode. Wie Sie sehen, ist die Sichtbarkeit der Methode privat. Als Rückgabetyp wird `void` vereinbart – also „Nichts“. Dann folgen der Name der Methode und die Parameterliste. Der erste Parameter `sender` steht dabei für das Steuerelement, das das Ereignis ausgelöst hat. Über den zweiten Parameter `e` werden weitere Informationen an das Ereignis übergeben – zum Beispiel, welche Maustaste gedrückt wurde. Die Übergabe der Parameter beziehungsweise der Argumente erfolgt automatisch beim Auslösen des Ereignisses. Darum müssen Sie sich nicht weiter kümmern.

Danach folgt der leere Rumpf der Methode. Hier geben Sie die Anweisungen ein, die beim Eintreten des Ereignisses ausgeführt werden sollen.

Bitte beachten Sie unbedingt:

Die Methoden werden direkt in der Klasse für das Formular angelegt. Gehen Sie deshalb mit der nötigen Vorsicht ans Werk. Achten Sie zum Beispiel sorgfältig darauf, dass Sie keine der beiden letzten schließenden geschweiften Klammern im Quelltext überschreiben. Diese Klammern gehören **nicht** zu der Methode, sondern zur Vereinbarung der gesamten Klasse beziehungsweise zur Vereinbarung des Namensraums für die Anwendung.



Zur Sicherheit sollten Sie an den automatisch angelegten Quelltexten zunächst keinerlei Änderungen vornehmen. Alle erforderlichen Daten werden von Visual Studio selbst aktualisiert, sobald Sie ein neues Steuerelement in das Formular einfügen oder die Eigenschaften eines vorhandenen Steuerelements verändern.

Wenn Details der Klasse für das Formular Sie interessieren, sehen Sie sich ruhig auch den Quelltext in der Datei **Form1.Designer.cs** an. Sie werden hier zum Beispiel auch die Vereinbarungen der beiden Steuerelemente wiederfinden, die wir über die Tool-box in das Formular eingefügt haben. Nehmen Sie aber auf keinen Fall Änderungen vor. Andernfalls funktioniert möglicherweise die gesamte Anwendung nicht mehr.

Hinweis:

In der Klasse finden Sie an vielen Stellen eine `this`-Referenz. Diese Referenz wird automatisch erzeugt und zeigt – wie Sie ja bereits wissen – immer auf die aktuelle Instanz der Klasse.

In unserem Beispiel soll das Formular und damit auch die Anwendung geschlossen werden, wenn auf die Schaltfläche **Beenden** geklickt wird. Dazu muss beim Klicken auf die Schaltfläche die Methode `Close()`¹² für das Formular ausgeführt werden.

Die vollständige Methode `Button1_Click()` sieht dann so aus:

```
private void Button1_Click(object sender, EventArgs e)
{
    Close();
}
```

Code 2.2: Die vollständige Methode `Button1_Click()`



Noch einmal, weil es so wichtig ist:

Achten Sie beim Eingeben der Anweisung bitte peinlich genau darauf, dass Sie die Anweisung im Rumpf der Methode eingeben. Normalerweise steht die Einfügemarke bereits an der richtigen Stelle – nämlich hinter der öffnenden geschweiften Klammern für den Rumpf der Methode. Kontrollieren Sie aber vor der Eingabe noch einmal zur Sicherheit, ob Sie sich wirklich an der richtigen Position befinden.

Übernehmen Sie bitte den Aufruf der Methode `Close()` in die Methode für das Ereignis `Click` der Schaltfläche. Wenn Sie möchten, können Sie dabei auch die `this`-Referenz noch einmal ausdrücklich vor den Namen der Methode setzen. Die Anweisung würde dann so aussehen:

```
this.Close();
```

Da die `this`-Referenz in diesem Fall aber für das Fenster der Anwendung beziehungsweise die aktuelle Instanz der Klasse des Fensters steht und der Compiler ohne die Angabe die Methode sowieso für das Fenster der Anwendung ausführt, können Sie die `this`-Referenz in vielen Fällen auch weglassen.

Speichern Sie die Änderungen und testen Sie die Anwendung noch einmal. Jetzt sollte sich das Programm auch über die Schaltfläche **Beenden** schließen lassen.

Tipp:

Wenn Sie einmal versehentlich die Methode für ein falsches Ereignis angelegt haben, löschen Sie im Eigenschaftenfenster den Eintrag rechts im Feld hinter dem Ereignis. Visual Studio entfernt dann auch die leere Methode wieder aus dem Quelltext. Das funktioniert allerdings nur dann, wenn sich in der Methode noch keine Anweisungen befinden.

Damit haben Sie Ihr erstes echtes Windows-Programm erstellt. Im nächsten Kapitel werden wir uns noch mit einigen weiteren Standardsteuerelementen beschäftigen – zum Beispiel mit den Kombinationsfeldern.

12. *Close* bedeutet übersetzt „Schließe“.

Zusammenfassung

Die Eigenschaften von Steuerelementen legen Sie über das Eigenschaftenfenster des Designers fest.

Ereignisse sind eine Art Signal, das von bestimmten Aktionen ausgelöst wird. Zu den Aktionen, die ein Ereignis auslösen, gehören zum Beispiel das Klicken und Doppelklicken mit der Maus oder eine Tastatureingabe.

Ereignisse können in Objekten bestimmte Methoden aufrufen und so dazu führen, dass – abhängig von einem Ereignis – bestimmte Befehle ausgeführt werden.

Die Reaktion auf ein Ereignis legen Sie ebenfalls im Eigenschaftenfenster fest.

Aufgaben zur Selbstüberprüfung

- 2.1 Wie heißt die Eigenschaft, die den Text in der Titelleiste eines Formulars festlegt?

- 2.2 Vor einer Eigenschaft im Eigenschaftenfenster wird ein kleines Symbol angezeigt. Welche Bedeutung hat dieses Symbol?

- 2.3 Wie können Sie im Eigenschaftenfenster die Reaktion auf einen Mausklick festlegen? Beschreiben Sie bitte alle erforderlichen Schritte.

2.4 In welcher Datei werden die Methoden für die Ereignisverarbeitung angelegt?



3 Einige kleine Spielereien

In diesem Kapitel werden wir uns an einem kleinen Beispielprogramm weitere Möglichkeiten zum Umgang mit Steuerelementen ansehen. Außerdem beschäftigen wir uns mit einigen anderen Standardsteuerelementen – zum Beispiel mit Kombinationsfeldern und Kontrollkästchen.

Legen Sie bitte zunächst ein neues Projekt für eine Windows Forms-Anwendung an. Rufen Sie dazu die Funktion zum Anlegen eines neuen Projekts auf und wählen Sie dann im Fenster **Neues Projekt erstellen** den Eintrag **Windows Forms-App (.NET Framework)** für die Programmiersprache C#.

Hinweis:

Wir verwenden in unserem Beispiel den Namen **Cshp07d_p02** für das Projekt.

Ändern Sie den Titel des Formulars in **Spielereien** und setzen Sie die Höhe des Formulars auf 220 Pixel. Die Breite ändern Sie bitte in 300 Pixel. Fügen Sie anschließend am unteren Rand zwei Schaltflächen ein. Die eine Schaltfläche soll den Text **Kopieren** erhalten und die andere Schaltfläche den Text **Beenden**. Sorgen Sie außerdem dafür, dass beim Anklicken der Schaltfläche **Beenden** das Formular geschlossen wird. Erstellen Sie dazu eine Methode, die beim Ereignis **Click** für die Schaltfläche die Anweisung `Close()` ausführt. Wenn Sie nicht mehr genau wissen, wie Sie dabei vorgehen müssen, lesen Sie bitte noch einmal im letzten Kapitel nach.

Das Formular im Rohbau sollte ungefähr so aussehen:

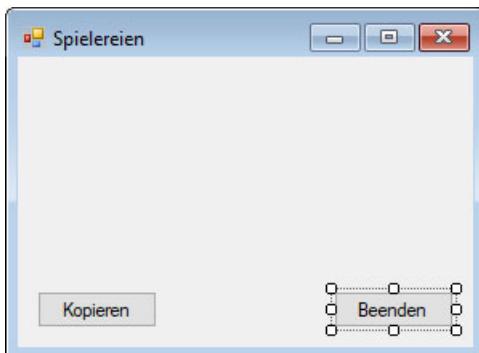


Abb. 3.1: Das Formular im „Rohbau“

Hinweis:

Visual Studio setzt den Namen eines Steuerelements immer aus dem Typ und einer laufenden Nummer zusammen. Die erste Schaltfläche in einem Formular erhält zum Beispiel den Namen `button1`, die zweite Schaltfläche den Namen `button2` und so weiter. Damit wird eine eindeutige Zuordnung unter Umständen schwierig – vor allem dann, wenn Sie viele identische Steuerelemente verwenden.

Sie sollten sich daher angewöhnen, sprechende Namen für die Steuerelemente zu vergeben. Dazu ändern Sie die Eigenschaft **(Name)** in der Gruppe **Entwurf** des Eigenschaftenfensters. Für die Schaltfläche zum Kopieren könnten Sie beispielsweise

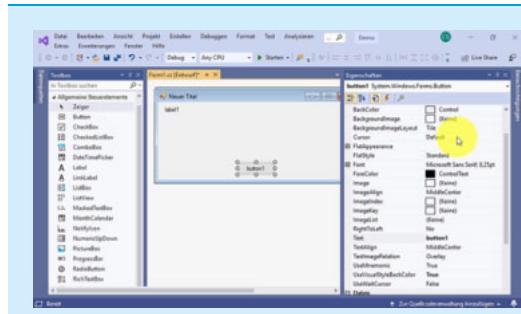
den Namen `buttonKopieren` verwenden und für die Schaltfläche zum Beenden den Namen `buttonBeenden`. Bitte denken Sie dann aber auch daran, dass Sie die Steuerelemente im Quelltext über diese Namen ansprechen müssen.

Wenn Sie ein Formular nach dem Einfügen von Steuerelementen verkleinern, wird die Position der Steuerelemente nicht automatisch angepasst, da die Elemente am oberen und linken Rand verankert sind. Das kann dazu führen, dass die Steuerelemente scheinbar aus dem Formular verschwinden. Sie können sie aber ganz einfach wieder korrekt positionieren. Wählen Sie dazu zuerst das Steuerelement über das Kombinationsfeld oben im Eigenschaftenfenster aus. Ziehen Sie es dann mit der Maus wieder an die gewünschte Position im Formular.

Auch beim Positionieren von mehreren Steuerelementen unterstützt Sie der Designer von Visual Studio durch Führungslinien. So können Sie zum Beispiel an der blauen Linie erkennen, dass zwei Steuerelemente exakt auf derselben Höhe ausgerichtet sind. Probieren Sie die verschiedenen Positionierhilfen einfach einmal mit den beiden Schaltflächen aus.



dfz.media/6zwh2c



In diesem Video zeigen wir Ihnen, wie Sie Steuerelemente einfügen und positionieren.

www.dfz.media/6zwh2c

Video 3.1: Steuerelemente einfügen und positionieren

In den nächsten Schritten werden wir die Anwendung mit ein wenig Leben füllen.

3.1 Texte zwischen Labels kopieren

Beginnen wir mit dem Kopieren von Texten zwischen zwei Labels.

Fügen Sie oben in dem Formular zwei Labels ein. Ändern Sie die Eigenschaft (**Name**) des einen Labels bitte in `labelQuelle` und die Eigenschaft (**Name**) des anderen Labels in `labelZiel`.

Hinweis:

Sie können auch andere Namen für die Labels verwenden. Dann müssen Sie allerdings später in den Quelltexten ebenfalls Ihre eigenen Namen verwenden.

Richten Sie die Labels anschließend an den Schaltflächen unten im Formular aus und setzen Sie die Eigenschaft **Text** der Labels auf einen beliebigen Text. Achten Sie dabei aber bitte darauf, dass für das zweite Label ein wenig Platz nach rechts bleibt.

Das Formular könnte nun ungefähr so aussehen:

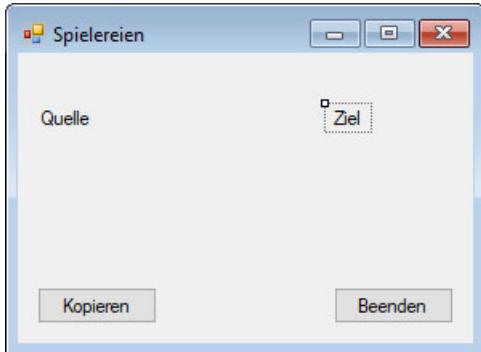


Abb. 3.2: Das Formular mit den beiden Labels

Beim Anklicken der Schaltfläche **Kopieren** soll jetzt der Text aus dem einen Label in das andere kopiert werden. Dazu müssen wir nichts weiter machen, als über eigene Anweisungen für das Ereignis **Click** der Schaltfläche die Eigenschaft **Text** des einen Labels auf den Wert der Eigenschaft **Text** des anderen Labels zu setzen. Damit wir den Text des jeweils anderen Labels nicht überschreiben, bringen wir ihn in einer Variablen vom Typ **string** in Sicherheit. Diese Variable legen wir lokal in der Methode beim Anklicken der Schaltfläche an.

Markieren Sie bitte die Schaltfläche **Kopieren** im Formular. Wechseln Sie dann über das Symbol Ereignisse in der Symbolleiste des Eigenschaftenfensters zu den Ereignissen. Doppelklicken Sie anschließend in das Feld rechts hinter dem Ereignis **Click** und übernehmen Sie den Quelltext aus dem folgenden Code für die Methode.

Tipp:

Die Methode für die Verarbeitung des Standardereignisses eines Steuerelements können Sie auch durch einen Doppelklick auf das Steuerelement im Designer erzeugen. Für eine Schaltfläche ist das Standardereignis zum Beispiel **Click** – also genau das Ereignis, das wir bearbeiten wollen.

```
string tempKette;
//den Text aus dem Ziel in Sicherheit bringen
tempKette = labelZiel.Text;
//den Text aus der Quelle in das Ziel kopieren
labelZiel.Text = labelQuelle.Text;
//den gesicherten Text in die Quelle kopieren
labelQuelle.Text = tempKette;
```

Code 3.1: Der Quelltext für die Methode `ButtonKopieren_Click()`

Hinweis:

Wir zeigen Ihnen hier nur die Anweisungen für den Rumpf. Der Kopf der Methode und auch die Klammern für den Rumpf werden automatisch von Visual Studio angelegt.

Richtig viel Neues gibt es in diesem Quelltext eigentlich nicht. Wir vereinbaren zunächst eine lokale Variable vom Typ `string`. Dieser Variablen weisen wir dann mit der Anweisung

```
tempKette = labelZiel.Text;
```

den aktuellen Wert der Eigenschaft `Text` aus dem Steuerelement `labelZiel` zu – also den Text, der gerade im Label steht.

Danach vertauschen wir dann die Texte in den beiden Labels durch die entsprechenden Zuweisungen.

Speichern Sie jetzt bitte die Änderungen und testen Sie das Programm. Wenn Sie alles richtig gemacht haben, wird nach jedem Mausklick auf die Schaltfläche **Kopieren** der Text in den Labels getauscht. Beim Klicken auf die Schaltfläche **Beenden** dagegen wird das Formular geschlossen.

Im nächsten Schritt wollen wir jetzt noch dafür sorgen, dass der Text nicht nur beim Anklicken der Schaltfläche **Kopieren** getauscht wird, sondern auch bei einem Doppelklick auf eins der beiden Labels. Dazu müssen wir das Ereignis **DoubleClick** der beiden Labels bearbeiten.

Wechseln Sie wieder zurück in den Designer. Klicken Sie dazu entweder auf die Registerzeile **Form1.cs [Entwurf]** oben im Editor oder wählen Sie im Kontextmenü des Quelltexts die Funktion **Ansicht-Designer**. Markieren Sie dann im Designer eines der beiden Labels und wechseln Sie zu den Ereignissen im Eigenschaftenfenster.

Da wir die Anweisungen für das Wechseln der Texte bereits programmiert haben, müssen wir nichts weiter machen, als dem Ereignis **DoubleClick** für den Doppelklick die Methode `ButtonKopieren_Click()` zuzuweisen. Diese Methode sorgt ja dafür, dass die Texte getauscht werden. Klicken Sie dazu mit der Maus in das Feld rechts neben dem Ereignis **DoubleClick** für das Label. Öffnen Sie dann die Liste der Methoden durch einen Mausklick auf das Symbol hinten im Kombinationsfeld und wählen Sie den Eintrag **ButtonKopieren_Click** aus.

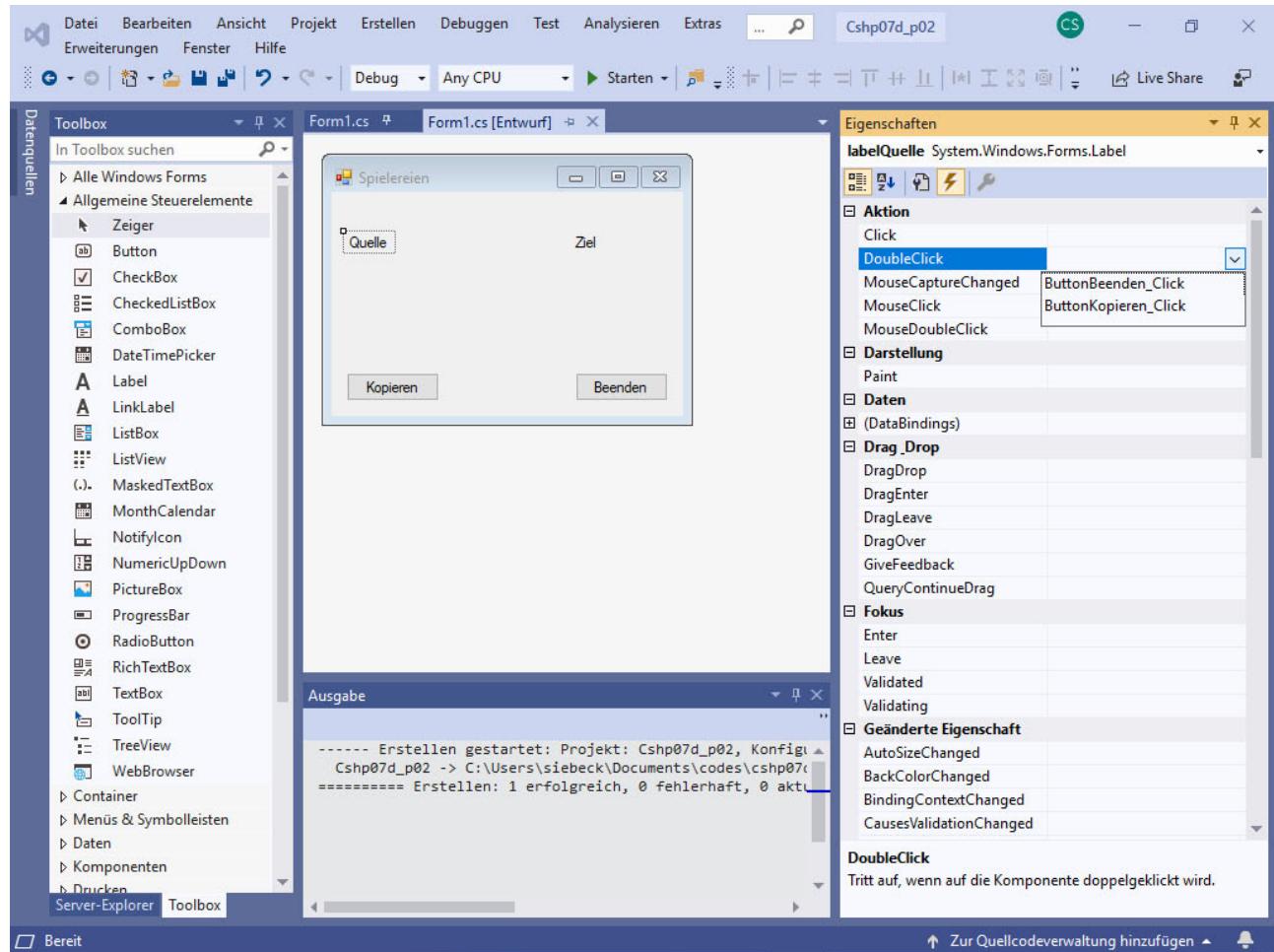


Abb. 3.3: Die Auswahl der Methoden für das Ereignis **DoubleClick** des Labels
(der Projektmappen-Explorer ist zur besseren Übersicht ausgeblendet)

Über das Kombinationsfeld für ein Ereignis eines Steuerelements können Sie auch bereits vorhandene Methoden für die Ereignisverarbeitung auswählen. In der Liste werden allerdings nur die Methoden angezeigt, die zum Steuerelement und zum Ereignis passen. Die Methode für das Ereignis **MouseDoubleClick** wird beim Ereignis **DoubleClick** zum Beispiel nicht angezeigt.



Nehmen Sie diese Änderung dann auch noch für das zweite Label vor. Speichern Sie anschließend das Projekt und testen Sie es noch einmal. Jetzt sollten die Texte auch nach einem Doppelklick auf eines der Labels getauscht werden.

3.2 Texte aus Kombinations- und Listenfeldern kopieren

Damit der Anwender die Texte in den beiden Labels verändern kann, bauen wir noch ein Kombinationsfeld und ein Listenfeld ein. Das Kombinationsfeld soll dabei Texte für das linke Label enthalten und das Listenfeld Texte für das rechte Label.

Hinweis:

Mit der Eingabe von freien Texten über Eingabefelder beschäftigen wir uns im nächsten Kapitel.



Sowohl Kombinationsfelder als auch Listenfelder bieten feste Einträge in Listenform an. Bei einem Kombinationsfeld wird allerdings immer nur der aktuell ausgewählte Eintrag angezeigt. Die Liste erscheint nur dann, wenn auf das Symbol hinten im Feld geklickt wird. Bei einem Listenfeld dagegen wird ständig die gesamte Liste angezeigt. Der aktuell ausgewählte Eintrag ist farbig hinterlegt.

Beginnen wir mit dem Einfügen des Kombinationsfelds. Öffnen Sie die Toolbox und wählen Sie den Eintrag **ComboBox**¹³ aus. Sie finden den Eintrag in der Gruppe **Allgemeine Steuerelemente**. Positionieren Sie das Kombinationsfeld dann links im Formular zwischen dem Label und der Schaltfläche.

Die Einträge für das Kombinationsfeld legen Sie über den **Zeichenfolgen-Editor** fest. Diesen Editor öffnen Sie durch einen Mausklick auf das Symbol hinten im Feld **Items**¹⁴ in der Gruppe **Daten** bei den Eigenschaften. Alternativ können Sie auch auf das Symbol rechts oben an dem Steuerelement im Designer klicken und bei den ComboBox-Aufgaben den Eintrag **Einträge bearbeiten...** auswählen.

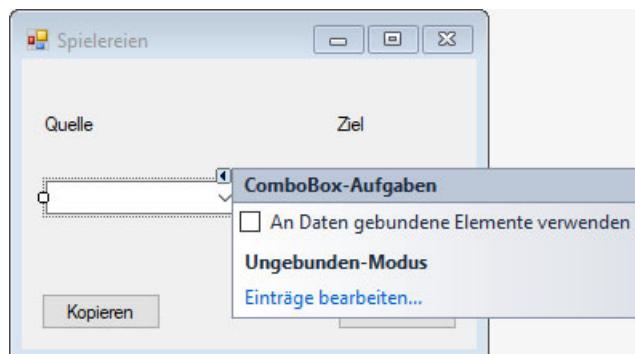


Abb. 3.4: Die ComboBox-Aufgaben

Öffnen Sie jetzt den Zeichenfolgen-Editor und legen Sie einige Einträge für das Kombinationsfeld an. Trennen Sie die einzelnen Einträge dabei jeweils mit der Eingabetaste. Wenn Sie fertig sind, klicken Sie auf die Schaltfläche **OK**, um den Zeichenfolgen-Editor wieder zu schließen.

13. *ComboBox* steht für *Combination box* – übersetzt etwa „Kombinationsfeld“.

14. *Items* bedeutet übersetzt so viel wie „Gegenstände, Dinge“ oder „Einträge“.

Bitte beachten Sie:

Im Entwurf werden die Einträge in einem Kombinationsfeld nicht angezeigt. Wenn Sie die Einträge sehen möchten, müssen Sie das Programm ausführen lassen.



Nun müssen wir den aktuell ausgewählten Eintrag aus dem Kombinationsfeld in das linke Label kopieren. Dabei hilft uns das Ereignis **Click** für das Kombinationsfeld nicht viel weiter, denn dieses Ereignis tritt ja bei jedem Mausklick in das Feld ein. Wir benutzen daher das Ereignis **SelectedValueChanged**¹⁵. Dieses Ereignis wird ausgelöst, wenn ein anderer Wert im Kombinationsfeld ausgewählt wurde. Sie finden das Ereignis im Eigenschaftenfenster bei den Ereignissen in der Gruppe **Geänderte Eigenschaft**.

In der Methode für das Ereignis beschaffen wir uns über den Ausdruck `comboBox1.SelectedItem.ToString()`¹⁶ den Text des ausgewählten Eintrags und weisen ihn dem Label `labelQuelle` zu. Die entsprechende Anweisung sieht so aus:

```
labelQuelle.Text = comboBox1.SelectedItem.ToString();
```

Code 3.2: Die Anweisung für die Methode `ComboBox1_SelectedIndexChanged()`

Hinweis:

Wie die Methode genau heißt, hängt vom Namen des Steuerelements ab. Wenn Sie einen eigenen Namen für das Kombinationsfeld benutzen, passen Sie bitte auch den Namen in der Anweisung entsprechend an.

Legen Sie jetzt bitte die entsprechende Methode an. Testen Sie dann das Programm.

Kommen wir nun zum Listenfeld. Sie finden es in der Toolbox in der Gruppe **Allgemeine Steuerelemente** unter dem Eintrag **ListBox**¹⁷. Fügen Sie ein entsprechendes Steuer-element ein und positionieren Sie es rechts neben dem Kombinationsfeld. Verbreitern Sie nach dem Einfügen gegebenenfalls das Formular und achten Sie auch darauf, dass die Schaltfläche **Beenden** nicht vom Listenfeld überdeckt wird.

Das Formular mit dem eingefügten Listenfeld sollte ungefähr so aussehen:



Abb. 3.5: Das Formular mit dem Listenfeld

15. *SelectedValueChanged* bedeutet frei übersetzt so viel wie „ausgewählter Wert verändert“.

16. *SelectedItem* steht für „ausgewählter Eintrag“.

17. *ListBox* steht für „Listenfeld“.

Erfassen Sie dann über den Zeichenfolgen-Editor einige Einträge für das Listenfeld. Denken Sie dabei bitte daran, die einzelnen Einträge mit der Eingabetaste zu trennen.

Hinweis:

Anders als bei einem Kombinationsfeld werden die Einträge in einem Listenfeld auch direkt im Designer angezeigt. Bearbeiten können Sie die Einträge aber nur über den Zeichenfolgen-Editor.

Das Kopieren des markierten Eintrags aus dem Listenfeld erfolgt fast genauso wie das Kopieren aus dem Kombinationsfeld. Sie weisen dem Label `labelZiel` in der Methode für das Ereignis `SelectedValueChanged` des Listenfelds den Text des aktuellen Eintrags zu. Die entsprechende Anweisung sieht so aus:

```
labelZiel.Text = listBox1.SelectedItem.ToString();
```

Code 3.3: Die Anweisung für die Methode `ListBox1_SelectedValueChanged()`

3.3 Steuerelemente ein- und ausblenden

Nachdem wir jetzt aus verschiedenen Quellen Texte kopieren können, wollen wir dem Anwender noch die Möglichkeit geben, das Kombinationsfeld und das Listenfeld aus- und wieder einzublenden. Dazu nehmen wir ein Kontrollkästchen – ein Ankreuzfeld – in das Formular auf. Wenn dieses Kontrollkästchen markiert ist, sollen die beiden Felder angezeigt werden, andernfalls sollen sie ausgeblendet werden.

Fügen Sie im ersten Schritt ein Kontrollkästchen in das Formular ein. Sie finden es in der Toolbox in der Gruppe **Allgemeine Steuerelemente** unter dem Eintrag **CheckBox**¹⁸. Setzen Sie dann den Text, der neben dem Kontrollkästchen angezeigt wird, über die Eigenschaft **Text** auf `Einblenden`. Damit das Häkchen in dem Kontrollkästchen erscheint, setzen Sie außerdem noch die Eigenschaft **Checked**¹⁹ auf `True`. Über diese Eigenschaft können Sie auch sehr einfach feststellen, ob ein Kontrollkästchen markiert ist oder nicht.

Wenn der Anwender den Zustand eines Kontrollkästchens verändert, wird das Ereignis `CheckedChanged`²⁰ ausgelöst, mit dem wir die beiden Felder aus- beziehungsweise wieder einblenden können. Dazu setzen wir die Eigenschaft **Visible**²¹ auf `true` beziehungsweise `false`. Die dazu erforderlichen Anweisungen finden Sie im folgenden Code.

```
//ist das Kontrollkästchen markiert?  
//dann die Listen einblenden  
if (checkBox1.Checked == true)  
{  
    listBox1.Visible = true;  
    comboBox1.Visible = true;  
}  
//sonst ausblenden
```

18. Übersetzt bedeutet *check box* so viel wie „Häkchenfeld“.

19. *Checked* bedeutet übersetzt so viel wie „markiert“.

20. *CheckedChanged* lässt sich frei mit „Markierung geändert“ übersetzen.

21. Übersetzt bedeutet *visible* „sichtbar“.

```
else
{
    listBox1.Visible = false;
    comboBox1.Visible = false;
}
```

Code 3.4: Die Anweisungen zum Ein- und Ausblenden der Felder

Mit der Anweisung

```
if (checkBox1.Checked == true)
```

überprüfen wir, ob das Kontrollkästchen im Moment markiert ist. Wenn das der Fall ist, setzen wir die Eigenschaft `Visible` der beiden Listen auf `true`. Wenn das Kontrollkästchen nicht markiert ist – `checkBox1.Checked` also den Wert `false` hat –, blenden wir die beiden Listen aus. Dazu setzen wir die Eigenschaft `Visible` jeweils auf `false`.

Hinweis:

Sie könnten der Eigenschaft `Visible` auch direkt den Wert der Eigenschaft `Checked` des Kontrollkästchens zuweisen. Dadurch sparen Sie sich die komplette Abfrage. Der Code ist dann deutlich kompakter, aber nicht mehr ganz so einfach nachzuvollziehen.

Übernehmen Sie jetzt die Änderungen. Speichern Sie dann das Projekt und probieren Sie die Erweiterungen aus.

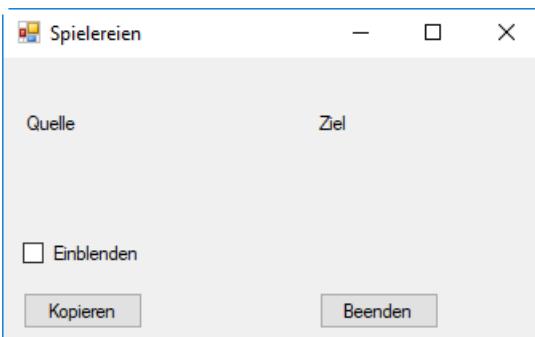


Abb. 3.6: Das Kontrollkästchen im Einsatz

Damit ist die Funktionalität des Formulars weitgehend komplett. Im nächsten Abschnitt werden wir noch für ein wenig Feinschliff sorgen.

3.4 Der letzte Schliff

Im ersten Schritt gleichen wir die Breite der Schaltflächen und des Kombinationsfelds beziehungsweise des Listenfelds an. Das geht am einfachsten, wenn Sie zuerst das Steuerelement markieren, an dessen Größe die anderen Elemente angepasst werden sollen. Markieren Sie dann alle weiteren gewünschten Elemente und klicken Sie abschließend auf das Symbol **Breite angleichen**  in der Symbolleiste **Layout**.

Probieren Sie das aus. Lassen Sie die Breite der Schaltfläche **Kopieren** an die Breite des Kombinationsfelds anpassen. Wiederholen Sie die Anpassung dann auch noch für das Listenfeld und die Schaltfläche **Beenden**.

Das Formular sollte danach ungefähr so aussehen:

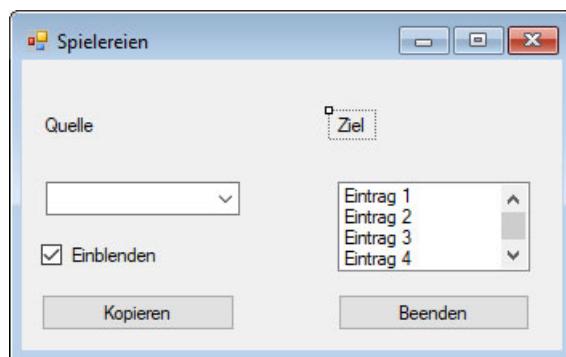


Abb. 3.7: Die angepassten Steuerelemente

Tipp:

Über die Symbole **Höhe angleichen**  und **Größe angleichen**  können Sie auch die Höhe von Steuerelementen beziehungsweise die Breite und die Höhe gleichzeitig angelichen.

Im zweiten Schritt sorgen wir nun noch dafür, dass im Kombinationsfeld und im Listenfeld beim Anzeigen des Formulars jeweils der erste Eintrag angezeigt beziehungsweise markiert wird. Denn bisher ist ja zum Beispiel das Kombinationsfeld direkt nach dem Start des Programms leer.

Für die Anzeige des ersten Eintrags setzen wir den Index in der Liste der Felder über die Eigenschaft **SelectedIndex**²² jeweils auf 0. Dieser Wert steht für den ersten Eintrag in der Liste.



Bitte beachten Sie:

In Listen- und Kombinationsfeldern beginnt die Nummerierung der Einträge mit dem Wert 0 und nicht mit dem Wert 1. Der Wert -1 steht in der Regel für eine leere Auswahl.

22. *SelectedIndex* lässt sich frei mit „ausgewählter Index“ oder „Index des Ausgewählten“ übersetzen.

Damit die Änderungen bei der Anzeige des Formulars automatisch erfolgen, nehmen wir sie im Ereignis **Load**²³ des Formulars vor. Dieses Ereignis tritt nach dem Erzeugen eines Formulars, aber noch vor der Anzeige ein. Sie finden das Ereignis im Eigenschaftenfenster bei den Ereignissen des Formulars in der Gruppe **Verhalten**.

Erstellen Sie jetzt eine Methode für das Ereignis **Load** des Formulars. Übernehmen Sie in diese Methode die folgenden Anweisungen:

```
//die ersten Einträge in den beiden Feldern auswählen  
listBox1.SelectedIndex = 0;  
comboBox1.SelectedIndex = 0;
```

Code 3.5: Die Anweisungen für die Methode Form1_Load()

Bevor Sie weiterlesen ...

Das Setzen der Einträge für die beiden Listen auf den Wert 0 beeinflusst auch direkt die beiden Labels. Überlegen Sie einmal, woran das liegen könnte.

Die Erklärung ist einfach. Sowohl für das Listenfeld als auch für das Kombinationsfeld haben wir eine Reaktion auf das Ereignis **SelectedValueChanged** programmiert. Dieses Ereignis tritt ein, sobald sich der Eintrag in dem Feld verändert – und zwar auch dann, wenn die Änderungen durch das Programm selbst vorgenommen werden. Da wir beim Ereignis **Load** für das Formular einen anderen Eintrag in den beiden Feldern auswählen, lösen wir damit automatisch auch das Ereignis **SelectedValueChanged** aus.

Wenn Sie die ursprünglichen Werte in den beiden Labels wieder anzeigen lassen möchten, müssten Sie im Ereignis **Load** nach dem Auswählen der Einträge in den Listen also zusätzlich noch die Eigenschaft **Text** der beiden Labels wieder neu setzen. Das sollte Sie jetzt nicht mehr vor große Schwierigkeiten stellen.

Damit beenden wir die Arbeit an dem Projekt. Experimentieren Sie aber ruhig noch selbst ein wenig mit den verschiedenen Steuerelementen. Sehen Sie sich noch weitere Eigenschaften und Ereignisse an. Testen Sie auch die verschiedenen Funktionen zum Ausrichten der Steuerelemente.

Im nächsten Kapitel werden wir uns an ein etwas größeres Projekt heranwagen – wir programmieren einen einfachen Taschenrechner.

Zusammenfassung

Wenn Sie mehrere identische Steuerelemente in einem Formular verwenden, sollten Sie sprechende Namen verwenden. Andernfalls kann es schwierig werden, die einzelnen Steuerelemente korrekt zuzuordnen.

Die Methoden für die Verarbeitung eines Ereignisses müssen Sie nicht in jedem Fall neu programmieren. Sie können auch bereits vorhandene Methoden verwenden.

Über Kombinationsfelder und Listenfelder können Sie einem Anwender feste Einträge in Listenform anbieten. Die Einträge in der Liste werden über den Zeichenfolgen-Editor erfasst.

23. *Load* bedeutet übersetzt „lade“.

Wenn der Anwender in einem Kombinationsfeld oder Listenfeld einen anderen Eintrag auswählt, tritt das Ereignis **SelectedValueChanged** ein.

Über Kontrollkästchen können Sie den Anwender Einstellungen ein- beziehungsweise ausschalten lassen. Beim Verändern des Zustands in einem Kontrollkästchen wird das Ereignis **CheckedChanged** ausgelöst.

Über die Eigenschaft **Visible** können Sie ein Steuerelement in einem Formular ein- beziehungsweise ausblenden.

Beim Laden eines Formulars wird das Ereignis **Load** ausgelöst. Es tritt nach dem Erzeugen, aber noch vor dem Anzeigen ein.

Aufgaben zur Selbstüberprüfung

- 3.1 Was geschieht, wenn Sie auf ein Steuerelement im Designer doppelklicken?

- 3.2 Was ist eine **CheckBox**?

- 3.3 Sie wollen im Quelltext auf die Eigenschaft `Text` eines Steuerelements `textLabel` zugreifen. Wie lautet der entsprechende Ausdruck?

- 3.4 Wie können Sie einem Ereignis eine bereits vorhandene Methode zuweisen?
Können Sie dabei beliebige Methoden verwenden?

- 3.5 Sie wollen den aktuellen Eintrag aus einem Kombinationsfeld `comboBoxAuswahl` in ein Label `textLabel` kopieren. Formulieren Sie bitte die entsprechende Anweisung.

- 3.6 Mit welcher Eigenschaft überprüfen Sie, ob ein Kontrollkästchen gerade markiert ist?

- 3.7 Über welche Eigenschaft eines Listen- beziehungsweise Kombinationsfelds können Sie selbst mit einer Anweisung einen Eintrag auswählen? Worauf müssen Sie dabei achten?

4 Ein einfacher Taschenrechner

In diesem Kapitel werden wir das bisher größte Projekt erstellen – einen einfachen Taschenrechner.

4.1 Vorüberlegungen und Vorbereitungen

Für unseren Taschenrechner benötigen wir folgende Steuerelemente:

- zwei Eingabefelder für die beiden Zahlen, die verarbeitet werden sollen,
- eine Gruppe mit Optionsfeldern für die Auswahl der Rechenoperation,
- zwei Labels für die Ausgabe des Ergebnisses sowie
- eine Schaltfläche zum Starten der Berechnung und eine Schaltfläche zum Beenden der Anwendung.

Beim Anklicken der Schaltfläche zur Berechnung soll das Programm zunächst die beiden Zahlen aus den Eingabefeldern beschaffen und sie dann anhand der ausgewählten Rechenoperation verarbeiten. Das Ergebnis der Rechenoperation soll anschließend in einem der beiden Labels ausgegeben werden. Das andere Label benutzen wir nur, um einen beschreibenden Text für das zweite Label anzuzeigen.

So viel zu den Vorüberlegungen. Beginnen wir jetzt mit den Vorbereitungen.

Legen Sie bitte ein neues Projekt für eine Windows Forms-Anwendung mit C# an. Wir verwenden in unserem Beispiel den Namen **Taschenrechner**.

Setzen Sie dann den Text in der Titelleiste des Formulars auf **Ein einfacher Taschenrechner**. Die Höhe des Formulars setzen Sie bitte auf 200 Pixel und die Breite auf 400 Pixel.

Fügen Sie anschließend unten im Formular zwei Schaltflächen ein. Die Schaltfläche links soll den Text **Berechnen** erhalten und die Schaltfläche rechts den Text **Beenden**. Sorgen Sie dann dafür, dass beim Anklicken der Schaltfläche rechts die Anwendung beendet wird.

Als letzten Schritt fügen Sie bitte noch zwei Labels ein. Das erste Label soll den Text **Ergebnis:** enthalten und das zweite Label soll den Wert 0 anzeigen. Das zweite Label werden wir später für die Ausgabe des Ergebnisses benutzen. Positionieren Sie die Labels nebeneinander links im Formular oberhalb der Schaltfläche **Berechnen**.

Hinweis:

Sorgen Sie bitte durch sprechende Namen dafür, dass Sie die beiden Schaltflächen und auch die Labels ohne Schwierigkeiten auseinanderhalten können. Wir verwenden in unserem Beispiel die Namen `buttonBerechnen` und `buttonBeenden` für die beiden Schaltflächen sowie die Namen `labelBeschreibung` und `labelAnzeige` für die beiden Labels. Das Label `labelBeschreibung` enthält dabei den Text **Ergebnis:** und das Label `labelAnzeige` den Wert 0. Wenn Sie andere Namen verwenden, denken Sie bitte daran, dass Sie die folgenden Quelltexte entsprechend anpassen müssen.

Der Rohbau des Formulars sollte ungefähr so aussehen:

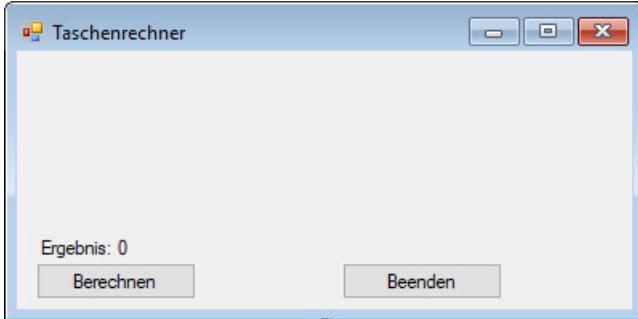


Abb. 4.1: Das Formular für den Taschenrechner im „Rohbau“

4.2 Die Eingabefelder

Im nächsten Schritt fügen wir jetzt die beiden Eingabefelder für die Zahlen ein. Dazu verwenden wir einfache Eingabefelder. Das entsprechende Steuerelement finden Sie in der Toolbox unten in der Gruppe **Allgemeine Steuerelemente** unter **TextBox**²⁴.

Fügen Sie jetzt zwei Steuerelemente vom Typ **TextBox** ein. Positionieren Sie die Steuerelemente untereinander links oben im Formular. Setzen Sie anschließend die Eigenschaft **Text** der beiden Steuerelemente jeweils auf 0 und vergeben Sie „sprechende“ Namen an die beiden Textboxen. Wir verwenden in unserem Beispiel die Namen `textBoxZahl1` und `textBoxZahl2`.

Das Formular sollte nun so aussehen:

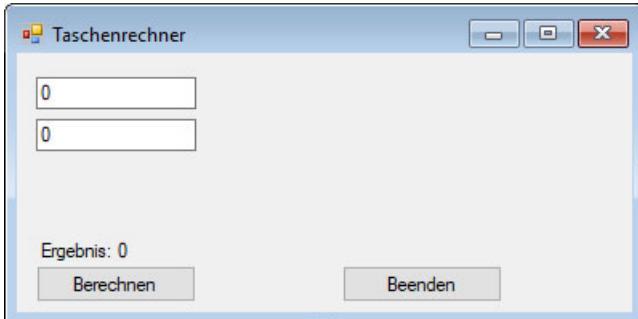


Abb. 4.2: Die beiden Eingabefelder für die Zahlen

Tipp:

Wenn Sie möchten, können Sie vor die beiden Eingabefelder auch noch beschreibende Labels stellen – zum Beispiel mit dem Text **Zahl 1** und dem Text **Zahl 2**. Das sollten Sie aber ohne Schwierigkeiten selbst bewerkstelligen können.

In der Standardeinstellung werden Werte in einem Eingabefeld linksbündig ausgerichtet. Sie können über die Eigenschaft **TextAlign**²⁵ aber auch eine andere Ausrichtung wählen.

24. *TextBox* steht für „Textfeld“.

25. *Align* bedeutet übersetzt „ausrichten“.

4.3 Die Optionsfelder

Für den Entwurf des Formulars fehlen jetzt nur noch die Optionsfelder zur Auswahl der Rechenoperation.

Da wir in unserem Programm mehrere verschiedene Alternativen über die Optionsfelder anbieten wollen, fassen wir die Felder in einer Gruppe zusammen. Diese Gruppe bildet dann eine Art **Container** für die einzelnen Optionsfelder.



Zur Auffrischung:

Nach dem Windows-Standard werden **Optionsfelder** für Entweder-Oder-Alternativen eingesetzt. Von mehreren Optionen in einer Gruppe kann also immer nur eine aktiviert sein.

Bei **Kontrollkästchen**, die in einer Gruppe zusammengefasst werden, können auch mehrere Kästchen gleichzeitig markiert sein.

Fügen Sie nun bitte die Gruppe für die Optionsfelder ein. Benutzen Sie dazu das Steuerelement **GroupBox**²⁶ in der Gruppe **Container** der Toolbox. Positionieren Sie die Gruppe rechts oben in dem Formular. Benutzen Sie zum Verschieben im Formular dabei das Symbol , das oben links auf dem Rahmen der Gruppe angezeigt wird.

Ändern Sie dann den Text, der oben in der Gruppe angezeigt wird, in **Rechenoperation**. Setzen Sie dazu die Eigenschaft **Text** der **GroupBox** auf den entsprechenden Wert.

Jetzt kommen die einzelnen Optionsfelder an die Reihe. Das entsprechende Steuerelement finden Sie unter **RadioButton**²⁷ in der Gruppe **Allgemeine Steuerelemente** der Toolbox.

Fügen Sie insgesamt vier Optionsfelder in die Gruppe ein. Achten Sie dabei bitte sorgfältig darauf, dass Sie die Elemente tatsächlich in der Gruppe ablegen. Ordnen Sie die Elemente anschließend untereinander an. Falls der Platz für das vierte Optionsfeld nicht mehr ausreicht, vergrößern Sie die Gruppe etwas nach unten.

Vergeben Sie dann über die Eigenschaft **Text** der einzelnen Optionsfelder Bezeichnungen für die Rechenoperationen Addition, Subtraktion, Division und Multiplikation. Ändern Sie die Namen der Steuerelemente außerdem so, dass Sie die einzelnen Felder problemlos zuordnen können – zum Beispiel in `radioButtonAddition`, `radioButtonSubtraktion` und so weiter.

Setzen Sie abschließend die Eigenschaft **Checked** für das erste Optionsfeld in der Gruppe auf `True`. Dadurch wird der erste Eintrag automatisch markiert.

Hinweis:

Sie können immer nur für ein Optionsfeld in einer Gruppe die Eigenschaft **Checked** auf `True` setzen. Bei allen anderen Optionsfeldern in der Gruppe wird die Eigenschaft **Checked** dann automatisch auf `False` gesetzt. Diese automatischen Änderungen erfolgen auch beim Ausführen des Programms, wenn der Anwender ein an-

26. *GroupBox* bedeutet wörtlich übersetzt so viel wie „Gruppenfeld“.

27. Wörtlich übersetzt bedeutet *radio button* „Funkschalter“ oder „Radioschalter“.

deres Optionsfeld markiert. Das funktioniert allerdings nur dann, wenn sich die Optionsfelder gemeinsam in einer Gruppe befinden beziehungsweise gemeinsam im Formular abgelegt werden.

Merken Sie sich:

Optionsfelder werden in der Regel in zwei Schritten erstellt. Zuerst legen Sie über eine GroupBox einen Container an. Danach legen Sie in diesem Container die einzelnen Optionsfelder ab.



Damit ist der Entwurf des Formulars fertig. Es sollte nun ungefähr so aussehen:

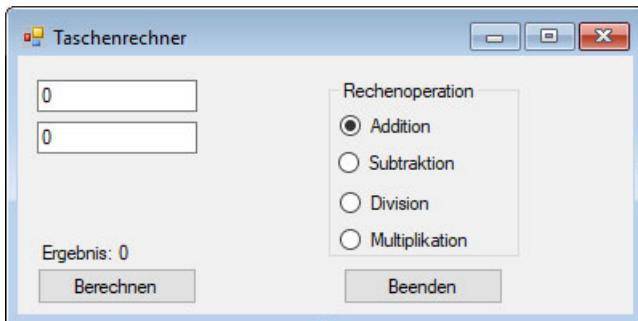


Abb. 4.3: Das fertige Formular für den Taschenrechner

Speichern Sie alle Änderungen und führen Sie einen ersten Test durch. Probieren Sie dabei auch das Verändern der Rechenoperation über die Optionsfelder und die Eingabe in die Textfelder aus.

4.4 Die Berechnungen

Jetzt müssen wir noch die Anweisungen für die Berechnungen programmieren. Da diese Anweisungen etwas komplexer sind, wollen wir uns erst einmal überlegen, was überhaupt beim Anklicken der Schaltfläche **Berechnen** geschehen soll.

Zuerst müssen die Werte in den beiden Eingabefeldern ausgelesen und in zwei lokalen Variablen zwischengespeichert werden. Dazu weisen wir den aktuellen Wert der Eigenschaft **Text** für die beiden Felder jeweils einer Variable zu. Da die Daten in Eingabefeldern aber immer als Zeichenkette geliefert werden, müssen wir vor der Zuweisung noch dafür sorgen, dass eine Konvertierung in eine Zahl erfolgt. Die dazu erforderlichen Techniken kennen Sie ja bereits von den Konsolenprogrammen.

Die Eigenschaft **Text** eines Eingabefelds liefert Ihnen immer eine Zeichenkette – auch dann, wenn nur numerische Werte in dem Feld stehen.

**Hinweis:**

Sie können auf das Zwischenspeichern auch verzichten und die Werte direkt aus den Eingabefeldern auslesen. Dann werden die Rechenausdrücke allerdings recht sperrig.

Danach führen wir mit den beiden Variablen – abhängig von der ausgewählten Option – eine Rechenoperation durch. Welche Rechenoperation ausgewählt wurde, ermitteln wir über verschiedene `if`-Abfragen, die nacheinander den Status der vier Optionsfelder über die Eigenschaft **Checked** abfragen.

Um Probleme bei einer Division durch null zu vermeiden, prüfen wir außerdem bei der Rechenoperation Division, ob die zweite eingegebene Zahl eine 0 ist. Wenn das der Fall ist, soll keine Division durchgeführt werden, sondern der Text „Nicht definiert!“ erscheinen.

Abschließend übertragen wir das Ergebnis in das Label für die Ausgabe. Dazu formen wir die Zahl in eine Zeichenkette um und weisen diese Zeichenkette der Eigenschaft **Text** des Labels zu.

Legen Sie jetzt bitte eine Methode für das Ereignis **Click** der Schaltfläche **Berechnen** an und übernehmen Sie die Anweisungen aus dem folgenden Code. Was es genau mit den Anweisungen auf sich hat, erklären wir Ihnen im Anschluss.

```
float zahl1, zahl2, ergebnis = 0;
bool divDurchNull = false;
//die beiden Zahlen einlesen und konvertieren
zahl1 = Convert.ToSingle(textBoxZahl1.Text);
zahl2 = Convert.ToSingle(textBoxZahl2.Text);
//die Rechenoperation ermitteln und ausführen
if (radioButtonAddition.Checked == true)
    ergebnis = zahl1 + zahl2;
if (radioButtonSubtraktion.Checked == true)
    ergebnis = zahl1 - zahl2;
if (radioButtonMultiplikation.Checked == true)
    ergebnis = zahl1 * zahl2;
if (radioButtonDivision.Checked == true)
{
    //wird eine Division durch Null versucht?
    if (zahl2 == 0)
        divDurchNull = true;
    else
        ergebnis = zahl1 / zahl2;
}
//wurde durch Null dividiert?
if (divDurchNull == true)
    labelAnzeige.Text = "Nicht definiert!";
else
    labelAnzeige.Text = Convert.ToString(ergebnis);
```

Code 4.1: Die Methode zum Berechnen

Zunächst einmal vereinbaren wir vier lokale Variablen: drei `float`-Variablen für die Eingaben und das Ergebnis sowie ein Variable `divDurchNull` vom Typ `bool`. Diese Variable verwenden wir, um die Ausgabe zu steuern, falls eine Division durch 0 durchgeführt wurde. Damit wir von eindeutigen Zuständen ausgehen können, setzen wir den Wert der Variablen `ergebnis` auf 0 und den Wert der Variablen `divDurchNull` auf `false`.

Anschließend lesen wir die Werte der beiden Eingabefelder aus, konvertieren sie in float- beziehungsweise Single-Typen und weisen die Werte dann den Variablen zahl1 und zahl2 zu.

Auch die folgenden if-Anweisungen sollten Sie nicht vor große Probleme stellen. Wir überprüfen hier jeweils, ob die Eigenschaft Checked eines Optionsfelds true ist, und lassen dann die jeweilige Rechenoperation durchführen. Bei der Division kontrollieren wir vorher noch, ob eine Division durch 0 erfolgen würde. Wenn das der Fall ist, setzen wir den Wert der Variablen divDurchNull auf true. Andernfalls wird die Division durchgeführt.

Hinweis:

Sie könnten die Abfrage der Optionsfelder auch als stark geschachtelte if-else-Konstruktion aufbauen. Denn eigentlich müssen ja keine weiteren Abfragen erfolgen, sobald eine markierte Option gefunden wurde. Da der Quelltext dadurch aber sehr unübersichtlich wird, haben wir bewusst darauf verzichtet.

Am Ende des Programms schließlich überprüfen wir den Wert der Variablen divDurchNull. Wenn er false ist, setzen wir die Eigenschaft Text des Steuerelements labelAnzeige auf den aktuellen Wert der Variablen ergebnis. Damit alles seine Ordnung hat, formen wir den Wert vorher noch in eine Zeichenkette um.

Ist der Wert der Variablen divDurchNull dagegen true, setzen wir den Wert der Eigenschaft Text für das Label auf „Nicht definiert!“.

Speichern Sie jetzt die Änderungen und testen Sie das Programm dann. Bitte beachten Sie dabei aber, dass der Taschenrechner bei der Eingabe von ungültigen Daten – zum Beispiel einem Buchstaben – abstürzt. Anders als bei den Konsolenprogrammen führt der Absturz bei einer Windows Forms-Anwendung aber nicht sofort zum Ende des Programms. Sie können in einem Dialog nämlich auswählen, ob Sie den Fehler ignorieren wollen und mit dem Programm weiterarbeiten möchten.

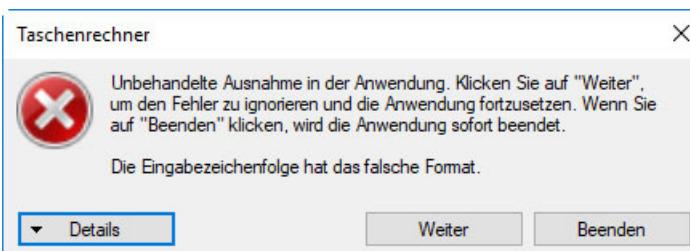


Abb. 4.4: Der Dialog bei einer ungültigen Eingabe

Die Ausnahmebehandlung von Windows Forms-Anwendungen ist also ein wenig ausgeföhler als die Ausnahmebehandlung bei Konsolenanwendungen.

Tipp:

Wenn Sie die Standardmeldung von Windows stört, können Sie über `try` und `catch` auch eine eigene Ausnahmebehandlung festlegen. Das erfolgt ähnlich wie bei unserer sehr einfachen Ausnahmebehandlung für die Konsolenanwendungen und könnte im einfachsten Fall für den Taschenrechner zum Beispiel so aussehen:

```
try
{
    zahl1 = Convert.ToSingle(textBoxZahl1.Text);
}
catch (FormatException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Ihre Eingabe " + textBoxZahl1.
    Text + " war nicht gültig. ", "Fehler");
}
```

Falls die Konvertierung nicht gelingt, wird über `MessageBox.Show()` ein einfacher Dialog mit einer **OK**-Schaltfläche erzeugt. Das erste Argument gibt dabei den Text im Dialog an und das zweite Argument den Text in der Titelleiste des Dialogs.

Versuchen Sie einmal selbst, diese Ausnahmebehandlung einzubauen und auch noch ein wenig zu verfeinern. So müssen ja zum Beispiel die Rechenoperationen nur dann durchgeführt werden, wenn das Konvertieren beider Zahlen gelungen ist. Auch das Ergebnis darf eigentlich nur dann erscheinen, wenn mit gültigen Werten gerechnet wurde.

Wir werden uns aber später auch noch einmal intensiv mit dem Thema Ausnahmebehandlung beschäftigen.

4.5 Der Feinschliff

Im letzten Schritt geben wir dem Taschenrechner jetzt noch ein wenig Feinschliff. Wir passen die Größe der Steuerelemente an, sorgen dafür, dass das Fenster nicht vergrößert werden kann, und legen die Aktivierreihenfolge fest.

Beginnen wir mit dem Anpassen der Steuerelemente.

Lassen Sie die Schaltfläche **Berechnen** in der gleichen Breite wie die beiden Eingabefelder darstellen. Das geht am schnellsten, wenn Sie zuerst eines der Eingabefelder markieren und anschließend mit gedrückter Taste auf die Schaltfläche klicken. Klicken Sie dann auf das Symbol **Breite angleichen** in der Symbolleiste **Layout**. Setzen Sie anschließend die zweite Schaltfläche auf die Breite der ersten. Falls die zweite Schaltfläche dabei über den Rand des Formulars hinaus verbreitert wird, müssen Sie sie danach wieder per Hand neu positionieren.

Hinweis:

Achten Sie vor dem Verschieben der zweiten Schaltfläche darauf, dass die erste Schaltfläche nicht mehr markiert sein darf. Sonst verschieben Sie nämlich beide Schaltflächen.

Beim Steuerelement **Label** wird die Größe in der Standardeinstellung automatisch an den Inhalt angepasst. Daher haben manuelle Änderungen der Größe im Designer keine Auswirkungen. Wenn Sie die Größe eines Labels selbst setzen möchten, müssen Sie zunächst die Eigenschaft **AutoSize**²⁸ in der Gruppe **Layout** auf **False** setzen. Dann werden allerdings Texte, die nicht in das Label passen, schlicht und einfach abgeschnitten.

Passen Sie nun noch die Breite der Gruppe mit den Optionsfeldern so an, dass sie möglichst exakt für die Inhalte passt. Das geht am einfachsten, wenn Sie an einem der Anfasser am linken oder rechten Rand der Gruppe ziehen. Richten Sie dann die zweite Schaltfläche zentriert unterhalb der Gruppe aus. Markieren Sie dazu zuerst die Gruppe und danach die Schaltfläche. Klicken Sie anschließend auf die Schaltfläche **Zentriert**  in der Symbolleiste **Layout** des Editors.

Abschließend können Sie jetzt noch das Formular ein wenig verkleinern. Nach diesen kosmetischen Korrekturen sollte das Formular nun ungefähr so aussehen:

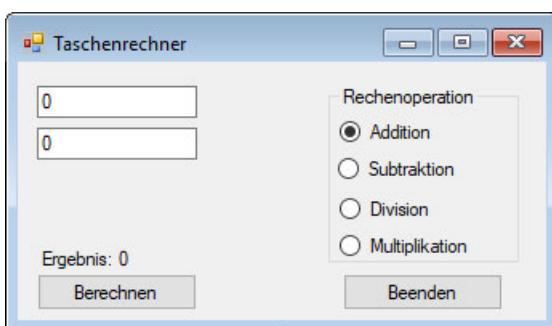


Abb. 4.5: Das Formular nach den kosmetischen Korrekturen

Allerdings sind diese ganzen Verbesserungen umsonst, wenn der Anwender das Fenster unseres Taschenrechners vergrößert oder maximiert. Dann verlieren sich nämlich die Steuerelemente oben links in der Ecke und die restliche Fläche des Formulars ist schlicht und einfach leer.

28. *AutoSize* lässt sich mit „automatische Größe“ übersetzen.

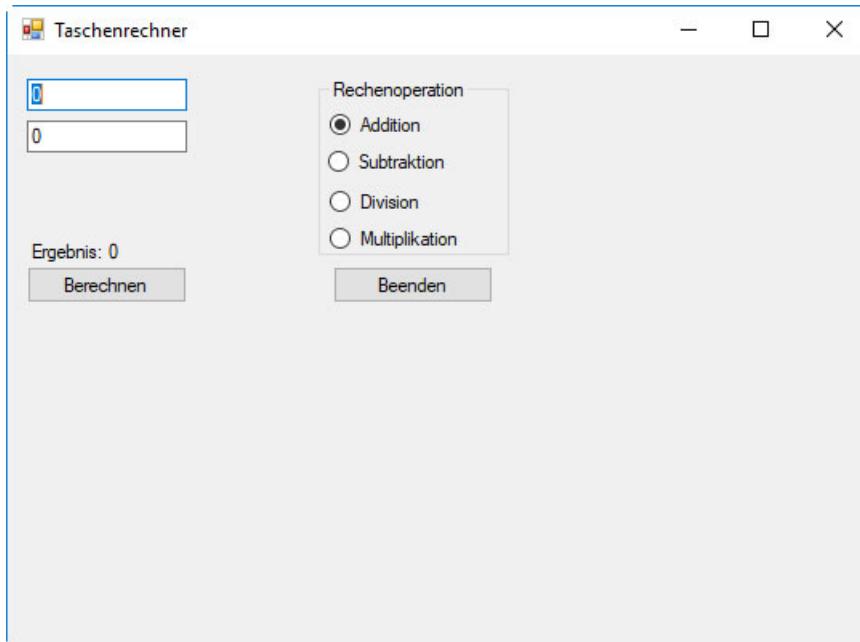


Abb. 4.6: Das vergrößerte Fenster des Taschenrechners

Wir müssen also noch sicherstellen, dass das Fenster nicht vergrößert werden kann. Dazu deaktivieren wir zunächst einmal die Maximieren-Funktion. Das geht ganz einfach, indem Sie die Eigenschaft **MaximizeBox**²⁹ in der Gruppe **Fensterstil** auf `False` setzen. Dadurch werden sowohl das Maximieren-Symbol rechts in der Titelleiste als auch die Funktion **Maximieren** im Systemmenü der Anwendung abgeblendet.

Hinweis:

Achten Sie bitte darauf, dass das Formular markiert ist. Andernfalls werden Sie die Eigenschaft **MaximizeBox** vergeblich suchen. Wenn Sie nicht sicher sind, klicken Sie einmal mit der Maus auf die Titelleiste des Formulars im Designer.

Damit auch Größenänderungen über den Rand des Fensters nicht mehr möglich sind, müssen Sie noch die Eigenschaft **FormBorderStyle**³⁰ in der Gruppe **Darstellung** anpassen. Sie können hier einen Eintrag benutzen, der mit **Fixed**³¹ beginnt – zum Beispiel **Fixed3D** für einen dreidimensionalen Rahmen.

Nehmen Sie jetzt bitte die entsprechenden Änderungen vor und testen Sie den Taschenrechner noch einmal. Das Fenster sollte sich nun nur noch minimieren und wiederherstellen lassen.

Nachdem das Formular jetzt optisch einigermaßen gelungen ist, wollen wir uns um die **Aktivierreihenfolge** kümmern. Diese Reihenfolge legt fest, wie der Fokus zwischen den Steuerelementen verschoben wird.

29. *MaximizeBox* lässt sich mit „Maximieren-Feld“ übersetzen.

30. *FormBorderStyle* lässt sich mit „Stil des Formularrahmens“ übersetzen.

31. *Fixed* bedeutet übersetzt so viel wie „fixiert“.

Der **Fokus** bestimmt, welches Steuerelement aktuell auf Tastatureingaben reagiert. Wenn der Fokus zum Beispiel auf einer Schaltfläche liegt, können Sie die Funktion der Schaltfläche auch mit der Leertaste oder der Eingabetaste starten. Wenn eine Gruppe mit Optionsfeldern den Fokus hat, können Sie die einzelnen Optionen auch mit den Cursortasten verändern.



In der Standardeinstellung wird der Fokus direkt nach dem Start des Programms immer auf das Steuerelement gesetzt, das Sie zuerst in das Formular eingefügt haben. Sollte es sich bei dem ersten Element um ein Steuerelement handeln, das den Fokus nicht erhalten kann, wird der Fokus automatisch auf das nächste mögliche Element gesetzt.

Mit der Taste beziehungsweise der Tastenkombination + können Sie dann den Fokus vorwärts beziehungsweise rückwärts über alle Elemente verschieben, die den Fokus erhalten können.

Probieren Sie das Verschieben des Fokus jetzt einmal in unserem Taschenrechner aus. Wenn Sie die Steuerelemente in der gleichen Reihenfolge eingefügt haben wie im Studienheft, steht der Fokus nach dem Start auf der Schaltfläche **Berechnen**. Das erkennen Sie an dem blauen Rahmen in der Schaltfläche. Mit der Taste können Sie den Fokus dann der Reihe nach auf die Schaltfläche **Beenden**, auf die Eingabefelder und auf die Gruppe mit den Optionsfeldern verschieben. Da die Labels den Fokus nicht erhalten können, werden sie beim Drücken der Taste übersprungen.

Jetzt könnten Sie natürlich sagen: „Warum soll ich mich überhaupt um den Fokus kümmern? Der Anwender kann ja den Fokus beliebig setzen. Er muss eben nur so oft die Taste drücken, bis das gewünschte Steuerelement markiert ist.“ Trotzdem sollten Sie die Reihenfolge im Auge behalten: Viele Anwender, die mit der Tastatur arbeiten, gehen aus Gewohnheit von einer bestimmten Reihenfolge aus.

Normalerweise liegt der Fokus direkt nach dem Aufruf eines Formulars immer auf dem ersten Steuerelement oben links, das überhaupt den Fokus erhalten kann. Bei jedem Drücken der Taste wird der Fokus dann ein Element nach rechts verschoben. Das letzte Element in der Reihenfolge ist das Element, das sich unten rechts im Formular befindet.

Die Standardreihenfolge für das Verschieben des Fokus geht von oben links nach unten rechts.



Wenn Sie sich nicht an diese übliche Reihenfolge halten und ein Anwender Ihr Programm vor allem mit der Tastatur bedienen möchte, kann der Komfort erheblich leiden. Anstatt den Fokus ohne Hinzusehen verschieben zu können, muss der Anwender jedes Mal nachsehen, wo sich der Fokus denn nun gerade befindet. Sie sollten deshalb darauf achten, dass Sie in Ihren Formularen die übliche Reihenfolge einhalten.

Das heißt nun aber nicht, dass Sie die Steuerelemente in dem Formular in jedem Fall von vornherein von links oben nach rechts unten einfügen müssen. Sie können die Aktivierreihenfolge jederzeit nachträglich verändern. Rufen Sie dazu die Funktion **Aktivierreihenfolge** im Menü **Ansicht** auf.

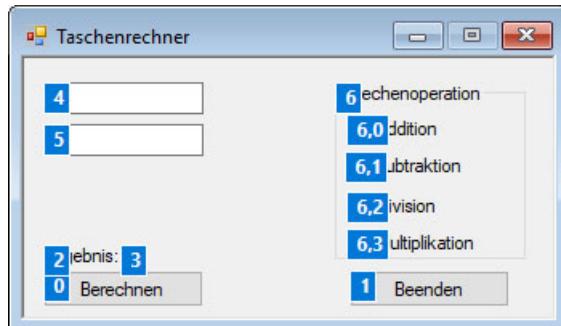


Abb. 4.7: Einstellen der Aktivierreihenfolge

Visual Studio zeigt Ihnen dann die aktuelle Aktivierreihenfolge durch die Zahlen in den Rechtecken an. Dabei werden auch die Steuerelemente mit nummeriert, die den Fokus eigentlich gar nicht erhalten können.

Hinweis:

Die Zahlen mit einem Komma – wie zum Beispiel 6,1 und 6,2 – geben die Reihenfolge von untergeordneten Steuerelementen an. In unserem Beispiel befindet sich die Gruppe mit den Optionsfeldern in der Aktivierreihenfolge zum Beispiel an der Position 6. Die einzelnen Optionsfelder werden dann von oben nach unten aktiviert.

Sie müssen das Formular im Designer anzeigen lassen, damit Sie die Aktivierreihenfolge verändern können. Wenn der Code angezeigt wird, fehlt die Funktion im Menü **Ansicht**.

Um die Reihenfolge zu ändern, klicken Sie die Steuerelemente einfach hintereinander in der gewünschten Reihenfolge an. Alternativ können Sie auch so oft auf ein Steuerelement klicken, bis es die gewünschte Position in der Aktivierreihenfolge hat. Visual Studio ändert dabei automatisch die Zahlen an den Elementen.

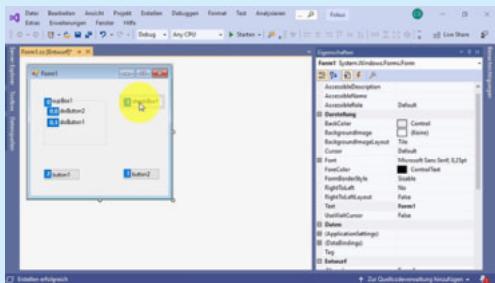
Anschließend drücken Sie die Taste **Esc**, um die neue Reihenfolge zu übernehmen.



Bitte beachten Sie:

Sie können auch zwei Elementen die gleiche Position in der Aktivierreihenfolge zuweisen. Da aber nicht zwei Elemente gleichzeitig den Fokus erhalten können, wird diese ungültige Zuweisung beim Ausführen des Programms automatisch korrigiert. In der Darstellung der Aktivierreihenfolge im Designer werden aber nach wie vor dieselben Zahlen angezeigt.

Probieren Sie das Ändern der Aktivierreihenfolge jetzt einmal selbst aus. Sorgen Sie dafür, dass der Fokus zuerst auf den beiden Eingabefeldern steht. Danach sollen dann die Gruppe mit den Optionsfeldern und die beiden Schaltflächen den Fokus erhalten.



Eine praktische Demonstration zum Ändern der Aktivierreihenfolge finden Sie auch in diesem Video.



www.dfz.media/zyrrwa

Video 4.1: Ändern der Aktivierreihenfolge

Abschließend noch zwei Hinweise:

Auch wenn Ihnen die Aktivierreihenfolge in einem Formular im Moment vielleicht noch recht unwichtig erscheint, sollten Sie trotzdem bei Ihren Formularen darauf achten, dass Sie die Standardreihenfolge einhalten. Viele Anwender sind es einfach so gewohnt und verlassen sich darauf, dass die Aktivierreihenfolge von links oben nach rechts unten läuft. Ein Fokus, der quer und quer durch ein Formular springt, kann für viel Verdruss sorgen und einen schlechten Eindruck hinterlassen.

Über die Aktivierreihenfolge können Sie auch gezielt steuern, welches Steuerelement unmittelbar nach der Anzeige des Formulars den Fokus erhält. Das ist zum Beispiel dann interessant, wenn Sie den Fokus gezielt auf eine bestimmte Schaltfläche setzen möchten. Die Funktion dieser Schaltfläche wird dann bei der Tastaturbedienung sozusagen zur Standardfunktion des Formulars.

Sie können aber auch über die Eigenschaften **AcceptButton** und **CancelButton**³² in der Gruppe **Sonstiges** des Formulars Standardschaltflächen für die Eingabetaste und die Taste **Esc** festlegen. Dazu wählen Sie jeweils die gewünschte Schaltfläche in der Liste für die Eigenschaft aus. Die Aktion der Schaltfläche für die Eigenschaft **AcceptButton** wird dann beim Drücken der Eingabetaste ausgelöst und die Aktion der Schaltfläche **CancelButton** beim Drücken der Taste **Esc**. Bei der Eigenschaft **CancelButton** spielt es dabei keine Rolle, auf welchem Steuerelement sich der Fokus aktuell befindet. Die Aktion für die Eigenschaft **AcceptButton** wird nur dann ausgelöst, wenn sich der Fokus nicht auf einem Steuerelement befindet, das selbst auf die Eingabetaste reagiert – zum Beispiel auf einer Schaltfläche. Welches Steuerelement aktuell auf die Eingabetaste reagiert, erkennen Sie in der laufenden Anwendung an dem Rahmen.

So viel zu unserem einfachen Taschenrechner.

32. *AcceptButton* lässt sich mit „Annehmen-Schaltfläche“ übersetzen und *CancelButton* mit „Abbrechen-Schaltfläche“.

Zusammenfassung

Für die Eingabe von beliebigen Werten können Sie das Steuerelement **TextBox** verwenden.

Optionsfelder werden normalerweise in einem Container abgelegt – zum Beispiel einer **GroupBox**. Die eigentlichen Optionsfelder werden über das Steuerelement **RadioButton** angelegt.

Über die Eigenschaft **Checked** eines Optionsfelds können Sie überprüfen, ob die Option aktuell ausgewählt ist.

Die Ausnahmebehandlung für Windows Forms-Anwendungen führt in vielen Fällen nicht direkt zum Ende des Programms, sondern erlaubt auch ein Ignorieren des Fehlers.

Über die Eigenschaft **FormBorderStyle** legen Sie den Rahmen des Formulars fest. Sie können hier auch Rahmen auswählen, die eine Größenänderung verhindern.

Die Aktivierreihenfolge legt fest, in welcher Reihenfolge die Steuerelemente in einem Formular den Fokus erhalten. Sie sollten dabei die Standardreihenfolge von oben links nach unten rechts einhalten.

Aufgaben zur Selbstüberprüfung

- 4.1 In welchem Format werden die Daten in einer **TextBox** immer geliefert?

- 4.2 Über welche Eigenschaft legen Sie die Textausrichtung in einer **TextBox** fest?

- 4.3 Sie haben in einem Formular mehrere Optionsfelder in einer **GroupBox** abgelegt und außerdem weitere Optionsfelder ohne Container direkt im Formular eingefügt. Beeinflusst eine Auswahl der Optionsfelder in der **GroupBox** die Auswahl der Optionsfelder, die direkt im Formular liegen? Probieren Sie das Verhalten einfach mit Visual Studio aus.

- 4.4 Sie haben eine **GroupBox** mit vier Optionsfeldern erstellt. Wie viele `if`-Abfragen benötigen Sie, um die aktuell ausgewählte Option zu ermitteln? Warum?

- 4.5 Sie wollen das Maximieren-Symbol in einem Formular deaktivieren. Welche Eigenschaft müssen Sie dazu auf welchen Wert setzen?

Schlussbetrachtung

Herzlichen Glückwunsch! Sie haben Ihre ersten Programme mit einer grafischen Oberfläche erstellt. Wie Sie gesehen haben, ist es mithilfe des Designers von Visual Studio gar nicht so schwer, schnell zu brauchbaren Ergebnissen zu kommen. Durch die vielen vorgefertigten Steuerelemente mussten Sie selbst nur sehr wenig Quelltext schreiben und konnten dennoch einen einfachen Taschenrechner erstellen.

Noch ein Tipp: Lassen Sie sich von der Vielzahl der Eigenschaften im Eigenschaftenfenster nicht verwirren oder gar entmutigen. Zahlreiche Einstellungen, die jetzt vielleicht noch geheimnisvoll wirken, werden sehr schnell klar, wenn Sie sie einmal eingesetzt haben. Wenn Sie nicht sicher sind, was sich hinter einer Eigenschaft verbirgt, versuchen Sie, den englischen Namen zu übersetzen, oder sehen Sie in der Hilfe nach.

Legen Sie sich Sicherungskopien der Programme an, die Sie in diesem Studienheft erstellt haben, und spielen Sie dann mit den verschiedenen Eigenschaften ein wenig herum. Probieren Sie einfach aus, welche Wirkung eine bestimmte Einstellung hat.

Sie werden sehen: Mit ein wenig Übung werden Sie schon nach kurzer Zeit souverän mit den nicht immer leicht verständlichen Abkürzungen der Eigenschaften umgehen können.

Versuchen Sie auch einmal, die Beispielprogramme ein wenig zu erweitern. So könnten Sie den Taschenrechner ja zum Beispiel so ausbauen, dass auch mehrere Zahlen verarbeitet werden können.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Es gibt fünf grundsätzliche Möglichkeiten:
 - das Programmieren über die Windows API,
 - den Einsatz der Microsoft Foundation Classes,
 - den Einsatz der Windows Forms,
 - den Einsatz der Windows Presentation Foundation und
 - das Erstellen von Universal Windows Platform Apps.

Für die richtige Lösung reicht es aus, wenn Sie drei der fünf Möglichkeiten genannt haben.

- 1.2 Zunächst rufen Sie die Funktion zum Anlegen eines neuen Projekts auf. Im Fenster **Neues Projekt erstellen** markieren Sie dann den Eintrag **Windows Forms-App (.NET Framework)** für die Programmiersprache C# in der Liste rechts und klicken auf **Weiter**. Danach legen Sie den Namen für das Projekt fest und klicken auf die Schaltfläche **Erstellen**.
- 1.3 Das Formular **Form1** ist das Hauptfenster der Anwendung.

Kapitel 2

- 2.1 Die Eigenschaft, die den Text in der Titelleiste eines Formulars festlegt, heißt **Text**.
- 2.2 Das Symbol weist darauf hin, dass es für die Eigenschaft weitere Untereigenschaften gibt.
- 2.3 Im ersten Schritt klicken Sie im Eigenschaftenfenster auf das Symbol **Ereignisse**  . Bei den Ereignissen doppelklicken Sie entweder in das Feld hinter dem Ereignis **Click** oder in das Feld hinter dem Ereignis **MouseClick**. Visual Studio legt dann automatisch eine Methode für die Ereignisverarbeitung an, in der Sie die erforderlichen Anweisungen ergänzen können.
- 2.4 Die Methoden werden in der Datei **Form1.cs** erstellt.

Kapitel 3

- 3.1 Beim Doppelklick auf ein Steuerelement im Designer wird die Methode für das Standardereignis des Steuerelements angelegt beziehungsweise angezeigt.
- 3.2 Eine CheckBox ist ein Steuerelement für ein Kontrollkästchen.
- 3.3 Um in einem Quelltext auf die Eigenschaft `Text` eines Steuerelements `textLabel` zuzugreifen, verwenden Sie den Ausdruck `textLabel.Text`.

- 3.4 Um einem Ereignis eine bereits vorhandene Methode zuzuweisen, öffnen Sie das Kombinationsfeld für das Ereignis und wählen dann die gewünschte Methode aus. In der Liste werden allerdings nur solche Ereignisse angezeigt, die zum Ereignis und zum Steuerelement passen. Sie können einem Ereignis also keine beliebige Methode zuweisen.

- 3.5 Die Anweisung zum Kopieren des aktuellen Eintrags aus einem Kombinationsfeld in ein Label kann zum Beispiel so aussehen:

```
textLabel.Text = comboBoxAuswahl.SelectedItem.ToString();
```

- 3.6 Den aktuellen Zustand eines Kontrollkästchens erhalten Sie über die Eigenschaft **Checked**.

- 3.7 Die Auswahl eines Eintrags in einem Listen- beziehungsweise Kombinationsfeld erfolgt über die Eigenschaft **SelectedIndex**. Die Nummerierung der Einträge beginnt dabei mit 0 und nicht mit 1.

Kapitel 4

- 4.1 Die Daten in einer **TextBox** werden immer als Zeichenkette geliefert – auch dann, wenn sich in dem Feld nur numerische Werte befinden.
- 4.2 Die Ausrichtung des Textes in einer **TextBox** wird über die Eigenschaft **TextAlign** gesteuert.
- 4.3 Nein, die Optionsfelder hängen nicht zusammen. Sie können daher in der **GroupBox** eine Option markieren, ohne dass die Markierung der Optionsfelder direkt im Formular verändert wird.
- 4.4 Es werden vier `if`-Abfragen benötigt. Der Status jedes einzelnen Optionsfelds muss einzeln abgefragt werden.
- 4.5 Um das Maximieren-Symbol zu deaktivieren, setzen Sie die Eigenschaft **MaximizeBox** des Formulars auf `False`.

B. Glossar

Aktivierreihenfolge	Die Aktivierreihenfolge in einem Formular bestimmt, in welcher Reihenfolge der Fokus auf die Steuerelemente gesetzt wird.
Container	Ein Container nimmt weitere Steuerelemente auf.
Eigenschaftenfenster	Das Eigenschaftenfenster ist ein Teil der Oberfläche von Visual Studio. Sie können hier Eigenschaften von Steuerelementen und die Reaktion auf Ereignisse festlegen.
Ereignis	Ein Ereignis ist eine Art Signal, das von bestimmten Aktionen ausgelöst wird. Zu den Aktionen, die ein Ereignis auslösen, gehören zum Beispiel das Bewegen der Maus, das Klicken, das Doppelklicken oder auch eine Tastatureingabe.
Event handling	<i>Event handling</i> ist die englische Bezeichnung für Ereignisverarbeitung.
Eventhandler	Der <i>Eventhandler</i> stellt eine Verbindung zwischen einem Ereignis und einer Methode her, die beim Eintreten des Ereignisses ausgeführt werden soll.
Fokus	Der Fokus in einem Formular bestimmt, welches Steuerelement aktuell auf Tastatureingaben reagiert.
Formular	Ein Formular von Visual Studio entspricht einem Fenster einer Windows Forms-Anwendung.
MFC	Siehe <i>Microsoft Foundation Classes</i>
Microsoft Foundation Classes	Bei den <i>Microsoft Foundation Classes</i> handelt es sich um spezielle Klassen, die zahlreiche Funktionen der Windows API kapseln. Sie ermöglichen so einen leichteren und gleichzeitig auch komfortableren Zugriff auf die Windows API.
Pixel	Ein Pixel ist ein Bildpunkt.
Signal	Ein Signal ist eine Form der Kommunikation zwischen Objekten.
Sender	Ein Sender setzt eine Nachricht ab, die dann von verschiedenen Empfängern verarbeitet werden kann. Ob die Nachricht verarbeitet wird, hängt vom Empfänger ab.
Steuerelement	Die Steuerelemente von Visual Studio stellen vorgefertigte Bedienelemente, Objekte und Methoden zur Verfügung.
this-Referenz	Die <i>this</i> -Referenz bezieht sich auf die Instanz einer Klasse, die eine Methode aufgerufen hat.

Toolbox	Die Toolbox ist ein Teil der Oberfläche von Visual Studio. Sie ermöglicht das komfortable Einfügen von Steuerelementen.
Universal Windows Platform App	Eine Universal Windows Platform App ist eine Anwendung, die speziell für das Betriebssystem Windows 10 erstellt wird. Sie ist für den Einsatz auf mobilen Geräten wie einem Tablet Computer oder einem Smartphone gedacht, kann aber auch auf einem normalen Rechner eingesetzt werden.
Windows API	Die Windows API ist eine Schnittstelle zum direkten Aufruf von Windows-Funktionen aus eigenen Programmen.
Windows Forms-Anwendung	Bei einer Windows Forms-Anwendung wird die Oberfläche eines Programms im Wesentlichen aus Formularen und vorgefertigten Steuerelementen zusammengebaut. Die eigentliche Programmierarbeit beschränkt sich vor allem auf die Logik.
Windows Presentation Foundation	Die <i>Windows Presentation Foundation</i> ist eine Plattform zur Entwicklung von grafischen Oberflächen. Die Entwicklung ist eindeutig getrennt in die Logik auf der einen Seite und die Präsentation auf der anderen Seite.
WPF	Siehe <i>Windows Presentation Foundation</i>

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmietechniken.* 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in Visual C# mit Visual Studio 2019. Ideal für Programmieranfänger geeignet.* 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Ein typisches Anwendungsfenster unter Windows	4
Abb. 1.2	Das Fenster Neues Projekt erstellen	5
Abb. 1.3	Das fertige Projekt für eine Windows Forms-Anwendung	6
Abb. 1.4	Die Toolbox von Visual Studio	7
Abb. 1.5	Die erste Windows-Anwendung	8
Abb. 2.1	Das eingeblendete Fenster Eigenschaften	11
Abb. 2.2	Das Fenster Eigenschaften mit der Anzeige der Ereignisse	12
Abb. 2.3	Der geänderte Titel des Formulars	13
Abb. 2.4	Die Eigenschaft Size	14
Abb. 2.5	Die Untereigenschaften der Eigenschaft Size	15
Abb. 2.6	Das Formular mit der neuen Breite und Höhe	15
Abb. 2.7	Die Schaltfläche im Formular	17
Abb. 2.8	Die beiden Steuerelemente im Formular	18
Abb. 2.9	Das fertige Formular	18
Abb. 2.10	Die Ereignisse für eine Schaltfläche im Eigenschaftenfenster	21
Abb. 2.11	Die Methode Button1_Click() im Editor	22
Abb. 3.1	Das Formular im „Rohbau“	27
Abb. 3.2	Das Formular mit den beiden Labels	29
Abb. 3.3	Die Auswahl der Methoden für das Ereignis DoubleClick des Labels ...	31
Abb. 3.4	Die ComboBox-Aufgaben	32
Abb. 3.5	Das Formular mit dem Listenfeld	33
Abb. 3.6	Das Kontrollkästchen im Einsatz	35
Abb. 3.7	Die angepassten Steuerelemente	36
Abb. 4.1	Das Formular für den Taschenrechner im „Rohbau“	41
Abb. 4.2	Die beiden Eingabefelder für die Zahlen	41
Abb. 4.3	Das fertige Formular für den Taschenrechner	43
Abb. 4.4	Der Dialog bei einer ungültigen Eingabe	45
Abb. 4.5	Das Formular nach den kosmetischen Korrekturen	47
Abb. 4.6	Das vergrößerte Fenster des Taschenrechners	48
Abb. 4.7	Einstellen der Aktivierreihenfolge	50

E. Codeverzeichnis

Code 2.1	Der Quelltext der Methode Button1_Click()	23
Code 2.2	Die vollständige Methode Button1_Click()	24
Code 3.1	Der Quelltext für die Methode ButtonKopieren_Click()	29
Code 3.2	Die Anweisung für die Methode ComboBox1_SelectedIndexChanged()	33
Code 3.3	Die Anweisung für die Methode ListBox1_SelectedIndexChanged()	34
Code 3.4	Die Anweisungen zum Ein- und Ausblenden der Felder	35
Code 3.5	Die Anweisungen für die Methode Form1_Load()	37
Code 4.1	Die Methode zum Berechnen	44

F. Medienverzeichnis

Video 2.1	Arbeiten mit dem Eigenschaftenfenster	17
Video 3.1	Steuerelemente einfügen und positionieren.....	28
Video 4.1	Ändern der Aktivierreihenfolge	51

G. Sachwortverzeichnis

A

Aktivierreihenfolge	48
Anwendungsfenster	4
Ausnahmebehandlung	45

C

Container	42
-----------------	----

E

Eigenschaft	
einstellen	10
Eigenschaftenfenster	10
Tipps zum Arbeiten mit dem	16
Eingabefeld	41
Ereignis	20
Verarbeitung von	20
Ereignisverarbeitung	20
event handling	20

F

Fokus	49
Form1	6

H

Hauptfenster	6
--------------------	---

M

MFC	3
Microsoft Foundation Classes	3

S

Steuerelement	
ein- und ausblenden	34
einfügen und positionieren	17
Tipps zum Einfügen von	19

T

Toolbox	6
---------------	---

U

Untereigenschaft	14
------------------------	----

W

Werkzeugkasten	6
Windows API	3
Windows Forms	3
Windows Forms-Anwendung	
Projekt für eine, erstellen	5
Windows Presentation Foundation	3
WPF	3

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP07D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:
Datum:
Note:
Unterschrift Fernlehrer/in:

Senden Sie bitte bei allen Aufgaben, bei denen Sie Programme erstellen oder ändern sollen, immer die vollständigen Projekte ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

- Wie lautet der vollständige Kopf der Methode, die für das Ereignis **MouseClick** einer Schaltfläche `buttonTest` angelegt wird? Notieren Sie bitte den gesamten Kopf der Methode, die Visual Studio erzeugt.

5 Pkt.

- Sie arbeiten in einer Anwendung mit zwei einfachen Eingabefeldern. In dem einen Feld befindet sich der Wert 20 und in dem anderen Feld der Wert 30. Welches Ergebnis erhalten Sie, wenn Sie die Werte in den beiden Feldern ohne Umwandlung addieren? Begründen Sie bitte Ihre Antwort.

5 Pkt.

- Ändern Sie den Taschenrechner aus diesem Studienheft so, dass die Auswahl der Rechenoperation nicht mehr über die Optionsfelder erfolgt, sondern über ein Kombinationsfeld. Sorgen Sie dafür, dass in diesem Kombinationsfeld die erste Rechenoperation in Ihrer Liste bereits beim Programmstart angezeigt wird.

Beschreiben Sie für die Lösung dieser Aufgabe bitte zusätzlich die einzelnen Schritte, die für die Änderungen erforderlich sind. Notieren Sie außerdem die vollständige Methode, die den Eintrag im Kombinationsfeld auswertet und die Rechenoperation durchführt.

Ein kleiner Tipp zur Lösung:

Sie müssen nicht auf das Ändern eines Eintrags im Kombinationsfeld reagieren. Es reicht, wenn Sie vor der Berechnung den aktuellen Eintrag im Kombinationsfeld auswerten – zum Beispiel über die Eigenschaft `SelectedIndex`.

40 Pkt.

4. Erstellen Sie ein Programm, in dem ein Label mit beliebigem Inhalt nach dem Anklicken einer Schaltfläche von oben nach unten durch das Formular wandert. Die Wanderung soll von den Koordinaten 0 bis 300 gehen und mindestens 10 Mal wiederholt werden. Diese Wiederholungen sollen beim Anklicken der Schaltfläche automatisch gestartet werden. Pro Schritt soll das Label um einen Pixel nach unten verschoben werden.

Ein Hinweis für die Lösung:

Der Abstand eines Steuerelements vom oberen Rand des Formulars wird durch die Eigenschaft **Top** bestimmt.

Die Anweisung

```
label1.Top = 10;
```

setzt zum Beispiel den Abstand des Labels `label1` vom oberen Rand des Formulars auf 10.

50 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Ein Bildbetrachter

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP08D

Objektorientierte Software-Entwicklung mit C#

Ein Bildbetrachter

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Ein Bildbetrachter

Inhaltsverzeichnis

Einleitung	1
1 Das Grundgerüst des Bildbetrachters	3
Zusammenfassung	7
2 Die Anzeige von einzelnen Bildern	9
2.1 Das Einfügen der Steuerelemente	9
2.2 Das Anzeigen der Grafik	10
2.3 Der Standarddialog Öffnen	14
2.4 Anzeige in einem neuen Formular	19
Zusammenfassung	25
3 Die Bilderschau	28
3.1 Vorüberlegungen und Vorbereitungen	28
3.2 Die Auswahl der Dateien	29
3.3 Die Anzeige der Bilder	32
Zusammenfassung	37
4 Der Feinschliff für den Bildbetrachter	40
4.1 Korrektur von Schönheitsfehlern	40
4.2 Soundwiedergabe	42
Zusammenfassung	45
Schlussbetrachtung	47
Anhang	
A. Lösungen der Aufgaben zur Selbstüberprüfung	48
B. Glossar	51
C. Literaturverzeichnis	53
D. Abbildungsverzeichnis	54
E. Tabellenverzeichnis	55
F. Codeverzeichnis	56
G. Sachwortverzeichnis	57
H. Einsendeaufgabe	59

Einleitung

In diesem Studienheft werden Sie Ihre Kenntnisse beim Erstellen von Windows Forms-Anwendungen vertiefen. Dazu erstellen wir einen Bildbetrachter zur Anzeige von Grafikdateien. Neben der Anzeige einzelner Bilder soll das Programm außerdem eine Art „Diaschau“ abspielen können, bei der die Bilder automatisch nach einem bestimmten Zeitraum wechseln.

Auch bei dieser Anwendung werden wir vor allem vorgefertigte Steuerelemente von Visual Studio einsetzen.

Im Einzelnen lernen Sie in diesem Studienheft,

- wie Sie ein Formular direkt nach dem Start maximieren lassen,
- wie Sie ein Formular mit mehreren Registern erstellen,
- wie Sie Steuerelemente automatisch an Größenänderungen eines Formulars anpassen,
- wie Sie eine **PictureBox** zum Anzeigen von Grafiken verwenden,
- wie Sie Grafiken zur Laufzeit eines Programms in eine **PictureBox** laden,
- wie Sie Meldungen über eine **MessageBox** erzeugen,
- wie Sie überprüfen, ob eine Datei vorhanden ist,
- wie Sie die Anzeige einer Grafik in einer **PictureBox** festlegen,
- wie Sie einen Öffnendialog von Windows in eigenen Programmen einsetzen,
- wie Sie Methoden für die Verarbeitung eines Ereignisses selbst aufrufen,
- wie Sie die Anzeige von Dateien in einem Öffnendialog über Filter festlegen,
- wie Sie neue Formulare in einer Windows Forms-Anwendung erzeugen,
- wie Sie ein Formular modal anzeigen lassen,
- wie Sie auf Felder und Methoden in einem anderen Formular zugreifen,
- wie Sie mehrere Dateien in einem Öffnendialog auswählen lassen,
- wie Sie ein Listenfeld zur Laufzeit eines Programms mit Einträgen füllen,
- wie Sie Anweisungen zeitgesteuert über einen Timer ausführen lassen,
- wie Sie Einträge in einem Listenfeld löschen und
- wie Sie Sounds in Ihren Anwendungen wiedergeben.

Christoph Siebeck

1 Das Grundgerüst des Bildbetrachters

In diesem Kapitel erstellen wir das Grundgerüst für unseren Bildbetrachter.

Dazu zunächst einige Vorüberlegungen:

- Wir bieten die Einzelanzeige und die Bilderschau in zwei getrennten Bereichen des Formulars an. Damit wir dabei so viel Platz wie eben möglich haben, untergliedern wir das Formular in zwei Register. Das erste Register soll die einzelnen Bilder anzeigen und das zweite Register die Bilderschau. Die Funktionen zur Steuerung bringen wir ebenfalls getrennt auf den beiden Registern unter.
- Damit der Anwender die Bilder so groß wie eben möglich anzeigen lassen kann, soll das Formular nach dem Start nicht mehr als Fenster angezeigt werden, sondern direkt als Vollbild – also maximiert. Gleichzeitig soll der Anwender aber auch die Möglichkeit haben, das Fenster über den Fensterrahmen selbst in der Größe zu verändern.
- Damit die Steuerelemente beim Verändern der Formulargröße nicht „irgendwo“ im Formular herumstehen, sorgen wir dafür, dass sie immer an derselben Position stehen bleiben – allerdings relativ zum Fenster. Damit bewegen sich die Steuerelemente also beim Verändern der Fenstergröße mit.

Die praktische Umsetzung dieser Vorüberlegungen ist nicht allzu schwierig. Legen Sie bitte im ersten Schritt ein neues Projekt für eine Windows Forms-Anwendung an. Als Projektnamen können Sie zum Beispiel **Bildbetrachter** verwenden.

Ändern Sie dann den Text in der Titelleiste des Fensters in **Bildbetrachter** und setzen Sie die Eigenschaft **WindowState**¹ in der Gruppe **Layout** auf **Maximized**. Damit wird das Fenster direkt nach dem Start maximiert dargestellt.

Fügen Sie dann ein Steuerelement **TabControl**² für die beiden Register ein. Sie finden dieses Steuerelement in der Toolbox in der Gruppe **Container**. Positionieren Sie das Steuerelement oben links im Formular und vergrößern Sie es, bis es nahezu den gesamten freien Platz im Fenster einnimmt.

Legen Sie dann die Eigenschaften für die beiden Register fest. Das geht am einfachsten, wenn Sie auf das Symbol im Feld rechts hinter der Eigenschaft **TabPages**³ in der Gruppe **Verhalten** klicken. Das Symbol wird allerdings nur dann angezeigt, wenn Sie die Eigenschaft ausgewählt haben.

1. **WindowState** lässt sich mit „Fensterzustand“ übersetzen.

2. *Tab* bedeutet übersetzt so viel wie „Karteireiter“ oder „Etikett“.

3. *Pages* bedeutet übersetzt „Seiten“.

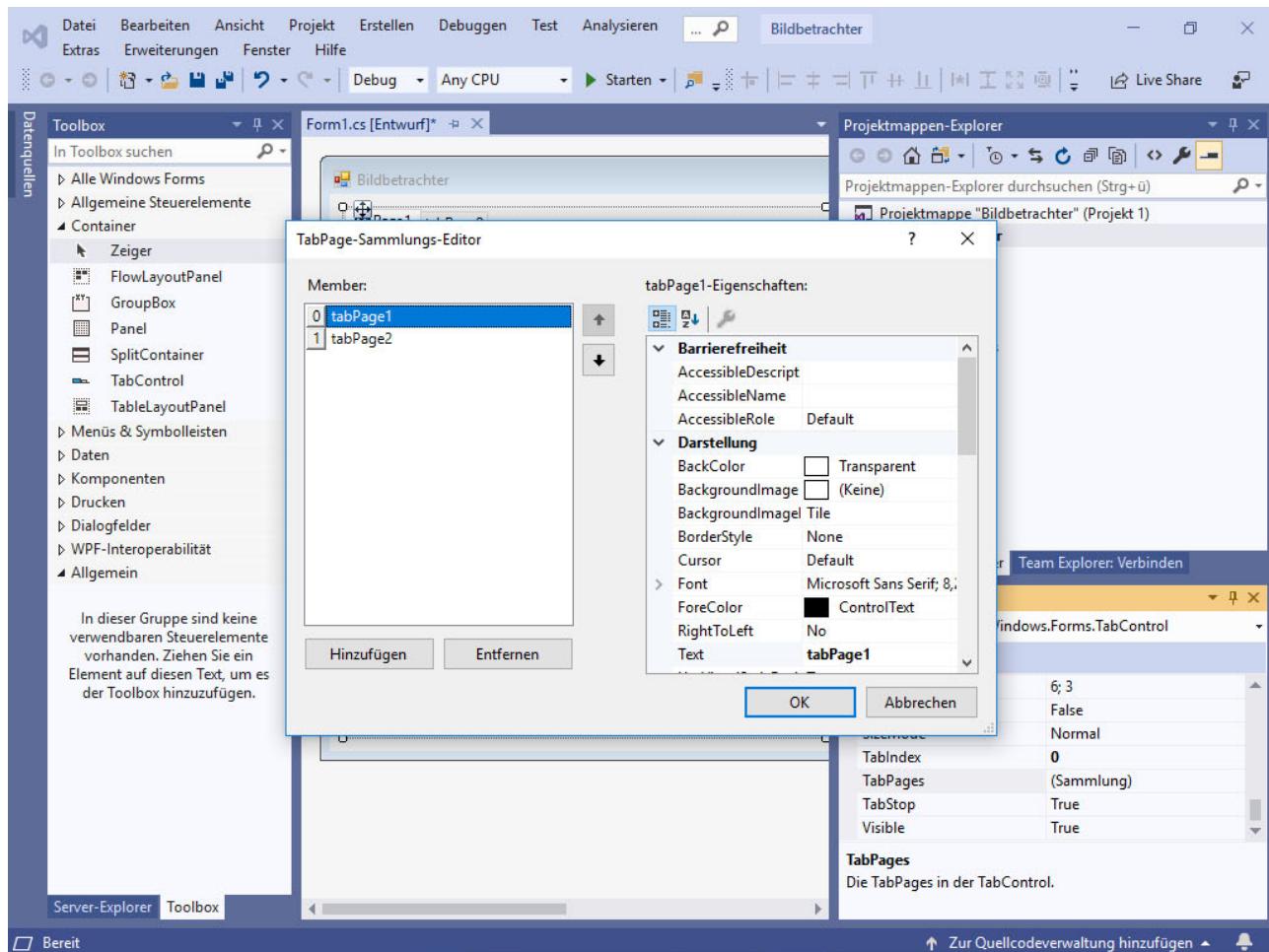


Abb. 1.1: Der tabPage-Sammlungs-Editor

Im Fenster **TabPage-Sammlungs-Editor** können Sie dann im linken Bereich das gewünschte Register auswählen und im rechten Bereich die Eigenschaften festlegen. Das erfolgt genauso wie im Eigenschaftenfenster. Über die Schaltflächen links unten im Fenster können Sie außerdem weitere Register hinzufügen oder vorhandene Register entfernen.

Ändern Sie jetzt bitte den Text auf der Zunge des ersten Registers in **Einzelbild**. Dazu bearbeiten Sie wie gewohnt die Eigenschaft **Text**. Vergeben Sie dann noch einen „sprechenden“ Namen für das Register – zum Beispiel `tabPageEinzel`.

Ändern Sie dann den Text auf der Zunge des zweiten Registers in **Bilderschau** und vergeben Sie auch für dieses Register einen „sprechenden“ Namen. Wir benutzen in unserem Beispiel den Namen `tabPageSchau`. Klicken Sie abschließend auf die Schaltfläche **OK**, um den tabPage-Sammlungs-Editor wieder zu schließen.

Tipp:

Sie können die Eigenschaften eines einzelnen Registers auch direkt im Eigenschaftenfenster bearbeiten. Wechseln Sie dazu zuerst im Designer über die Registerzunge in das gewünschte Register und klicken Sie anschließend in den großen Bereich in der Mitte des Registers. Im Eigenschaftenfenster werden dann die Eigenschaften des ausgewählten Registers angezeigt.

Die Funktionen zum Hinzufügen und Löschen von Registerkarten finden Sie auch im Kontextmenü des TabControl-Steuerelements.

Wenn Sie wieder das gesamte TabControl markieren wollen, klicken Sie am einfachsten in den Bereich mit den Registerzungen oben im Steuerelement. Ob nur ein einzelnes Register oder das gesamte Steuerelement markiert ist, erkennen Sie sehr schnell daran, ob die Markierung Anfasser hat. Diese Anfasser werden nämlich nur dann angezeigt, wenn Sie das gesamte Steuerelement markiert haben. Wenn Sie ganz sicher gehen wollen, können Sie auch im Kombinationsfeld oben im Eigenschaftenfenster nachsehen, welches Element aktuell markiert ist.

Ihr Formular sollte nun ungefähr so aussehen:

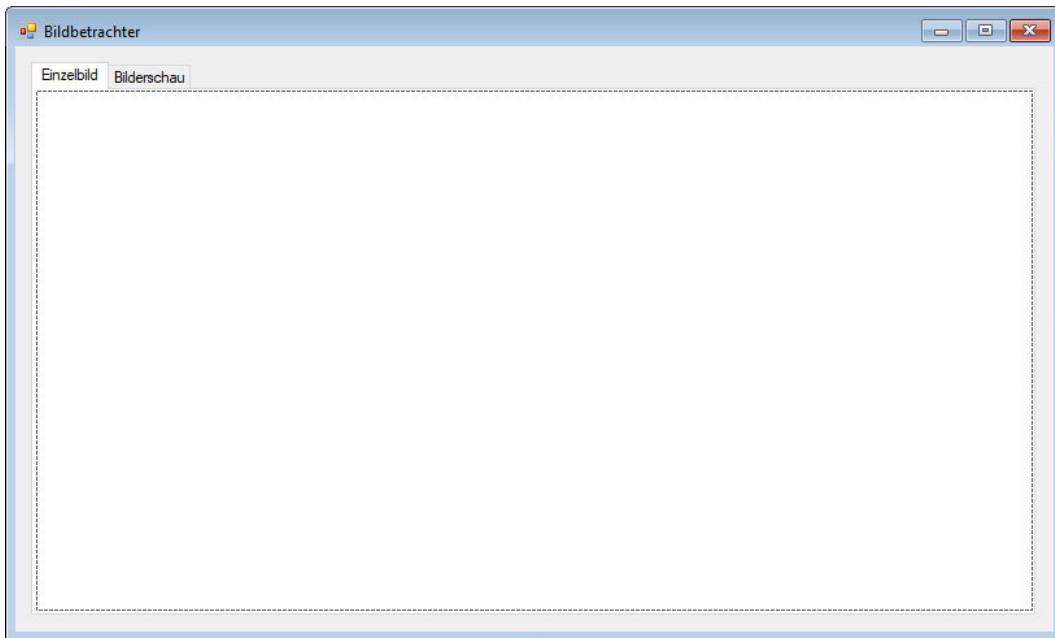


Abb. 1.2: Das Formular mit den beiden Registern

Jetzt müssen wir noch sicherstellen, dass sich der Abstand der Register zu den Rändern des Formulars nicht mehr verändert. Dadurch passen sich die Register bei Größenänderungen des Formulars automatisch an die neue Größe des Formulars an. Dazu müssen Sie nichts weiter machen, als das **TabControl**-Steuerelement an allen vier Rändern des Formulars zu verankern. Das erfolgt über die Eigenschaft **Anchor**⁴ in der Gruppe **Layout**.

Überprüfen Sie bitte, ob das **TabControl**-Steuerelement im Designer markiert ist. Klicken Sie dann in die Eigenschaft **Anchor** und öffnen Sie die Liste für die Eigenschaften mit einem Mausklick auf das Symbol .

4. Übersetzt bedeutet *anchor* „Anker“.

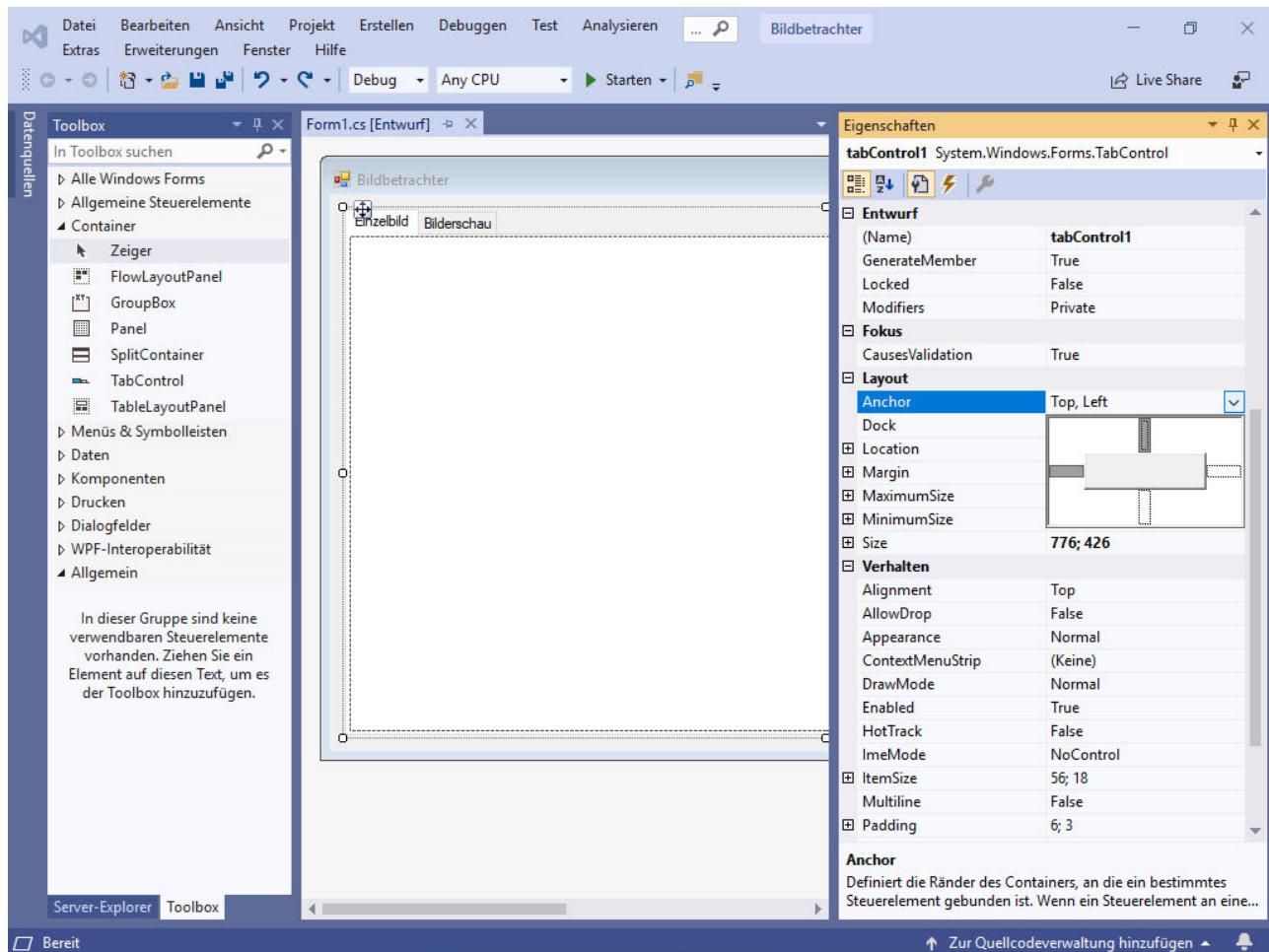


Abb. 1.3: Die Auswahl der Anker
(rechts im Eigenschaftenfenster; der Projektmappen-Explorer ist zur besseren Übersicht ausgeblendet)

In der Grafik, die jetzt eingeblendet wird, können Sie nun die gewünschten Anker durch einen Mausklick auf eins der Rechtecke festlegen beziehungsweise einen gesetzten Anker durch einen Mausklick wieder entfernen. Wie Sie an den Markierungen erkennen, wird ein Steuerelement in der Standardeinstellung immer oben und links verankert. Um es auch unten und rechts zu verankern, klicken Sie nacheinander auf die entsprechenden Markierungen. Danach sollten alle vier Anker markiert sein. Drücken Sie abschließend die Eingabetaste oder klicken Sie an eine beliebige Stelle außerhalb der Grafik, um die Änderungen zu übernehmen.

Speichern Sie dann das Projekt und testen Sie das Programm.

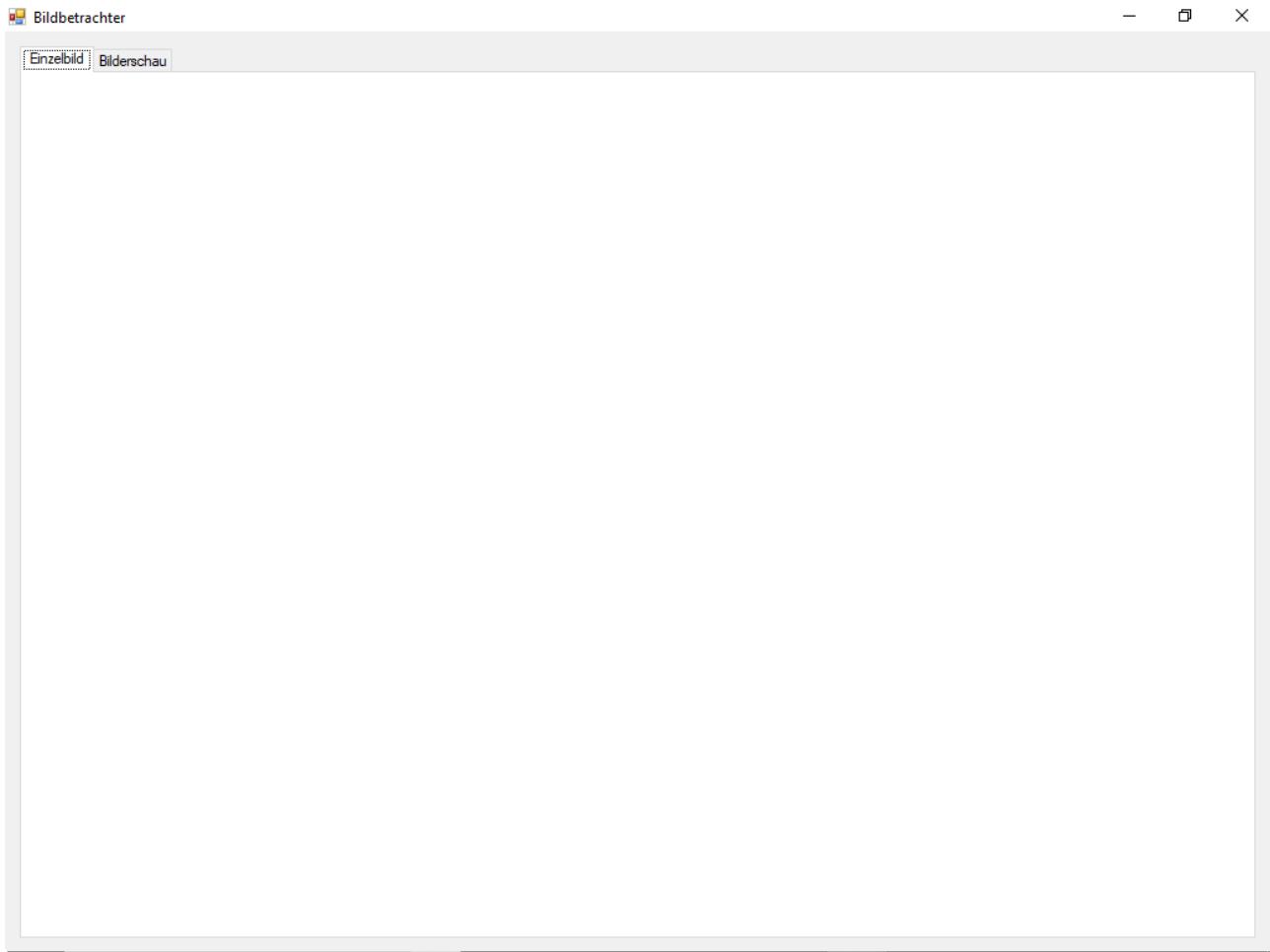


Abb. 1.4: Der erste Test des Programms

Das Fenster sollte als Vollbild erscheinen und die Register nahezu das gesamte Formular einnehmen. Probieren Sie auch einmal aus, was bei Größenänderungen des Fensters geschieht, und wechseln Sie ein paar Mal zwischen den Registern hin und her. Wenn Sie die Anker richtig gesetzt haben, passen sich die Register automatisch an die Größe des Formulars an.

So viel zum Grundgerüst des Bildbetrachters. Im nächsten Kapitel werden wir die Einzelanzeige der Bilder umsetzen.

Zusammenfassung

Über die Eigenschaft **WindowState** können Sie festlegen, wie ein Fenster direkt nach dem Start angezeigt wird.

Um Register in ein Formular einzufügen, benutzen Sie das Steuerelement **TabControl**.

Über die Eigenschaft **Anchor** können Sie Komponenten am Rand des Formulars verankern.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Auf welchen Wert setzen Sie die Eigenschaft **WindowState** eines Formulars, damit eine Anwendung direkt nach dem Start als Vollbild angezeigt wird?

- 1.2 Wie können Sie die Eigenschaften eines einzelnen Registers in einem **TabControl** verändern?

- 1.3 Sie wollen, dass ein Steuerelement automatisch bei Größenänderungen des Formulars angepasst wird. Wie können Sie das erreichen?

2 Die Anzeige von einzelnen Bildern

In diesem Kapitel programmieren wir die Anzeige von einzelnen Bildern.

Dazu zunächst wieder einige Vorüberlegungen:

- Im ersten Schritt soll der Anwender den Namen des Bildes selbst in ein Eingabefeld eingeben und dann die Anzeige über eine Schaltfläche starten.
- In einem zweiten Schritt stellen wir einen Öffnendialog zur Verfügung, in dem der Anwender komfortabel durch das Dateisystem navigieren und das gewünschte Bild per Mausklick auswählen kann.
- Neben der einfachen Anzeige der Bilder in dem Register sollen die Bilder zusätzlich auch noch vergrößert dargestellt und auf Wunsch in einem eigenen Formular angezeigt werden können.

2.1 Das Einfügen der Steuerelemente

Fügen Sie jetzt drei Schaltflächen rechts oben in das erste Register ein. Die erste Schaltfläche soll den Text **Anzeigen** erhalten, die zweite den Text **Öffnen** und die dritte den Text **Beenden**. Verankern Sie alle drei Schaltflächen oben und rechts. Sorgen Sie dann dafür, dass beim Anklicken der dritten Schaltfläche die Anwendung geschlossen wird.

Tipp:

Wenn Sie alle drei Schaltflächen markieren, wirken sich die Änderungen der Eigenschaft **Anchor** auf alle Schaltflächen gleichzeitig aus. Damit ersparen Sie sich das separate Setzen der Eigenschaft für jedes einzelne Steuerelement.

Fügen Sie anschließend ein Eingabefeld und zwei Kontrollkästchen in das erste Register ein. Das erste Kontrollkästchen soll die Beschriftung **Optimale Größe** erhalten und das zweite Kontrollkästchen die Beschriftung **Neues Fenster**. Positionieren Sie die drei Steuerelemente unten links. Das Eingabefeld können Sie dabei zum Beispiel oberhalb der beiden Kontrollkästchen ablegen und ein wenig verbreitern. Verankern Sie dann die drei Steuerelemente am unteren und linken Rand.

Hinweis:

Die Schaltfläche **Öffnen** benötigen wir erst später beim Anzeigen des Öffnendialogs. Wir fügen sie aber bereits jetzt in das Formular ein, um uns Umbauten zu ersparen.

Bitte denken Sie selbst daran, vor allem für Steuerelemente, die Sie mehrfach in einem Formular verwenden, sprechende Namen zu vergeben. Im weiteren Verlauf des Studienhefts werden wir Sie daran nicht immer wieder erinnern, sondern Ihnen nur noch – wenn erforderlich – die Bezeichner angeben, die wir für die Beispiele verwenden.

Im letzten Schritt fügen wir jetzt das Steuerelement für die Anzeige der Grafiken ein – eine **PictureBox**⁵. Sie finden dieses Steuerelement in der Gruppe **Allgemeine Steuerelemente** der Toolbox. Positionieren Sie die **PictureBox** bitte oben links im Register und vergrößern Sie sie dann so, dass sie den maximal freien Platz auf dem Register einnimmt. Verankern Sie das Steuerelement anschließend an allen vier Seiten.

Das Register sollte nach dem Einfügen und Positionieren der Steuerelemente ungefähr so aussehen:

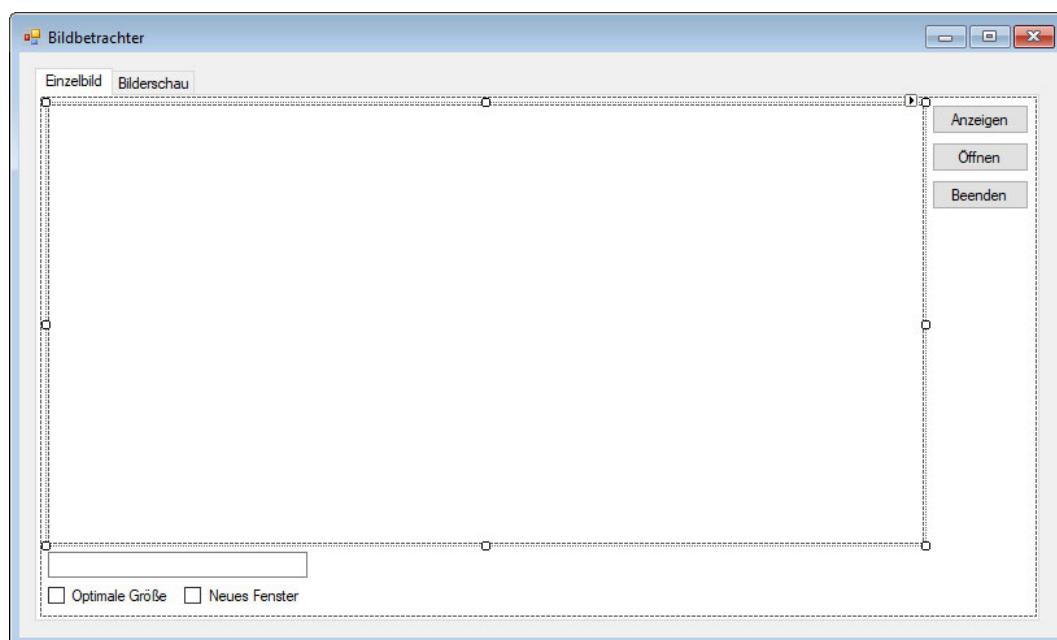


Abb. 2.1: Die Steuerelemente im ersten Register

Probieren Sie jetzt auch noch einmal aus, ob Sie die Steuerelemente richtig verankert haben. Lassen Sie dazu die Anwendung ausführen und verkleinern beziehungsweise vergrößern Sie das Fenster einige Male.

Tipp:

Die Größe der **PictureBox** können Sie in der laufenden Anwendung nicht erkennen, da sie leer ist. Sie können aber zum Test das Steuerelement mit einem Rahmen versehen. Setzen Sie dazu die Eigenschaft **BorderStyle** in der Gruppe **Darstellung** der **PictureBox** auf den gewünschten Wert.

2.2 Das Anzeigen der Grafik

Kommen wir nun zur Anzeige der Bilder. Dazu überprüfen wir beim Anklicken der Schaltfläche **Anzeigen** zuerst, ob überhaupt ein Eintrag im Eingabefeld gemacht wurde. Wenn das der Fall ist, kontrollieren wir über die Methode `System.IO.File.Exists()`⁶, ob die angegebene Datei existiert. Danach rufen wir

5. **PictureBox** lässt sich mit „Bildfeld“ übersetzen.

6. *Exists* bedeutet übersetzt so viel wie „existiert“.

entweder die Methode `Load()` für die **PictureBox** auf und übergeben dabei den Text aus dem Eingabefeld als Argument, oder wir lassen über die Methode `MessageBox.Show()` eine Meldung ausgeben.

Hinweis:

Sie können das Bild in einer **PictureBox** auch über das Eigenschaftenfenster setzen. Dazu wählen Sie das gewünschte Bild über die Eigenschaft **Image** in der Gruppe **Darstellung** und die Schaltfläche **Importieren...** für eine lokale Ressource aus. In unserem Fall hilft uns diese feste Vorgabe aber nicht viel weiter, da wir die Bilder ja zur Laufzeit des Programms verändern möchten.

Die entsprechenden Anweisungen für das Ereignis **Click** der Schaltfläche **Anzeigen** finden Sie im folgenden Code:

```
//wenn ein Eintrag im Eingabefeld steht, laden wir das  
//entsprechende Bild  
if (textBox1.Text != String.Empty)  
{  
    //existiert die Datei überhaupt?  
    if (System.IO.File.Exists(textBox1.Text))  
        //wenn ja, dann laden und anzeigen  
        pictureBox1.Load(textBox1.Text);  
    else  
        MessageBox.Show("Die Datei existiert nicht!", "Fehler");  
}
```

Code 2.1: Die Methode für das Anklicken der Schaltfläche **Anzeigen**

Tipp:

Sie können in einer **MessageBox** auch Standardsymbole anzeigen lassen. Die folgende Anweisung erzeugt zum Beispiel eine Meldung mit einer **OK**-Schaltfläche und einem weißen Kreuz in einem roten Kreis:

```
MessageBox.Show("Die Datei existiert nicht!",  
    "Fehler", MessageBoxButtons.OK,  
    MessageBoxIcon.Error);
```

Wenn Sie allerdings ein Symbol anzeigen lassen, müssen Sie auch die Schaltflächen in dem Dialog angeben. Auch die Angaben `MessageBoxButtons`. und `MessageBoxIcon`. vor dem Namen der Schaltfläche beziehungsweise vor dem Namen des Symbols sind zwingend erforderlich. Mehr zu den verschiedenen Möglichkeiten für eine **MessageBox** finden Sie in der Hilfe unter dem Stichwort **MessageBox-Class**.

Übernehmen Sie jetzt die Anweisungen aus dem vorigen Code und probieren Sie die Anzeige der Bilder aus. Wenn Sie keine eigenen Bilder zum Test haben, finden Sie einige Grafiken in unterschiedlichen Formaten bei den Beispielen auf der Online-Lernplattform im Ordner `\beispiele\cshp08d`.

Die Anzeige könnte jetzt zum Beispiel so aussehen:

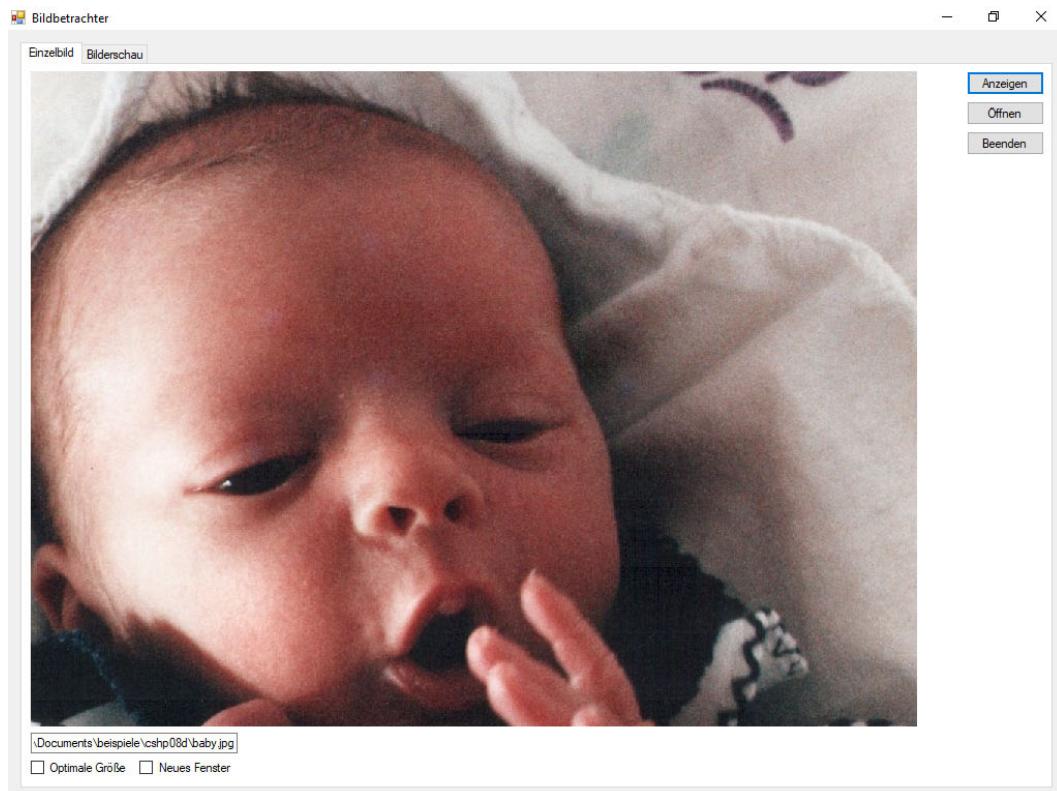


Abb. 2.2: Die erste angezeigte Grafik

Bei Bildern, deren Größe für die **PictureBox** passt, ist die Anzeige schon ganz brauchbar. Zu große Bilder werden allerdings – wie in der vorigen Abbildung – schlicht und einfach abgeschnitten. Wir müssen daher noch dafür sorgen, dass der Anwender über das Kontrollkästchen **Optimale Größe** bei Bedarf auch das komplette Bild in der **PictureBox** anzeigen lassen kann beziehungsweise kleine Bilder auf die Größe der **PictureBox** vergrößern kann. Dazu verändern wir beim Markieren des Kontrollkästchens die Eigenschaft **SizeMode**⁷ der **PictureBox** auf **Zoom**. Bei dieser Einstellung wird das Bild automatisch an die Größe der **PictureBox** angepasst, und zwar so, dass die Seitenverhältnisse beibehalten werden. Wenn die Markierung im Kontrollkästchen wieder gelöscht wird, lassen wir das Bild im Standardmodus **Normal** anzeigen.

Hinweis:

Für die Eigenschaft **SizeMode** gibt es noch einige weitere Werte. Die Einstellung **AutoSize**⁸ passt zum Beispiel die Größe der **PictureBox** automatisch an die Größe des Bildes an. Die Einstellung **StretchImage**⁹ sorgt dafür, dass die **PictureBox** vollständig mit dem Bild gefüllt wird. Dabei wird in der Regel aber auch das Seitenverhältnis des Bildes verändert und das Bild verzerrt. Probieren Sie die unterschiedlichen Einstellungen gleich einfach einmal selbst aus.

-
7. **SizeMode** lässt sich mit „Größenmodus“ übersetzen.
 8. **AutoSize** lässt sich mit „automatische Größe“ übersetzen.
 9. **StretchImage** steht für „strecke Bild“.

Damit der Anwender die Einstellungen für die optimale Größe sowohl vor als auch während der Anzeige verändern kann, programmieren wir die entsprechenden Anweisungen im Ereignis **CheckedChanged** des Kontrollkästchens **Optimale Größe**. Sie sehen so aus:

```
if (checkBoxOptGroesse.Checked == true)
    pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
else
    pictureBox1.SizeMode = PictureBoxSizeMode.Normal;
```

Code 2.2: Die Anweisungen zum Setzen der Eigenschaft **SizeMode**

Bitte beachten Sie:

Das Kontrollkästchen **Optimale Größe** hat in unserem Beispiel den Namen `checkBoxOptGroesse`. Sie müssen diesen Namen durch den Namen Ihres Kontrollkästchens ersetzen.

Die Angabe `PictureBoxSizeMode`. vor dem Modus ist zwingend erforderlich. Ohne diese Angaben beschwert sich der Compiler, dass er die Bezeichner `Zoom` beziehungsweise `Normal` nicht kennt.



Übernehmen Sie jetzt auch diese Änderungen und testen Sie das Programm dann noch einmal. Die Anzeige der Grafik in der optimalen Größe könnte dann zum Beispiel so aussehen:

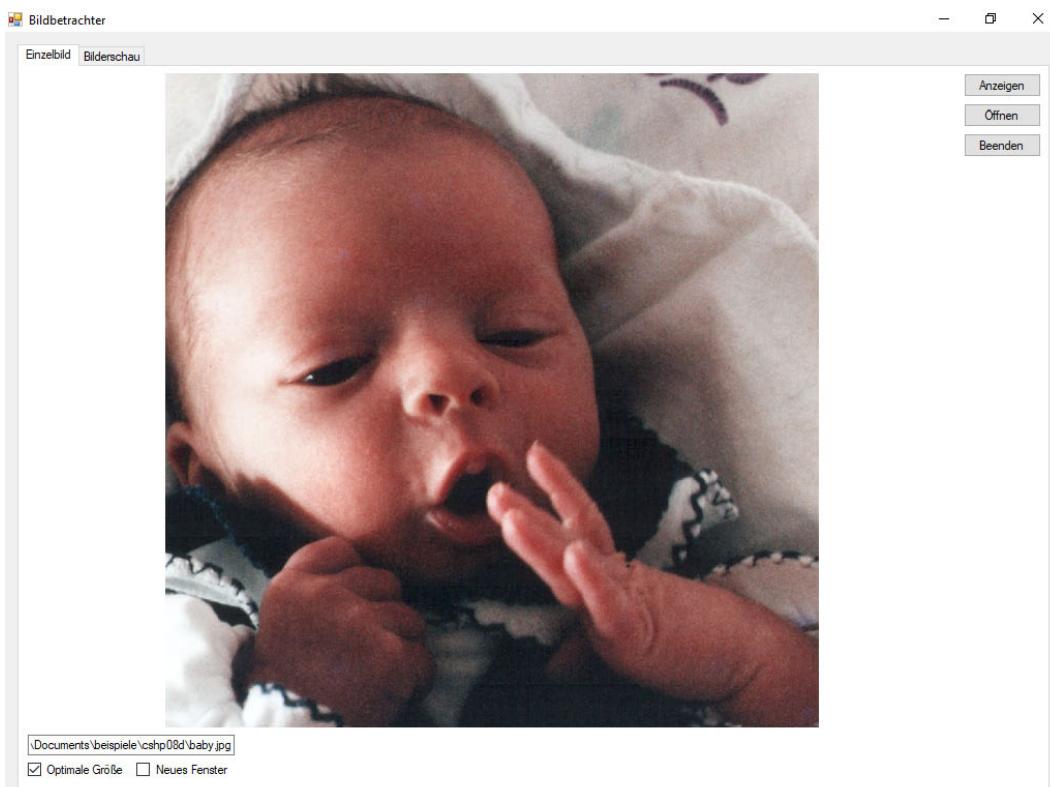


Abb. 2.3: Eine Grafik in der optimalen Größe

2.3 Der Standarddialog Öffnen

Im nächsten Schritt werden wir jetzt einen **Standarddialog Öffnen** zum Auswählen von Dateien in den Bildbetrachter einbauen. Dieser Dialog soll nach dem Anklicken der Schaltfläche **Öffnen** angezeigt werden. Damit muss der Anwender nicht mehr die umständliche und fehlerträchtige Auswahl über das Eingabefeld vornehmen.

Für den Dialog **Öffnen** verwenden wir ebenfalls ein fertiges Steuerelement – und zwar das Steuerelement **OpenFileDialog**¹⁰ in der Gruppe **Dialogfelder** der Toolbox. Fügen Sie dieses Element jetzt bitte in das Formular ein. Wo Sie den Dialog ablegen, spielt dabei keine Rolle.

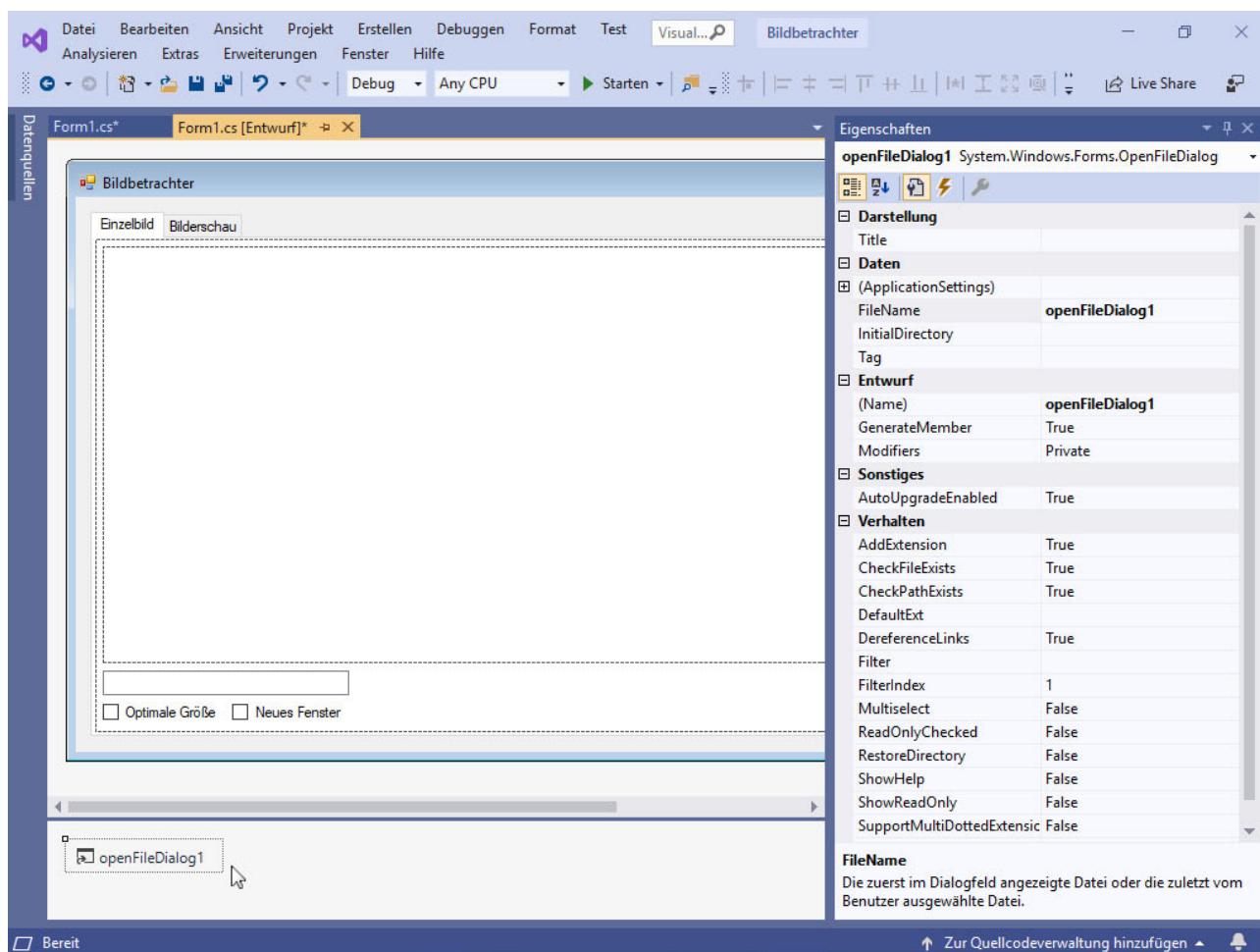


Abb. 2.4: Das Steuerelement **OpenFileDialog** im Formular
(unten links am Mauszeiger)

Wie Sie sehen, wird das Steuerelement nicht direkt im Formular abgelegt, sondern in einem eigenen Bereich unten im Designer. Denn bei dem Steuerelement **OpenFileDialog** handelt es sich um ein sogenanntes **nicht visuelles Steuerelement**.



Nicht visuelle Steuerelemente werden beim Ausführen nicht direkt im Formular angezeigt. Es handelt sich also nicht um Bedienelemente im eigentlichen Sinn.

10. OpenFileDialog bedeutet übersetzt so viel wie „Datei-Öffnen-Dialog“.

Damit der Anwender nun Dateien über den Dialog auswählen kann, müssen wir zunächst einmal dafür sorgen, dass der Dialog beim Anklicken der Schaltfläche **Öffnen** angezeigt wird. Dazu rufen wir in der Methode für das Ereignis **Click** der Schaltfläche die Methode `ShowDialog()`¹¹ für den Öffnendialog auf. Die entsprechende Anweisung sieht so aus:

```
openFileDialog1.ShowDialog();
```

Im zweiten Schritt müssen wir noch herausfinden, welche Datei der Anwender in dem Dialog ausgewählt hat, und diese Datei dann in die **PictureBox** übergeben. Dazu verwenden wir das Ereignis **FileOk** für den Dialog. Es tritt ein, wenn in dem Dialog eine Auswahl vorgenommen wurde und der Dialog mit der Schaltfläche **Öffnen** verlassen wird. Der Name der ausgewählten Datei wird dabei in der Eigenschaft `FileName`¹² des Steuerelements **OpenFileDialog** geliefert.

Da wir bereits eine fertige Methode zum Laden und Anzeigen einer Bilddatei haben, machen wir in der Methode für das Ereignis **FileOk** für den Dialog nichts weiter, als den Namen der ausgewählten Datei in das Eingabefeld zu schreiben und dann die Methode `ButtonAnzeigen_Click()` aufzurufen. Diese Methode lädt ja die Datei über das Eingabefeld in die **PictureBox**. Da die Methode `ButtonAnzeigen_Click()` die Argumente `sender` und `e` erwartet, reichen wir einfach die Werte weiter, die an die Methode `openFileDialog1_FileOk()` übergeben werden. Das ist in unserem Fall problemlos möglich, da wir ja in der Methode `ButtonAnzeigen_Click()` weder den Parameter `sender` noch den Parameter `e` verarbeiten.

Hinweis:

Wir haben in unserem Beispiel der Schaltfläche zum Anzeigen den Namen `buttonAnzeigen` gegeben. Wenn Ihre Schaltfläche einen anderen Namen hat, weicht auch der Name der Methode ab.

Noch einmal zu Auffrischung:

Die Parameter `sender` und `e` werden automatisch von der Anwendung an die Methoden weitergegeben. Der Parameter `sender` steht für das Steuerelement, das das Ereignis ausgelöst hat, und der Parameter `e` enthält weitere Informationen – zum Beispiel, welche Maustaste gedrückt wurde.



Die Anweisungen für die Methode `OpenFileDialog1_FileOk()` könnten dann zum Beispiel so aussehen:

```
//den Namen der ausgewählten Datei in das Eingabefeld
//schreiben
textBox1.Text = openFileDialog1.FileName;
//die eigene Methode ButtonAnzeigen_Click() aufrufen
ButtonAnzeigen_Click(sender, e);
```

Code 2.3: Die Anweisungen für die Methode `OpenFileDialog1_FileOk()`

11. `ShowDialog` bedeutet übersetzt so viel wie „zeige Dialog“.

12. `FileName` steht für „Dateiname“.

Hinweis:

Sie können die Methode `Load()` für die **PictureBox** auch direkt in der Methode `OpenFileDialog1_FileOk()` aufrufen. Denn der Öffnendialog überprüft selbst, ob eine Datei vorhanden ist, und erzeugt bei Fehlern auch selbst eine Meldung. Außerdem kann ja auch das Eingabefeld nicht mehr leer sein, da wir es mit dem Namen der ausgewählten Datei füllen. Damit müssten die entsprechenden Prüfungen in der Methode `ButtonAnzeigen_Click()` also nicht noch einmal ausgeführt werden. Die zweite Anweisung in der Methode `OpenFileDialog1_FileOk()` könnte also auch so aussehen:

```
pictureBox1.Load(textBox1.Text);
```

Möglich wäre auch das direkte Laden über den Dateinamen aus dem Dialog. Dann würde die Anweisung so aussehen:

```
pictureBox1.Load(openFileDialog1.FileName);
```

Welche der drei Varianten Sie wählen, ist Geschmackssache. In jedem Fall sollten Sie aber den Dateinamen in das Eingabefeld schreiben lassen, damit klar ist, welche Datei gerade angezeigt wird.

Probieren Sie den Öffnendialog jetzt einmal aus. Sie werden sehen, Sie können damit auch durch die Ordner navigieren und über die Einträge links im Dialog direkt auf Standardordner zugreifen – ohne viel selbst programmieren zu müssen.

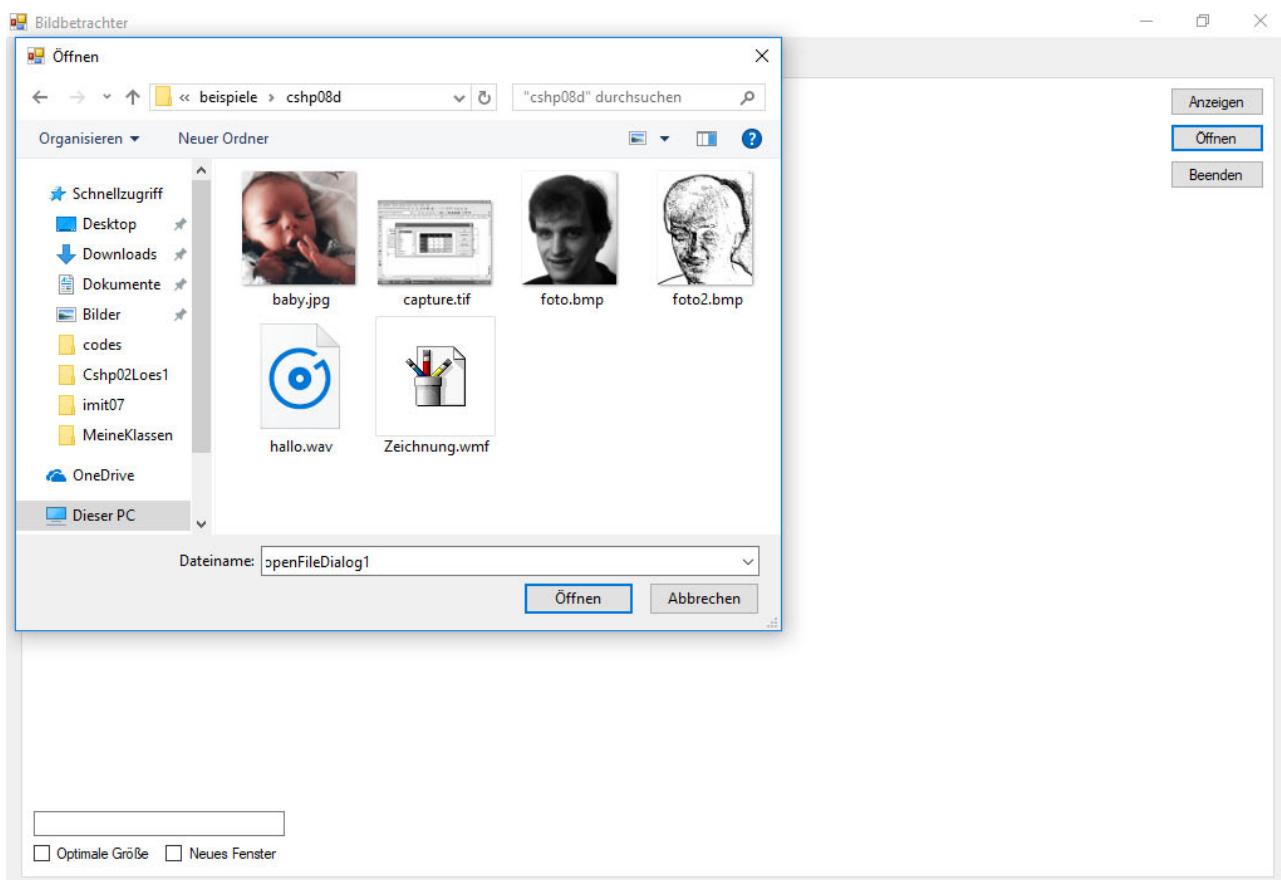


Abb. 2.5: Der Dialog **Öffnen** im Einsatz

Wenn Sie einmal genau hingesehen haben, werden Sie beim Test noch zwei Problemchen in unserem Dialog festgestellt haben: Zum einen erscheint direkt nach dem Öffnen immer der Text **openFileDialog1** im Feld **Dateiname:** und zum anderen werden immer sämtliche Dateien in einem Ordner angezeigt. Der Anwender kann also auch eine Datei auswählen, die unser Bildbetrachter gar nicht anzeigen kann – zum Beispiel eine Anwendung. Und das führt in dem Programm zu einer Ausnahme und einem unschönen Fehlerdialog. Wir müssen daher noch ein wenig an dem Dialog feilen.

Im ersten Schritt sorgen wir dafür, dass das Feld **Dateiname:** nach dem Öffnen leer ist. Dazu löschen Sie den Eintrag im Feld **FileName** in der Gruppe **Daten** für das Steuerelement.

Damit der Anwender nur noch Grafikdateien angezeigt bekommt, müssen wir dann noch einen Filter definieren. Dazu geben Sie bei der Eigenschaft **Filter** in der Gruppe **Verhalten** des Steuerelements zuerst den Namen des Filters an, der im Dialog **Öffnen** angezeigt werden soll, anschließend das Trennzeichen | und dann die Ausdrücke für den Filter. Die einzelnen Ausdrücke trennen Sie dabei durch ein Semikolon.

Hinweis:

Bei dem Zeichen | handelt es sich um das Pipe-Symbol. Sie erhalten das Zeichen mit der Tastenkombination **Alt Gr** + **[;]**.

Einige Beispiele für Filter finden Sie in der folgenden Tab. 2.1:

Tab. 2.1: Beispiele für die Eigenschaft **Filter**

Filter	angezeigter Name	angezeigte Dateien
BMP-Dateien *.bmp	BMP-Dateien	alle Dateien mit der Erweiterung .bmp
Grafikdateien *.bmp;*.jpg	Grafikdateien	alle Dateien mit den Erweiterungen .bmp und .jpg
Alle Dateien *.*	Alle Dateien	sämtliche Dateien

Ein Filter, der einige Standardbildformate für unsere **PictureBox** selektiert, könnte dann so aussehen:

Grafikdateien|*.bmp;*.gif;*.jpg;*.png;*.tif;*.wmf

Damit werden alle Dateien mit den Erweiterungen .bmp, .gif, .jpg, .png, .tif und .wmf aufgelistet.

Bitte beachten Sie:

Sie müssen beim Festlegen des Filters sehr sorgfältig arbeiten. Es erfolgen keinerlei Prüfungen, ob der Filter gültig ist. Wenn Sie sich zum Beispiel vertippen und statt *.bmp *.bpm eingeben, werden nur Dateien mit der Erweiterung .bpm selektiert. Und solche Dateien wird es nur in seltenen Fällen geben. Überprüfen Sie daher Ihre Filter immer gründlich.



Hinweis:

Sie können auch mehrere Filter definieren. Dazu trennen Sie die einzelnen Filter bei der Eingabe ebenfalls über das Zeichen |. Der Eintrag

BMP-Dateien|*.bmp|Alle Dateien|*.*

für die Eigenschaft **Filter** vereinbart zum Beispiel einen Filter **BMP-Dateien** und einen Filter **Alle Dateien**. Diese Filter werden auch automatisch im Dialog **Öffnen** angezeigt. Welcher Filter zuerst aktiv ist, können Sie über die Eigenschaft **Filter-Index** in der Gruppe **Verhalten** festlegen. Die Nummerierung beginnt dabei mit dem Wert 1.

Das Auswahlfeld für den Filter wird nur dann angezeigt, wenn Sie mindestens einen Filter über die Eigenschaften für den Dialog festlegen. Ohne Filter erscheint auch kein Auswahlfeld.

Legen Sie jetzt bitte einen Filter für unseren Öffnendialog fest. Welche Grafikdateien Sie dabei über den Filter auswählen, ist relativ beliebig. Sie sollten nur darauf achten, dass alle Dateiformate auch von der **PictureBox** angezeigt werden können. Welche Formate das im Detail sind, finden Sie ganz einfach heraus, indem Sie sich zum Beispiel einmal die Import-Filter für den Dialog der Eigenschaft **Image** der **PictureBox** ansehen. Klicken Sie dazu im Dialog **Ressource auswählen** für die Eigenschaft **Image** auf die Schaltfläche **Importieren...**

Testen Sie anschließend die Anwendung und probieren Sie den Filter aus.

So viel zum Öffnendialog.

2.4 Anzeige in einem neuen Formular

Als letzte Funktion für die Einzelanzeige wollen wir jetzt noch die Anzeige in einem neuen Formular programmieren. Diese Anzeige soll das aktuelle Bild in einem eigenen Fenster anzeigen, das so groß wie möglich dargestellt wird und ständig im Vordergrund liegt.

Dazu erstellen wir im ersten Schritt zunächst einmal das neue Formular. Öffnen Sie das Menü Projekt und klicken Sie auf **Windows Form hinzufügen...**

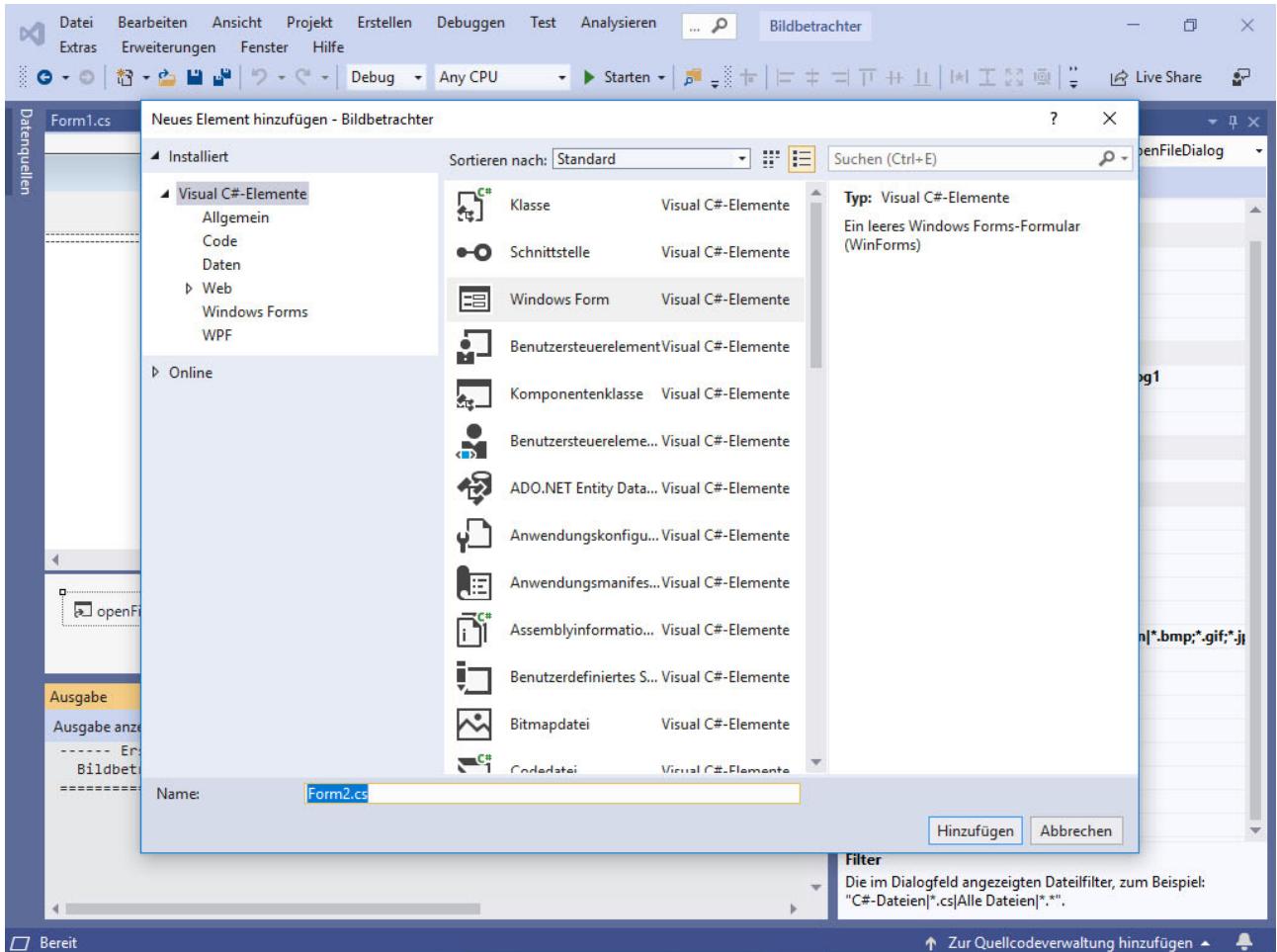


Abb. 2.6: Der Dialog **Neues Element hinzufügen**

Im Dialog **Neues Element hinzufügen** überprüfen Sie, ob der Eintrag **Windows Form** bei den Vorlagen markiert ist, und geben in das Feld **Name:** einen Namen für das Formular ein. Klicken Sie anschließend auf die Schaltfläche **Hinzufügen**. Wir verwenden in unserem Beispiel den Namen **FormMax** für das zweite Formular.

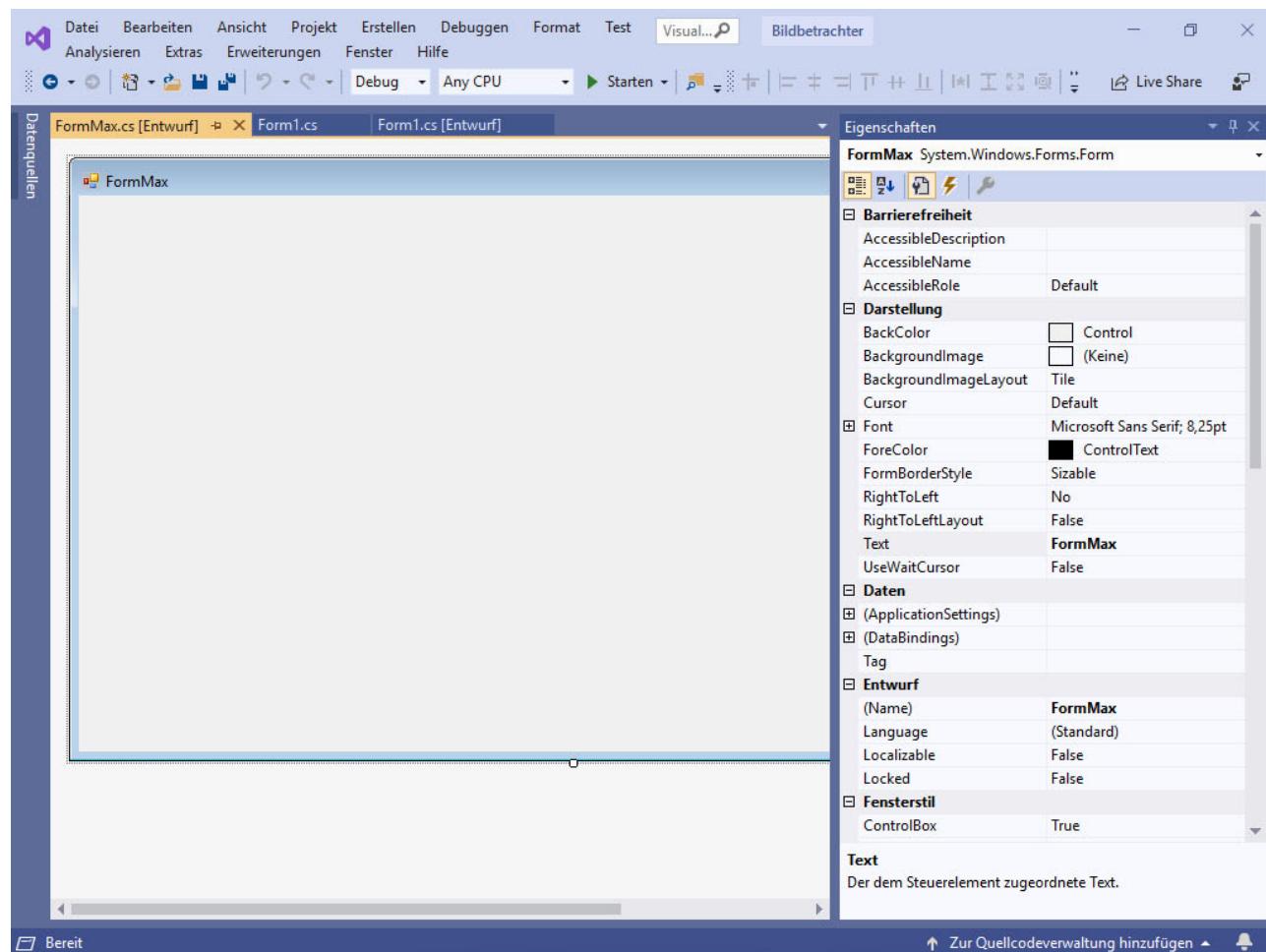


Abb. 2.7: Das neue Formular im Designer

Visual Studio erzeugt das neue Formular und zeigt es im Designer an. Sie können jetzt wie gewohnt die Eigenschaften festlegen. Die Grundeinstellungen finden Sie in der Tab. 2.2.

Tab. 2.2: Grundeinstellungen für das neue Formular

Eigenschaft	Wert
FormBorderStyle	FixedSingle
Text	Große Darstellung
MaximizeBox	False
MinimizeBox	False
WindowState	Maximized

Damit das Fenster ständig im Vordergrund angezeigt wird, setzen Sie außerdem bitte noch die Eigenschaft **TopMost**¹³ in der Gruppe **Fensterstil** auf **True**.

Fügen Sie dann noch ein Steuerelement **PictureBox** ein. Positionieren Sie das Steuerelement so, dass es möglichst groß im Formular erscheint. Setzen Sie die Eigenschaft **SizeMode** für die **PictureBox** auf **Zoom** und verankern Sie das Steuerelement an allen vier Rändern des Formulars. Da wir das neue Formular direkt nach dem Anzeigen maximieren, werden durch diese Einstellungen sowohl die **PictureBox** als auch das Bild im Steuerelement in der maximal möglichen Größe angezeigt.

Damit Sie die **PictureBox** im ersten Formular eindeutig von der **PictureBox** im zweiten Formular unterscheiden können, sollten Sie abschließend für die neue **PictureBox** noch einen anderen Namen benutzen – zum Beispiel **pictureBoxFormMax**.

Hinweis:

Auf eine eigene Schaltfläche zum Schließen des Formulars verzichten wir, da der Anwender das Formular ja auch über das entsprechende Symbol rechts in der Titelleiste schließen kann. Wenn Sie möchten, können Sie ja selbst auch noch eine Schaltfläche zum Schließen einbauen. Besonderheiten gibt es dabei nicht.

Damit wäre der Entwurf des zweiten Formulars fertig. Jetzt müssen wir noch dafür sorgen, dass es auch aus dem ersten Formular angezeigt werden kann. Dazu ändern wir das Ereignis beim Anklicken der Schaltfläche **Anzeigen**. Wir überprüfen hier jetzt zusätzlich, ob das Kontrollkästchen **Neues Fenster** markiert ist. Wenn das der Fall ist, erfolgt die Anzeige im neuen Formular, andernfalls wie gewohnt im ersten Formular.

Speichern Sie bitte die Änderungen im neuen Formular. Wechseln Sie dann in den Quelltext für das erste Formular. Das geht am schnellsten, wenn Sie mit der Maus auf das Register **Form1.cs** oben im Editor klicken. Suchen Sie dann im Quelltext nach der Methode **ButtonAnzeigen_Click()**.

Tipp:

Wenn Sie die Methode nicht finden oder nicht nach ihr suchen möchten, können Sie auch über die Ereignisse im Eigenschaftenfenster in den Quelltext der Methode wechseln. Lassen Sie dazu zunächst wieder das Formular im Designer anzeigen und doppelklicken Sie dann bei den Ereignissen für die Schaltfläche **Anzeigen** auf den Eintrag hinter dem Ereignis **Click**. Sie können aber auch direkt auf die Schaltfläche **Anzeigen** doppelklicken.

Nehmen Sie im Quelltext der Methode anschließend die folgenden Änderungen vor.

```
//wenn ein Eintrag im Eingabefeld steht, laden wir das  
//entsprechende Bild  
if (textBox1.Text != String.Empty)  
{  
    //existiert die Datei überhaupt?  
    if (System.IO.File.Exists(textBox1.Text))  
    {
```

13. Übersetzt bedeutet *topmost* „oberstes“

```

//soll die Datei in einem neuen Fenster angezeigt werden?
if (checkBoxNeuesFenster.Checked == true)
{
    //das neue Formular anzeigen
    FormMax neuesFormular = new FormMax();
    //das Formular modal anzeigen
    neuesFormular.ShowDialog();
}
else
{
    //wenn ja, dann laden und anzeigen
    pictureBox1.Load(textBox1.Text);
}
else
    MessageBox.Show("Die Datei existiert nicht!", "Fehler");
}

```

Code 2.4: Die geänderte Methode ButtonAnzeigen_Click()
(die neuen Teile sind fett markiert)



Bitte beachten Sie:

Wenn Sie für Ihr zweites Formular einen anderen Namen als **FormMax** benutzt haben, müssen Sie den Namen in der Anweisung

```

FormMax neuesFormular = new FormMax();
entsprechend anpassen.

```

Schauen wir uns die Änderungen der Reihe nach an.

Zunächst einmal überprüfen wir mit der neuen Anweisung

```

if (checkBoxNeuesFenster.Checked == true)
ob das Kontrollkästchen Neues Fenster markiert ist. Wenn ja, erzeugen wir mit der An-
weisung

```

```
FormMax neuesFormular = new FormMax();
```

über eine Variable **neuesFormular** eine neue Instanz für das zweite Formular und lassen es mit der Anweisung

```
neuesFormular.ShowDialog();
```

modal anzeigen.



Ein modales Fenster steht immer im Vordergrund und kann auch nicht minimiert werden. Mit der eigentlichen Anwendung kann erst dann weitergearbeitet werden, wenn das modale Fenster wieder geschlossen wird. Typische Beispiele für modale Fenster sind unter anderem die Öffnen- und Speicherndialoge von Windows.

Sie können ein Fenster auch nicht modal anzeigen lassen. Dazu verwenden Sie die Methode **Show()** für das Formular.

Speichern Sie jetzt die Änderungen und testen Sie das Programm. Das neue Fenster sollte korrekt angezeigt werden. Allerdings ist es komplett leer, da die **PictureBox** ja noch keinen Inhalt hat.

Der einfachste Weg wäre es nun, das Bild in der **PictureBox** im zweiten Formular durch eine entsprechende Anweisung in der Methode `ButtonAnzeigen_Click()` über die Methode `Load()` zu laden. Diese Anweisung könnte zum Beispiel so aussehen:

```
//das Bild in die PictureBox im zweiten Formular laden  
neuesFormular.pictureBoxFormMax.Load(textBox1.Text);
```

Hier würde die **PictureBox** `pictureBoxFormMax` im zweiten Formular `neuesFormular` über die Methode `Load()` mit dem Namen des Bildes aus dem Eingabefeld `textBox1` „gefüttert“. Bevor Sie diese Anweisung jetzt in den Quelltext übernehmen: Diese Technik funktioniert nicht. Denn das Steuerelement `pictureBoxFormMax` im zweiten Formular wird mit der Sichtbarkeit `private` verdeckt und steht damit nur dem Formular selbst zur Verfügung. Aus einem anderen Formular können Sie nicht auf das Steuerelement zugreifen. Eine recht rabiate Lösung wäre es, die Sichtbarkeit kurzerhand in `public` zu ändern. Das verletzt aber die DatenkapSELUNG und ist schlechter Stil.

Wir sorgen daher dafür, dass die beiden Formulare miteinander kommunizieren können. Das hört sich kompliziert an, ist aber sehr einfach. Wir erstellen im zweiten Formular eine Methode, die den Namen des anzuseigenden Bildes als Parameter erhält und das Bild über die Methode `Load()` wie gewohnt lädt. Im ersten Formular rufen wir diese Methode dann auf und übergeben dabei den Namen des gewünschten Bildes als Argument.

Führen wir diese Erweiterungen jetzt schrittweise durch.

Wechseln Sie bitte in den Quelltext des zweiten Formulars. Fügen Sie dort unterhalb des Konstruktors die Anweisungen aus dem folgenden Code ein.

Falls Sie den Konstruktor nicht auf Anhieb finden:

Es handelt sich um die erste und einzige Methode in der Klasse. Sie enthält nur die Anweisung `InitializeComponents()`.



Achten Sie beim Eingeben der Anweisungen bitte wieder sorgfältig darauf, dass Sie keine vorhandenen Klammern löschen und die Methode an der richtigen Stelle einfügen – also vor der vorletzten schließenden geschweiften Klammer.

```
public void BildLaden(string bildname)  
{  
    pictureBoxFormMax.Load(bildname);  
}
```

Code 2.5: Die Methode zum Laden des Bildes

Jetzt müssen wir die gerade erstellte Methode noch im anderen Formular aufrufen. Speichern Sie den Quelltext des zweiten Formulars. Wechseln Sie dann wieder in den Quelltext des ersten Formulars und ergänzen Sie in der Methode `buttonAnzeigen_Click()` vor der Anweisung

```
neuesFormular.ShowDialog();
```

die Anweisung

```
neuesFormular.BildLaden(textBox1.Text);
```

Damit rufen wir die Methode `BildLaden()` im zweiten Formular auf und übergeben den Namen des Bildes aus dem Eingabefeld `textBox1` als Argument.



Bitte beachten Sie:

Sie müssen das Bild vor dem Anzeigen des zweiten Formulars laden lassen. Denn sonst wird die Anweisung erst dann ausgeführt, wenn das zweite Formular wieder geschlossen wird, und hat damit keine sichtbare Wirkung. Das liegt daran, dass das zweite Formular modal geöffnet wird und das Programm erst nach dem Schließen des Formulars weiterarbeitet.

Speichern Sie dann noch einmal alle Änderungen und testen Sie das Programm. Jetzt sollte auch die Anzeige im zweiten Formular funktionieren.

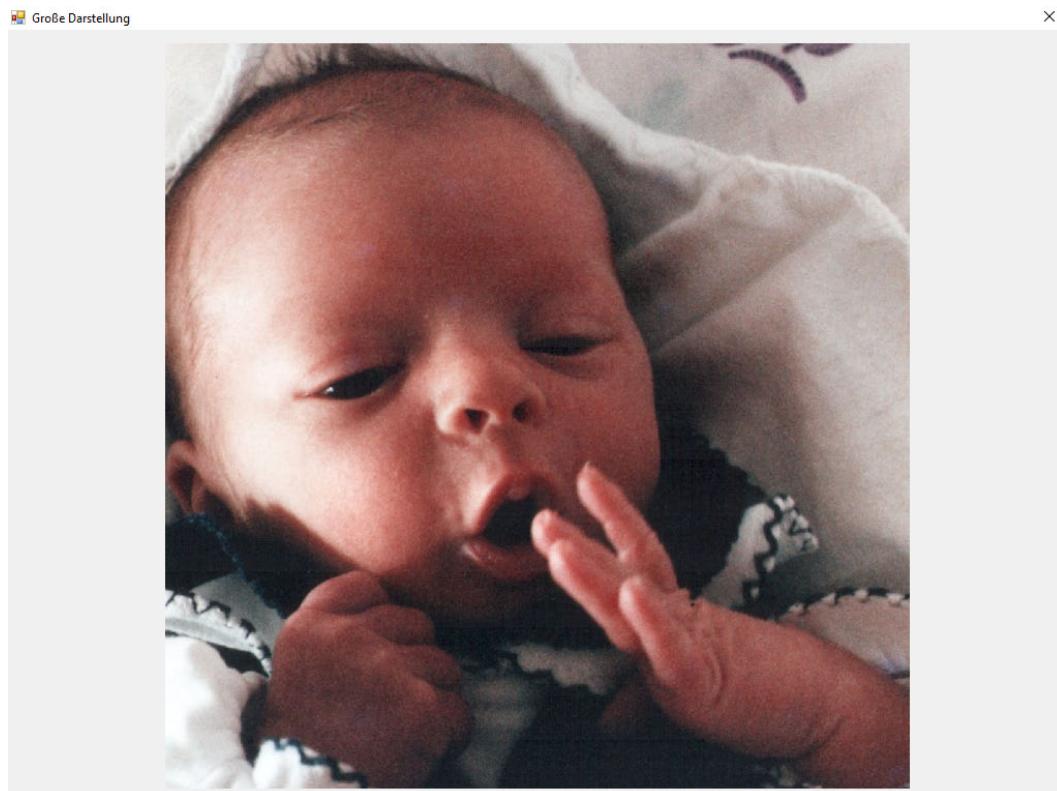


Abb. 2.8: Die Anzeige im zweiten Formular

Hinweis:

Wir reagieren in unserem Programm nicht direkt auf Änderungen an dem Kontrollkästchen **Neues Fenster**. Wenn ein Anwender also das Kontrollkästchen **Neues Fenster** markiert, wird nicht sofort ein neues Fenster geöffnet, sondern erst dann, wenn ein neues Bild geladen wird oder wenn der Anwender noch einmal auf die Schaltfläche **Anzeigen** klickt. Wenn Sie Lust haben, können Sie ja einmal versuchen, die direkte Reaktion auf das Verändern des Kontrollkästchens selbst zu programmieren. Dazu müssen Sie im Ereignis **CheckedChanged** des Kontrollkästchens überprüfen, ob das Kontrollkästchen aktiviert ist, und dann die Methode `ButtonAnzeigen_Click()` mit den beiden Argumenten `sender` und `e` aufrufen. Alles Weitere – zum Beispiel die Prüfung, ob ein Bild ausgewählt wurde – erledigt die Methode `ButtonAnzeigen_Click()`.

So viel zur Einzelanzeige des Bildbetrachters. Im nächsten Kapitel werden wir uns um die Bilderschau kümmern.

Zusammenfassung

Über das Steuerelement **PictureBox** können Sie Bilder in einem Formular anzeigen.

Mit der Eigenschaft **SizeMode** einer **PictureBox** legen Sie fest, wie das Bild innerhalb der **PictureBox** angezeigt wird.

Über das Steuerelement **OpenFileDialog** in der Gruppe **Dialogfelder** der Toolbox fügen Sie einen Öffnendialog ein. Das Steuerelement **OpenFileDialog** ist ein nicht visuelles Steuerelement.

Wenn der Anwender eine Datei in einem Öffnendialog ausgewählt hat, wird das Ereignis **FileOk** für den Dialog ausgelöst. Die ausgewählte Datei erhalten Sie über die Eigenschaft **FileName** des Dialogs.

Über die Eigenschaft **Filter** des Öffnendialogs können Sie die Anzeige der Dateien gezielt festlegen. Der Filter wird in der Form `<angezeigter Name>|<Dateiliste>` angegeben. In der Dateiliste können Sie auch Platzhalter verwenden.

Sie können über die Eigenschaft **Filter** auch mehrere Filter vereinbaren. Die einzelnen Filter müssen dabei ebenfalls durch das Zeichen `|` getrennt werden.

Um ein neues Formular in einem Projekt anzulegen, verwenden Sie die Funktion **Windows Form hinzufügen...** im Menü **Projekt**. Im folgenden Dialog vergeben Sie einen Namen für das Formular.

Mit der Methode `ShowDialog()` zeigen Sie ein Formular modal an. Um ein Formular nicht modal anzuzeigen, benutzen Sie die Methode `Show()`.

Steuerelemente in einem Formular sind in der Regel nur für das Formular selbst sichtbar. Wenn Sie auf ein Steuerelement in einem anderen Formular zugreifen wollen, sollten Sie entsprechende Methoden erstellen.

Aufgaben zur Selbstüberprüfung

- 2.1 Über welche Eigenschaft der **PictureBox** können Sie ein Bild im Eigenschaftenfenster auswählen? Mit welcher Methode einer **PictureBox** können Sie ein Bild zur Laufzeit des Programms anzeigen?

- 2.2 Sie möchten, dass ein Bild in einer **PictureBox** so groß wie möglich angezeigt wird. Die Seitenverhältnisse sollen dabei erhalten bleiben. Welchen Wert setzen Sie für welche Eigenschaft der **PictureBox**? Worauf müssen Sie achten, wenn Sie diese Eigenschaft zur Laufzeit des Programms verändern?

- 2.3 Sie haben einen Öffnendialog mit dem Namen `openFileDialog2` in ein Formular eingefügt. Mit welcher Anweisung lassen Sie den Dialog anzeigen?

- 2.4 Sie haben für das Ereignis **Click** einer Schaltfläche `buttonAuswahl` eine Methode erstellt. Wie können Sie diese Methode selbst im Quelltext einer Methode für das Anklicken einer anderen Schaltfläche aufrufen? Was müssen Sie dabei beachten?

- 2.5 Mit welcher Methode können Sie überprüfen, ob eine Datei existiert?

- 2.6 Sie wollen in einem Öffnendialog nur Dateien mit der Erweiterung **.docx** anzeigen lassen. Der Filter soll den Namen **Word-Dateien** erhalten. Wie muss der vollständige Ausdruck für die Eigenschaft **Filter** heißen?

- 2.7 Sie haben ein neues Formular mit dem Namen `Form2` zu einem Projekt hinzugefügt. Beschreiben Sie alle nötigen Anweisungen, um das neue Formular aus dem bereits vorhandenen Formular über eine Schaltfläche anzeigen zu lassen.

3 Die Bilderschau

In diesem Kapitel setzen wir die Bilderschau um.

Dazu zunächst einmal wieder einige Vorüberlegungen und Vorbereitungen.

3.1 Vorüberlegungen und Vorbereitungen

Unsere Bilderschau soll grundsätzlich so funktionieren:

- Der Anwender wählt in einem Öffnendialog die gewünschten Dateien aus.
- Die Namen der ausgewählten Dateien werden in ein Listenfeld kopiert. Die Bilderschau startet dann entweder beim ersten Eintrag im Listenfeld oder bei dem aktuell markierten Eintrag.
- Jedes Bild soll zehn Sekunden lang in dem Formular für die maximale Größe angezeigt werden, das wir im letzten Kapitel erstellt haben.

Damit benötigen wir im Register **Bilderschau** zunächst einmal folgende Steuerelemente:

- ein Listenfeld für die Anzeige der Dateinamen,
- drei Schaltflächen zum Auswählen der Dateien, zum Starten der Bilderschau und zum Beenden der Anwendung sowie
- einen Öffnendialog.

Außerdem setzen wir später einen **Timer** für die Anzeige der Bilder ein. Damit werden wir uns aber gleich noch im Detail beschäftigen.

Wechseln Sie jetzt bitte im Designer in das zweite Register. Fügen Sie dort die drei Schaltflächen ein. Ordnen Sie sie untereinander rechts oben im Register an und verankern Sie sie rechts und oben. Vergeben Sie dann sprechende Namen an die Schaltflächen und setzen Sie die Texte auf **Auswählen**, **Starten** und **Beenden**. Weisen Sie der Schaltfläche **Beenden** beim Ereignis **Click** die Methode `ButtonBeenden_Click()` zu. Das geht am schnellsten über das Kombinationsfeld neben dem Eintrag für das Ereignis.

Hinweis:

Wir verwenden in unserem Beispiel die Namen `buttonAuswaehlen`, `buttonStarten` und `buttonBeenden1` für die Schaltflächen.

Die Schaltfläche **Beenden** ist doppelt vorhanden, damit wir das **TabControl**-Steuerelement so groß wie möglich darstellen können. Sie könnten die Schaltfläche auch außerhalb der Register ablegen und so unabhängig vom ausgewählten Register ständig anzeigen.

Fügen Sie dann über das Steuerelement **ListBox** ein Listenfeld ein. Legen Sie das Feld links im Formular ab und vergrößern Sie es so, dass es rechts bis zu den Schaltflächen reicht und unten ungefähr in der Mitte des Registers endet. Verankern Sie das Listenfeld anschließend an allen vier Seiten des Formulars.

Das Register sollte nun ungefähr so aussehen:

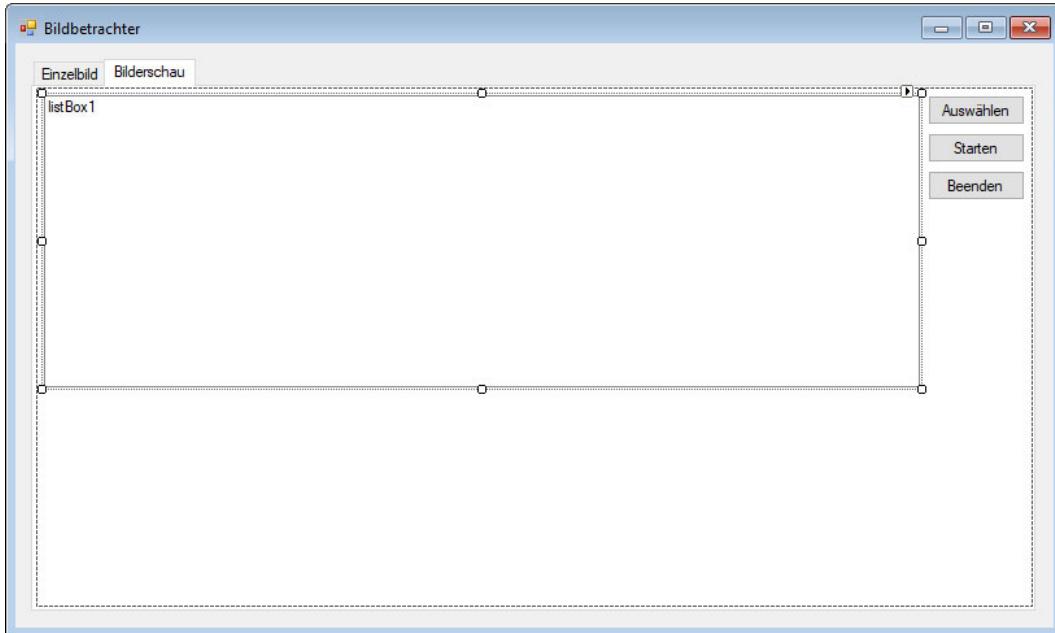


Abb. 3.1: Der Rohbau des Registers **Bilderschau**

Speichern Sie bitte die Änderungen und testen Sie das Programm. Überprüfen Sie dabei vor allem, ob die Steuerelemente korrekt verankert sind und bei Größenänderungen des Formulars mit verändert werden.

Im nächsten Schritt kümmern wir uns jetzt um die Auswahl der Dateien.

3.2 Die Auswahl der Dateien

Dazu erstellen wir einen zweiten Öffnendialog. Da sich dieser Dialog lediglich in einer einzigen Eigenschaft vom Öffnendialog `openFileDialog1` unterscheidet, kopieren wir das bereits vorhandene Steuerelement und passen es dann an.

Markieren Sie das Steuerelement `openFileDialog1` unten im Designer. Kopieren Sie es dann mit der Funktion **Bearbeiten/Kopieren** oder mit der Tastenkombination **Strg + C** in die Zwischenablage und fügen Sie es anschließend mit der Funktion **Bearbeiten/Einfügen** oder mit der Tastenkombination **Strg + V** wieder ein.

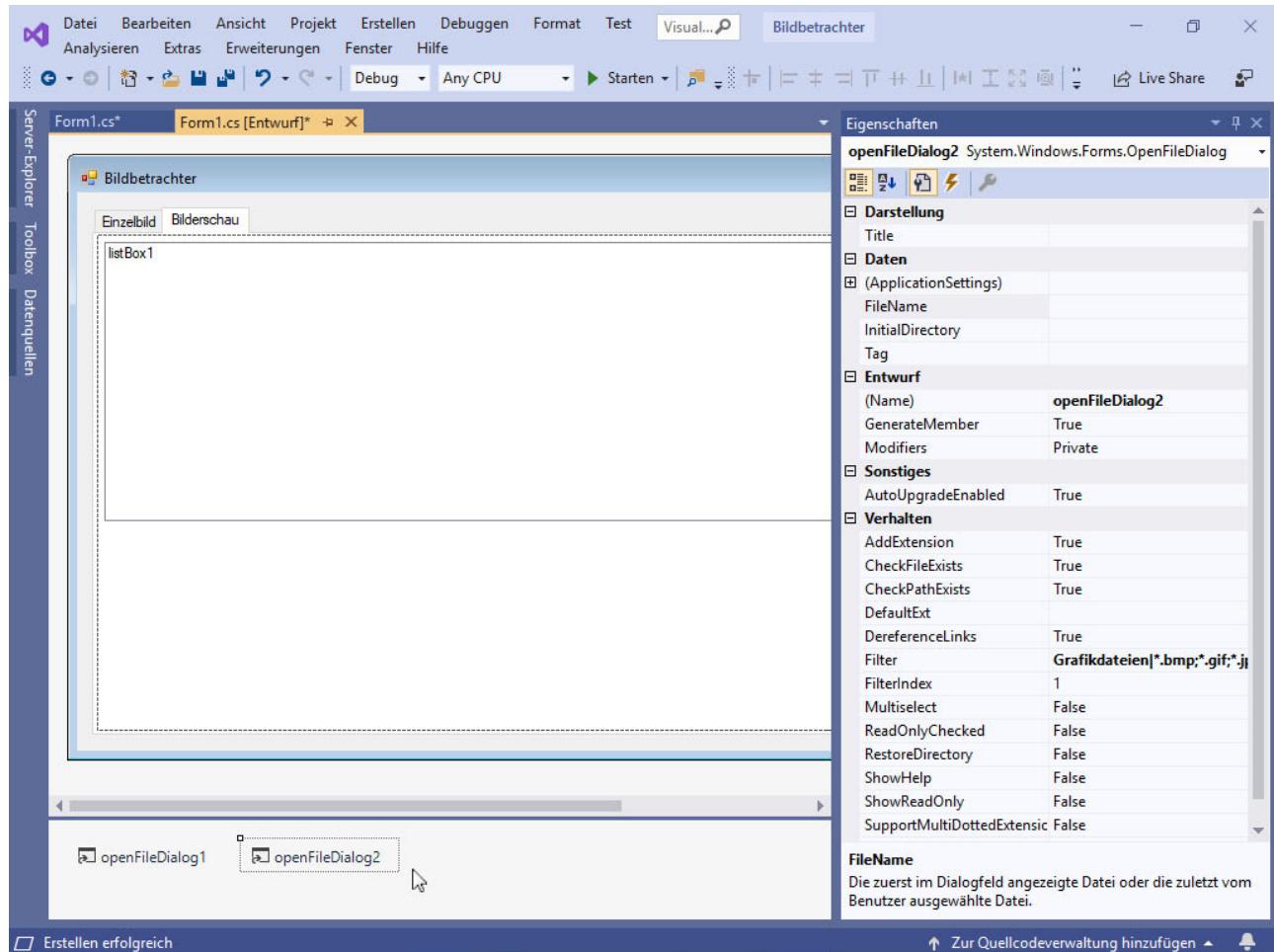


Abb. 3.2: Das kopierte Steuerelement (unten links am Mauszeiger)

Visual Studio erstellt eine Kopie unter dem Namen `openFileDialog2`. Wenn Sie sich einmal die Eigenschaften ansehen, werden Sie feststellen, dass nicht nur das Steuerelement, sondern auch sämtliche Eigenschaften mit kopiert worden sind. Wir müssen jetzt also nur noch dafür sorgen, dass der Anwender in dem neuen Dialog auch mehrere Dateien gleichzeitig markieren kann. Dazu setzen Sie die Eigenschaft `Multiselect`¹⁴ in der Gruppe `Verhalten` auf `True`.

Lassen Sie dann den neuen Dialog beim Anklicken der Schaltfläche **Auswählen** anzeigen. Die entsprechende Anweisung kennen Sie ja bereits vom Anzeigen des ersten Dialogs. Die komplette Methode `ButtonAuswaehlen_Click()` muss also so aussehen:

```
private void ButtonAuswaehlen_Click(object sender,
EventArgs e)
{
    openFileDialog2.ShowDialog();
}
```

Code 3.1: Die vollständige Methode `ButtonAuswaehlen_Click()`

14. `Multiselect` lässt sich frei mit „mehrfache Auswahl“ übersetzen.

Tipp:

Sie können auch den Titel für den Öffnendialog verändern – zum Beispiel in **Auswahl für die Bilderschau**. Die entsprechende Eigenschaft des Dialogs heißt **Title** und befindet sich in der Gruppe **Darstellung** ganz oben im Eigenschaftenfenster.

Jetzt müssen wir noch dafür sorgen, dass die Namen der ausgewählten Dateien aus dem Öffnendialog in unser Listenfeld kopiert werden. Da im Öffnendialog für die Bilderschau auch mehrere Dateien markiert sein können, geht das nicht mehr ganz so einfach wie beim Öffnendialog für die Einzelbildanzeige.

Zunächst einmal beschaffen wir uns über die Eigenschaft **FileNames** des Dialogs die Namen der ausgewählten Dateien. Diese Namen werden nicht einzeln geliefert, sondern in einem Array vom Typ `string`. Dieses Array verarbeiten wir mit einer `foreach`-Schleife und fügen dabei jedes einzelne Element über die Methode `Items.Add()` der **ListBox** am Ende des Listenfelds ein. Die ganze Verarbeitung erfolgt im Ereignis **FileOk** des zweiten Öffnendialogs. Die entsprechenden Anweisungen sehen so aus:

```
//ein neues Array vom Typ string anlegen und die  
//markierten Namen kopieren  
string[] dateien = openFileDialog2.FileNames;  
//die Namen in das Listenfeld kopieren  
foreach (string datei in dateien)  
    listBox1.Items.Add(datei);
```

Code 3.2: Die Anweisungen für das Ereignis FileOk des zweiten Öffnendialogs

Mit der ersten Anweisung legen wir ein Array vom Typ `string` an und weisen diesem Array die Liste mit den ausgewählten Namen zu. Danach kopieren wir dann jeden Eintrag aus der Liste mit der Methode `Items.Add()` in das Listenfeld `listBox1`.

Hinweis:

Sie können auf das Array `dateien` auch verzichten und die Eigenschaft `openFileDialog2.FileNames` direkt in der Schleife verarbeiten. Dann wird der Quelltext etwas kompakter, aber auch schwieriger zu lesen. Welche Variante Sie benutzen, ist Geschmackssache.

Die Eigenschaften **FileName** und **FileNames** unterscheiden sich nur durch das angehängte `s` und können schnell verwechselt werden. Die Eigenschaft **FileName** liefert Ihnen nur einen ausgewählten Namen aus dem Dialog als Zeichenkette. Die Eigenschaft **FileNames** dagegen liefert Ihnen alle ausgewählten Namen als Array vom Typ `string`. Den Unterschied können Sie sich ganz einfach an dem `s` merken: `Name` ist der Singular und `Names` der Plural.

Übernehmen Sie jetzt bitte die Anweisungen aus dem vorigen Code und testen Sie das Programm. Um mehrere Dateien in dem Dialog auswählen zu können, markieren Sie zunächst wie gewohnt die erste Datei und halten dann beim Anklicken der weiteren Dateien die Taste **Strg** gedrückt.

Das Listenfeld mit mehreren Einträgen könnte nach der Auswahl so aussehen:

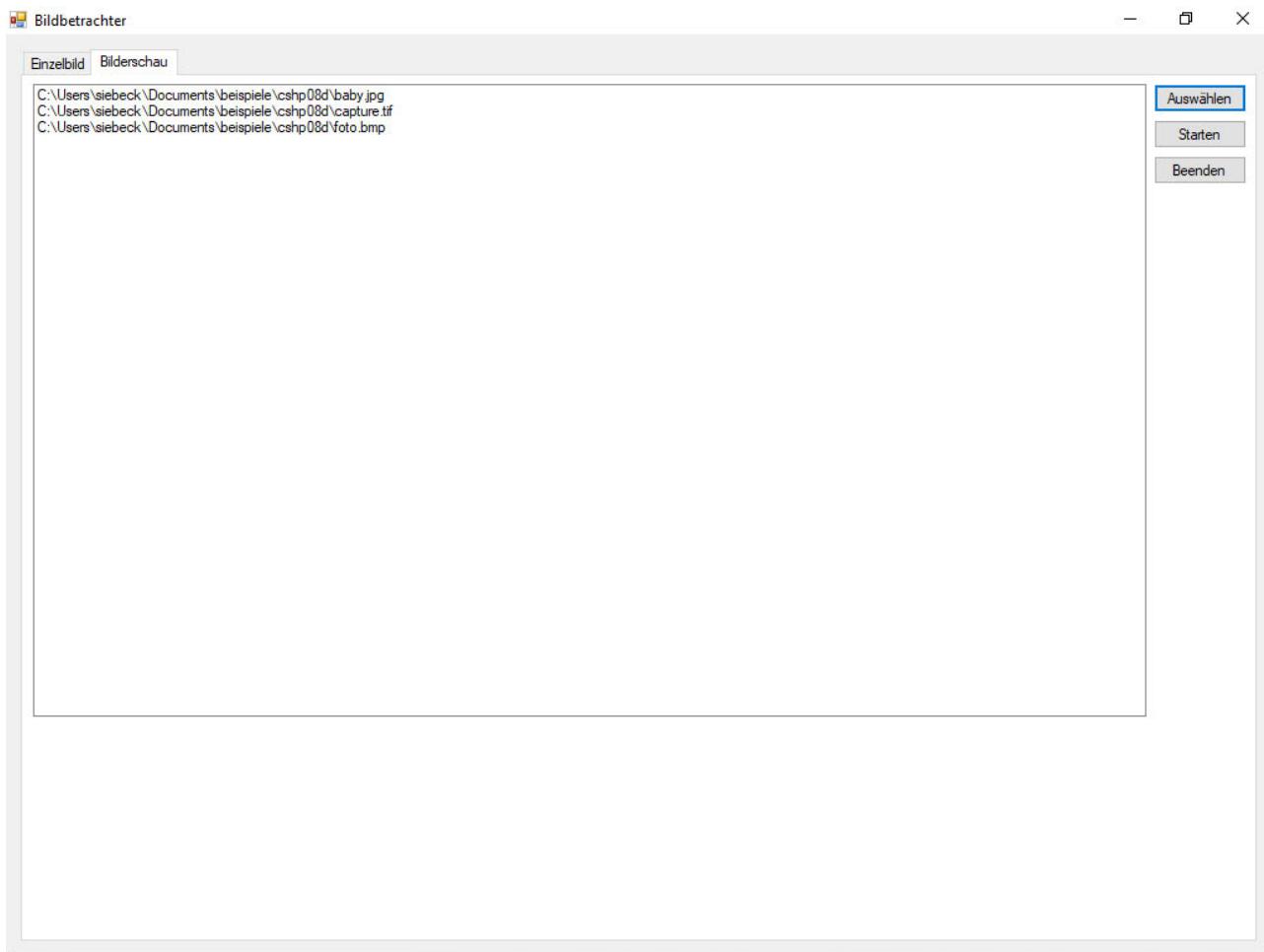


Abb. 3.3: Das Listenfeld mit einigen Einträgen

Hinweis:

Im Listenfeld wird immer der vollständige Pfad angezeigt. Dieser Pfad wird bei Ihnen anders aussehen.

Nachdem die Auswahl jetzt funktioniert, kommt der kniffligste Teil der Bilderschau – die zeitgesteuerte Anzeige der Bilder.

3.3 Die Anzeige der Bilder

Dazu verwenden wir ein Steuerelement **Timer**. Dieses Steuerelement löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis **Tick** aus. Da Sie für dieses Ereignis – genau wie für andere Ereignisse auch – eine Methode hinterlegen können, lassen sich mit einem Timer Anweisungen zeitgesteuert verarbeiten.



Bitte beachten Sie:

Mit einem Timer werden keine Wartezeiten festgelegt. Ein Timer sorgt dafür, dass in einem bestimmten Abstand immer wieder dieselbe Methode ausgeführt wird.

Fügen Sie jetzt bitte einen Timer in das Formular ein. Sie finden das Steuerelement in der Toolbox unten in der Gruppe **Komponenten**. Da es sich um ein nicht visuelles Steuerelement handelt, wird es unten im grauen Bereich neben den Öffnendialogen angezeigt.

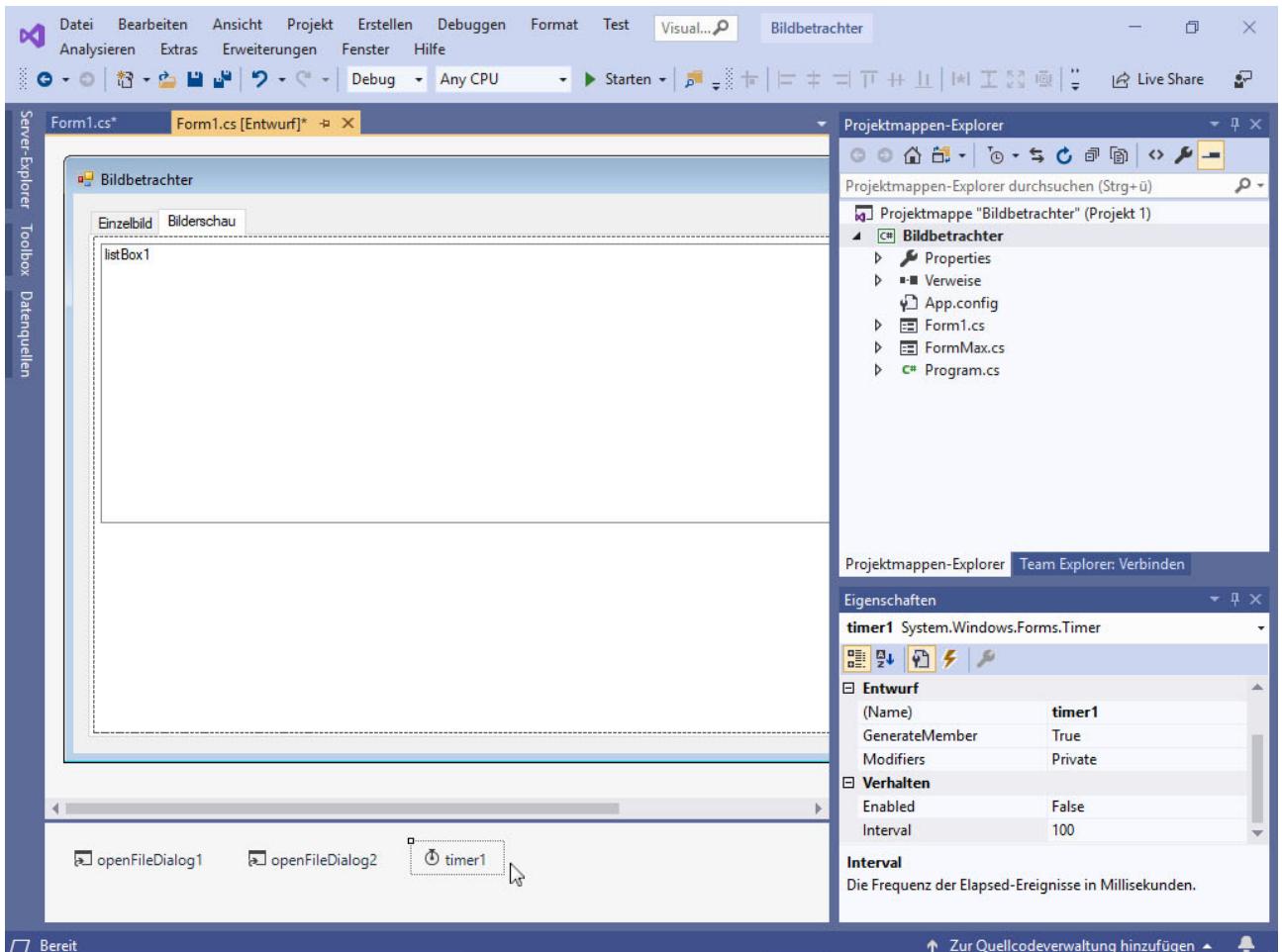


Abb. 3.4: Der eingefügte Timer
(unten links am Mauszeiger)

Setzen Sie dann die Eigenschaft **Interval** auf 10000. Dadurch wird der Timer alle 10 Sekunden ausgelöst.

Bitte beachten Sie:

Die Angabe des Timer-Intervalls erfolgt nicht in Sekunden, sondern in Millisekunden. Der Standardwert 100 für das Intervall löst den Timer also jede Zehntelsekunde aus.



Da die Eigenschaft **Enabled** des Timers auf `False` steht, ist der Timer direkt nach dem Einfügen deaktiviert. Er hat also zunächst einmal keine Wirkung. Wir werden ihn später durch eine eigene Anweisung im Quelltext starten. Dazu müssen wir aber zunächst einmal ein paar Vorbereitungen treffen.

Im ersten Schritt vereinbaren wir eine Variable für das Formular **FormMax**. Da wir diese Variable in mehreren unterschiedlichen Methoden benötigen, reicht eine lokale Vereinbarung nicht mehr aus. Wir müssen die Variable als Feld der Klasse `Form1` festlegen.

Wechseln Sie bitte in den Quelltext der Datei `Form1.cs`. Ergänzen Sie dann oberhalb des Konstruktors der Klasse die folgende Zeile:

```
private FormMax fensterBilderschau;
```

Beim Anklicken der Schaltfläche **Starten** im Register **Bilderschau** überprüfen wir jetzt zunächst einmal, ob sich überhaupt Einträge im Listenfeld befinden. Wenn das nicht der Fall ist, können wir die Methode direkt wieder verlassen. Andernfalls kontrollieren wir, ob eine Markierung in der Liste gesetzt ist, und markieren gegebenenfalls selbst den ersten Eintrag. Anschließend erzeugen wir über das Feld `fensterBilderschau` eine neue Instanz für das Formular **FormMax**, laden das Bild, dessen Eintrag aktuell markiert ist, und lassen das Formular dann anzeigen. Die entsprechenden Anweisungen finden Sie im folgenden Code 3.3.

```
//befinden sich überhaupt Einträge in der Liste?  
//wenn nicht, erzeugen wir eine Meldung und verlassen  
//die Methode wieder  
if (listBox1.Items.Count == 0)  
{  
    //bitte in einer Zeile eingeben  
    MessageBox.Show("Sie müssen erst Dateien  
auswählen!", "Fehler");  
    return;  
}  
//ist kein Eintrag im Listenfeld markiert?  
//dann den ersten Eintrag markieren  
if (listBox1.SelectedIndex == -1)  
    listBox1.SelectedIndex = 0;  
//ein neues Fenster für die Bilderschau erzeugen  
fensterBilderschau = new FormMax();  
//das erste Bild laden  
fensterBilderschau.BildLaden(listBox1.SelectedItem.ToString());  
//den Titel des Formulars auf Bilderschau setzen  
fensterBilderschau.Text = "Bilderschau";  
//das Formular anzeigen  
fensterBilderschau.Show();
```

Code 3.3: Das Anzeigen des ersten Bildes

Schauen wir uns die einzelnen Anweisungen noch einmal der Reihe nach an.

Mit der `if`-Abfrage

```
if (listBox1.Items.Count == 0)  
{  
    MessageBox.Show("Sie müssen erst Dateien auswählen!",  
    "Fehler");  
    return;  
}
```

überprüfen wir, wie viele Elemente sich im Listenfeld befinden. Dazu fragen wir den Wert der Eigenschaft `Items.Count` ab. Wenn er 0 ist – die Liste also leer ist –, erzeugen wir eine Meldung und verlassen die Methode über `return` wieder.

Danach überprüfen wir, ob die Eigenschaft `SelectedIndex` des Listenfelds den Wert `-1` hat und setzen gegebenenfalls die Markierung auf den ersten Eintrag. Das übernehmen die Anweisungen

```
if (listBox1.SelectedIndex == -1)  
    listBox1.SelectedIndex = 0;
```

Zur Erinnerung:

Der Wert `-1` für die Eigenschaft `SelectedIndex` bei einem Listen- oder Kombinationsfeld zeigt Ihnen, dass kein Eintrag ausgewählt ist.



Die Nummerierung der Einträge beginnt bei `0` und nicht bei `1`.

Anschließend erzeugen wir eine neue Instanz für das zweite Formular und laden das aktuell markierte Bild. Den Namen des Bildes beschaffen wir uns dabei mit dem Ausdruck `listBox1.SelectedItem.ToString()`.

Danach setzen wir den Text in der Titelleiste des zweiten Formulars auf **Bilderschau**. Das erfolgt durch die Anweisung

```
fensterBilderschau.Text = "Bilderschau";
```

Zu guter Letzt zeigen wir dann das Formular mit dem Bild über

```
fensterBilderschau.Show();
```

an. Bitte beachten Sie dabei, dass wir das Formular jetzt nicht mehr modal anzeigen lassen dürfen. Andernfalls wird nämlich die Verarbeitung im Programm so lange unterbrochen, bis das Fenster wieder geschlossen wird. Damit käme dann auch unser Timer nicht wie gewünscht zum Zuge.

Übernehmen Sie jetzt die Anweisungen aus dem vorigen Code für das Ereignis **Click** der Schaltfläche **Starten**. Testen Sie dann das Programm. Das erste Bild der Liste sollte schon korrekt angezeigt werden.

**Ein wichtiger Hinweis:**

Die Überprüfungen auf Einträge in der Liste und auf die Markierung in der Liste, die wir am Anfang der Methode `ButtonStarten_Click()` durchführen, mögen Ihnen vielleicht überflüssig erscheinen, sind aber für die Betriebssicherheit des Programms sehr wichtig. Denn Sie können keineswegs davon ausgehen, dass der Anwender auch tatsächlich Dateien ausgewählt und einen Eintrag in der Liste markiert hat. Und ohne ausgewählte Dateien beziehungsweise ohne markierten Eintrag läuft das Programm beim Laden der Bilddatei auf einen Fehler, da es ja gar nichts verarbeiten kann.

Anders als bei Programmen, die strikt sequenziell abgearbeitet werden, können Sie bei Programmen mit einer grafischen Oberfläche in vielen Fällen nicht vorhersehen, welchen Schritt ein Anwender als Nächstes ausführen wird. Sie müssen also selbst dafür sorgen, dass das Programm immer in einem stabilen Zustand bleibt – zum Beispiel eben durch umfangreiche Abfragen. Programmieren Sie dabei lieber eine Abfrage zu viel als eine zu wenig.

Welche Wirkung die fehlenden Abfragen haben, können Sie selbst ausprobieren. Kommentieren Sie die Zeilen aus und testen Sie das Programm dann einmal mit einer leeren Liste oder ohne Markierung im Listenfeld. Sie werden sehen, in beiden Fällen wird eine Ausnahme ausgelöst.

Im Ereignis **Tick** des Timers sorgen wir nun noch dafür, dass das nächste Bild aus der Liste geladen wird. Dazu überprüfen wir, ob überhaupt noch ein weiterer Eintrag in der Liste folgt, und verschieben die Markierung im Listenfeld gegebenenfalls auf diesen weiteren Eintrag. Anschließend laden wir das Bild in das Formular. Das Formular haben wir ja in der Methode `ButtonStarten_Click()` geöffnet, aber nicht wieder geschlossen. Es wird also immer noch angezeigt.

Wenn wir das Ende der Liste erreicht haben, räumen wir auf. Wir löschen die Markierung im Listenfeld wieder, schließen das Formular und halten den Timer über die Methode `Stop()` an. Die entsprechenden Anweisungen für das Ereignis **Tick** des Timers sehen dann so aus:

```
//ist der letzte Eintrag in der Liste noch nicht
//erreicht?
if (listBox1.SelectedIndex < listBox1.Items.Count - 1)
{
    //den nächsten Eintrag markieren
    listBox1.SelectedIndex++;
    //und das Bild laden
    //bitte in einer Zeile eingeben
    fensterBilderschau.BildLaden(listBox1.SelectedItem.
        ToString());
}
//beim letzten Bild wieder aufräumen
else
{
    //die Markierungen löschen
    listBox1.SelectedIndex = -1;
    //den Timer anhalten
    timer1.Stop();
    //das Fenster Bilderschau schließen
    fensterBilderschau.Close();
}
```

Code 3.4: Die Anweisungen für das Ereignis **Tick** des Timers

Bitte beachten Sie:

Die Eigenschaft `Items.Count` liefert Ihnen die echte Anzahl der Elemente in einem Listen- oder Kombinationsfeld. Beim Index beginnt die Nummerierung aber mit dem Wert 0. Wenn Sie feststellen wollen, ob das letzte Element erreicht ist, müssen Sie beim Vergleich des Index mit der Anzahl daher von der echten Anzahl noch 1 abziehen.



Jetzt fehlt nur noch eins, damit das Programm funktioniert: das Starten des Timers. Es soll direkt nach dem Anzeigen des ersten Bildes erfolgen. Ergänzen Sie dazu bitte die Anweisung

```
timer1.Start();
```

ganz am Ende der Methode `ButtonStarten_Click()`. Speichern Sie dann alle Änderungen und testen Sie das Programm. Wenn Sie alles richtig gemacht haben, sollte alle 10 Sekunden ein neues Bild angezeigt werden. Gestartet wird die Bilderschau dabei mit der Datei, die im Listenfeld markiert ist.

Hinweis:

Die Methode `Start()` startet lediglich den Timer, löst aber nicht sofort das Ereignis Tick für den Timer aus. Das Ereignis wird erst dann zum ersten Mal ausgelöst, wenn die angegebene Zeit abgelaufen ist. Daher bleibt das erste Bild, das wir ja per Hand geladen haben, zunächst einmal 10 Sekunden stehen.

Eigentlich könnten Sie den Timer auch anhalten, wenn das letzte Bild der Liste angezeigt wird. Denn ein weiteres Bild zur Anzeige gibt es dann ja nicht mehr. Allerdings müssten Sie in diesem Fall auch das Aufräumen und vor allem das Anhalten des Timers an geeigneter Stelle selbst erledigen. Mit unserer Technik sorgt der Timer quasi selbst dafür, dass er angehalten wird, wenn er seine Arbeit erledigt hat.

Damit ist der Bildbetrachter mit Einzelbildanzeige und Bilderschau im Wesentlichen fertiggestellt. Im nächsten Kapitel werden wir noch an der einen oder anderen Stelle ein wenig Feintuning vornehmen.

Zusammenfassung

Sie können nahezu alle vorhandenen Steuerelemente in einem Formular über die Zwischenablage kopieren und wieder einfügen. Dabei werden auch die gesetzten Eigenschaften des Steuerelements mitkopiert.

Damit ein Anwender in einem Öffnendialog mehrere Dateien auswählen kann, müssen Sie die Eigenschaft **Multiselect** des Dialogs auf `True` setzen. Die ausgewählten Dateien werden in der Eigenschaft **FileNames** zurückgeliefert.

Mit der Methode `Items.Add()` fügen Sie einen Eintrag in ein Listenfeld ein.

Über das Steuerelement **Timer** können Sie in frei wählbaren Abständen eine Methode regelmäßig ausführen lassen. Die Ausführung des Timers steuern Sie über die Methoden `Start()` und `Stop()`.

Beim Starten eines Timers wird das Ereignis **Tick** nicht direkt ausgelöst. Es tritt erst dann ein, wenn die eingestellte Zeit verstrichen ist.

Die Eigenschaft `Items.Count` liefert Ihnen die echte Anzahl der Elemente in einem Listen- oder Kombinationsfeld.

Aufgaben zur Selbstüberprüfung

- 3.1 Was unterscheidet die Eigenschaften `FileName` und `FileNames` des Steuerelements `OpenFileDialog`?

- 3.2 Sie wollen alle Dateien, die ein Anwender in einem Öffnendialog ausgewählt hat, in einer Schleife verarbeiten. Wie könnte solch eine Schleife aussehen? Benutzen Sie bitte eine möglichst einfache Form. Speichern Sie die Namen der ausgewählten Dateien vor dem Zugriff in der Schleife zwischen.

- 3.3 Über welche Eigenschaft können Sie den Text in der Titelleiste eines Öffnendialogs verändern?

- 3.4 Wie heißt das Ereignis, das von einem Timer ausgelöst wird?

- 3.5 Sie haben bei einem Timer die Eigenschaft **Interval** auf 2 000 gesetzt. In welchen Abständen wird der Timer ausgelöst? Geben Sie den Wert bitte in Sekunden an.

- 3.6 Sie lassen ein Formular modal anzeigen. In einem anderen Formular läuft ein Timer. Wird dieser Timer ausgelöst, solange das modale Fenster geöffnet ist? Begründen Sie kurz Ihre Antwort.

4 Der Feinschliff für den Bildbetrachter

In diesem Kapitel verpassen wir unserem Bildbetrachter den letzten Schliff.

4.1 Korrektur von Schönheitsfehlern

Beginnen wir mit der Korrektur einiger kleiner Schönheitsfehler. So bleiben ja zum Beispiel die Einträge im Listenfeld stehen, wenn der Anwender die Funktion zum Auswählen mehrere Male hintereinander startet. Und das führt dazu, dass die Liste immer länger wird.

Das Problem lässt sich ganz einfach beheben, indem Sie das Listenfeld vor dem Eintragen der ausgewählten Dateien in der Methode `OpenFileDialog2_FileOk()` leeren. Dazu verwenden Sie die Methode `Items.Clear()`. Die vollständige Methode `OpenFileDialog2_FileOk()` würde dann so aussehen:

```
private void OpenFileDialog2_FileOk(object sender,
CancelEventArgs e)
{
    //ein neues Array vom Typ string anlegen und die
    //markierten Namen kopieren
    string[] dateien = openFileDialog2.FileNames;
    //wenn sich noch Einträge in dem Listenfeld befinden,
    //löschen wir sie
    if (listBox1.Items.Count != 0)
        listBox1.Items.Clear();
    //die Namen in das Listenfeld kopieren
    foreach (string datei in dateien)
        listBox1.Items.Add(datei);
}
```

Code 4.1: Die Methode `OpenFileDialog2_FileOk()` mit dem Leeren des Listenfelds
(die neuen Anweisungen sind fett markiert)

Ob Sie vor dem Löschen überprüfen, ob sich Einträge in der Liste befinden oder nicht, ist Geschmackssache. Denn auch bei einem leeren Listenfeld können Sie die Methode `Items.Clear()` ohne Risiko aufrufen. Allerdings wird die Methode dann natürlich auch ausgeführt, wenn es eigentlich nicht erforderlich ist.

Auf die Geschwindigkeit der Ausführung haben die Variante mit der Abfrage und die Variante ohne die Abfrage übrigens keine besondere Auswirkung. Im ersten Fall wird das Löschen zwar nur dann ausgeführt, wenn es wirklich nötig ist. Allerdings beansprucht auch die Überprüfung der Listeneinträge Rechenzeit. Daher spielt es für die Ausführung kaum eine Rolle, ob Sie vor dem Löschen kontrollieren, ob das Löschen erforderlich ist. Die Variante mit der Prüfung ist allerdings – vom Programmiertechnischen her gesehen – sauberer.

Das nächste Problem betrifft den Ablauf der Bilderschau. Denn ein Anwender kann das Formular, in dem wir die Bilder anzeigen, über das Symbol ganz rechts in der Titelleiste jederzeit schließen. Dabei wird die Bilderschau allerdings nicht abgebrochen, sondern läuft weiter – eben ohne Anzeige der Bilder. Dafür verschiebt sich dann – wie von Geisterhand – die Markierung in dem Listenfeld alle 10 Sekunden.

Was auf den ersten Blick nur ein Schönheitsfehler ist, kann durchaus auch zum Programmabsturz führen. Denn beim Schließen eines Formulars werden auch sämtliche Ressourcen des Formulars für die Garbage Collection zur Freigabe markiert. Und damit verweist die Instanz, über die wir auf das Formular zugreifen, spätestens dann nicht mehr auf das Formular, wenn die nächste Garbage Collection durchgeführt wird. Wann das genau der Fall ist, lässt sich allerdings nicht vorhersagen.

Um mögliche Probleme von vornherein zu vermeiden, sorgen wir deshalb dafür, dass das Fenster nicht mehr über das Symbol geschlossen werden kann. Dazu setzen wir die Eigenschaft **ControlBox**¹⁵ des Formulars auf `false`. Damit werden sowohl das Systemmenü als auch das Schließensymbol ausgeblendet.

Die entsprechende Erweiterung können Sie zum Beispiel in der Methode `ButtonStarten_Click()` nach dem Setzen des Fenstertitels vornehmen.

```
...
//den Titel des Formulars auf Bilderschau setzen
fensterBilderschau.Text = "Bilderschau";
//das Systemmenü und das Schließensymbol ausblenden
fensterBilderschau.ControlBox = false;
//das Formular anzeigen
fensterBilderschau.Show();
...
```

Code 4.2: Die erweiterte Methode `ButtonStarten_Click()`
(abgedruckt sind nur Teile der Methode)

Da die Eigenschaft **ControlBox** in der Standardeinstellung im Eigenschaftenfenster auf `True` gesetzt wird, müssen Sie sich um das Einblenden des Systemmenüs und des Schließensymbols nicht weiter kümmern. Denn die geänderten Einstellungen gelten ausschließlich für die Instanz des Formulars, die wir für die Bilderschau benutzen. Bei der Einzelbildanzeige erzeugen wir ja neue Instanzen für das Formular, die wieder die Eigenschaften aus dem Eigenschaftenfenster verwenden.

Im nächsten Schritt sorgen wir dafür, dass unsere beiden Register auch dann auf die Eingabetaste reagieren, wenn keine der Schaltflächen den Fokus hat. Im Register **Einzelbild** soll beim Drücken der Eingabetaste die Funktion **Anzeigen** gestartet werden und im Register **Bilderschau** die Funktion **Auswählen**. Die entsprechenden Zuweisungen setzen wir über die Eigenschaft **AcceptButton** für das Formular.

Damit die beiden Register anders auf die Eingabetaste reagieren, müssen wir dieser Eigenschaft bei der Anzeige des jeweiligen Registers einen anderen Wert zuweisen. Das geht am einfachsten über das Ereignis **Enter** in der Gruppe **Fokus** der jeweiligen Registerkarte. Dieses Ereignis tritt ein, wenn ein Steuerelement zum aktiven Steuerelement wird – zum Beispiel beim Wechseln zwischen den Registern. Die vollständigen Methoden für die beiden Ereignisse finden Sie im Code 4.3:

```
private void TabPageEinzel_Enter(object sender, EventArgs e)
{
    AcceptButton = buttonAnzeigen;
}
```

15. **ControlBox** lässt sich mit „Kontrollfeld“ übersetzen.

```
private void TabPageSchau_Enter(object sender, EventArgs e)
{
    AcceptButton = buttonStarten;
}
```

Code 4.3: Die Methoden TabPageEinzel_Enter() und TabPageSchau_Enter()

Übernehmen Sie die Anweisungen für die Ereignisse jetzt bitte in Ihr Projekt. Achten Sie dabei sorgfältig darauf, dass Sie die Ereignisse tatsächlich für die beiden Register festlegen. Denn das Ereignis **Enter** gibt es für nahezu jedes Steuerelement, das den Fokus erhalten kann. Wenn Sie ganz sicher gehen wollen, wählen Sie die beiden Register über das Kombinationsfeld oben im Eigenschaftenfenster aus beziehungsweise kontrollieren Sie vor dem Eingeben der Anweisungen noch einmal über das Kombinationsfeld, welches Steuerelement Sie ausgewählt haben.

Hinweis:

Bitte denken Sie daran, dass die Eigenschaft **AcceptButton** nur dann Wirkung zeigt, wenn der Fokus nicht auf einem Steuerelement steht, das selbst auch auf die Eingabetaste reagiert. Wenn beispielsweise eine Schaltfläche den Fokus hat, wird beim Drücken der Eingabetaste immer das Anklicken der Schaltfläche simuliert. Die Eigenschaft **AcceptButton** wird dann schlicht und einfach ignoriert.

Im letzten Schritt sollten Sie jetzt noch die Aktivierreihenfolge prüfen und gegebenenfalls so korrigieren, dass die Steuerelemente in beiden Registern von oben links nach unten rechts durchlaufen werden. Dazu wechseln Sie in den Entwurf für das Formular und rufen die Funktion **Ansicht/Aktivierreihenfolge** auf. Klicken Sie dann in der gewünschten Reihenfolge nacheinander auf die Elemente. Bitte beachten Sie dabei, dass Sie die Aktivierreihenfolge für die beiden Register getrennt festlegen müssen.

4.2 Soundwiedergabe

Als „I-Tüpfelchen“ wollen wir unseren Bildbetrachter abschließend noch mit einem kleinen akustischen Feedback versehen. Immer wenn in der Bilderschau ein neues Bild angezeigt wird, soll gleichzeitig auch ein Sound abgespielt werden.

Dazu greifen wir über die Methode

System.Media.SystemSounds.<Soundname>.Play()¹⁶

auf die Systemsounds von Windows zu, die bei den Soundschemas hinterlegt sind. Den Platzhalter <Soundname> in der Anweisung ersetzen Sie dabei durch den englischen Namen des gewünschten Sounds. Die möglichen Werte und den dazugehörigen Sound beziehungsweise das dazugehörige Programmereignis finden Sie in der Tab. 4.1.

16. Play bedeutet übersetzt „spiele“.

Tab. 4.1: Die Systemsounds von Windows

Name	abgespielter Sound/Programmereignis
Asterisk	Sternchen
Beep	Standardton Warnsignal
Exclamation	Hinweis
Hand	Kritischer Abbruch
Question	Frage

Bitte beachten Sie:

Welche Sounds tatsächlich abgespielt werden, hängt von den Einstellungen ab, die Sie in Windows bei den Sundeigenschaften vorgenommen haben. Wenn Sie einem Ereignis keinen Sound zugeordnet haben, geschieht gar nichts.



Neben `System.Media.SystemSounds` gibt es auch noch `System.Media.SystemSound` – also ohne s am Ende. Damit wird aber ein Ereignis beschrieben, das ausgelöst wird, wenn eine Anwendung einen Sound abgespielt hat. Achten Sie daher bei der Eingabe sorgfältig darauf, dass Sie das s am Ende nicht vergessen.

Bevor Sie weiterlesen ...

Versuchen Sie zunächst einmal selbst, die Wiedergabe eines Standard-Windows-Sounds an der passenden Stelle einzubauen. Prüfen Sie vorher zur Sicherheit über die Einstellungen von Windows, ob Sie für das Ereignis überhaupt einen Sound festgelegt haben.

Eine Möglichkeit, beim Bildwechsel einen Sound abzuspielen, besteht zum Beispiel in der Methode `Timer1_Tick()` unmittelbar vor oder nach dem Laden des Bildes.

```
...
if (listBox1.SelectedIndex < listBox1.Items.Count - 1)
{
    //den nächsten Eintrag markieren
    listBox1.SelectedIndex++;
    //und das Bild laden
    fensterBilderschau.BildLaden(listBox1.SelectedItem.
    ToString());
    //einen Sound aus der Systemsteuerung abspielen
    //hier für den Stern
    System.Media.SystemSounds.Asterisk.Play();
}
...
```

Code 4.4: Das Abspielen eines Sounds beim Anzeigen eines neuen Bildes

Neben dem Abspielen von Windows-Sounds können Sie auch beliebige Audio-Dateien im Wave-Format wiedergeben.



Das Wave-Format ist ein weitverbreitetes Standardformat für Audiodateien. Die Dateien erkennen Sie in der Regel an der Erweiterung `.wav`. Übersetzt bedeutet *wave* so viel wie „Welle“.

Dazu erzeugen Sie eine neue Instanz der Klasse `System.Media.SoundPlayer` und weisen dieser Instanz über den Konstruktor den Namen der gewünschten Datei zu. Anschließend können Sie die Datei mit der Methode `Play()` abspielen.

Das hört sich vielleicht ein wenig kompliziert an, ist aber sehr einfach. Denn den Namen der Datei übergeben Sie einfach als Argument beim Erzeugen der Instanz. Das könnte zum Beispiel so aussehen:

```
//bitte in einer Zeile eingeben
System.Media.SoundPlayer player = new
System.Media.SoundPlayer("c:\\\\probe\\\\beppi.wav");
```

Erzeugt wird hier eine Instanz `player` für die Klasse `System.Media.SoundPlayer`. Dabei wird die Datei `beppi.wav` im Ordner `probe` auf dem Laufwerk C: an den Konstruktor übergeben.



Bitte beachten Sie:

Das Trennzeichen `\` muss in Pfadangaben doppelt gesetzt werden. Denn wie Sie bereits wissen, leitet das Zeichen `\` ja eigentlich eine Escape-Sequenz ein. Wenn Sie das Zeichen `\` nicht doppelt in einem Pfad verwenden, erscheint nur dann eine Fehlermeldung beim Übersetzen, wenn Sie hinter dem Zeichen `\` eine ungültige Escape-Sequenz angeben – zum Beispiel den Buchstaben `p`. Beim Ausführen löst das Programm aber in jedem Fall eine Ausnahme aus, da der Pfad nicht korrekt angegeben ist.

Neben der doppelten Angabe des Trennzeichens können Sie aber auch das Zeichen `@` vor die Zeichenkette setzen. Dann werden die Zeichen grundsätzlich **nicht** als Escape-Sequenzen interpretiert. Die Anweisung von oben würde dann so aussehen:

```
System.Media.SoundPlayer player = new
System.Media.SoundPlayer(@"c:\\probe\\\\beppi.wav");
```

Die Wiedergabe der Datei erfolgt dann einfach mit der Anweisung

```
player.Play();
```

Dabei überprüft Visual Studio allerdings nicht, ob die Datei auch tatsächlich vorhanden ist. Sie müssen bei der Pfadangabe also nicht nur darauf achten, dass Sie das Zeichen `\` doppelt verwenden beziehungsweise das Zeichen `@` vor die Zeichenkette setzen, sondern auch sorgfältig prüfen, ob Sie sich beim Pfad nicht möglicherweise vertippt haben. Andernfalls löst das Programm beim Versuch, die Datei abzuspielen, eine Ausnahme aus.

Probieren Sie die Wiedergabe über die Klasse `SoundPlayer` einfach einmal selber aus. Wenn Sie keine eigene Wave-Datei finden, können Sie auch die Datei `hallo.wav` im Ordner `\beispiele\cshp08d` verwenden.

Tipp:

Hinweise zu weiteren Methoden der Klasse `SoundPlayer` – zum Beispiel zum Laden oder zum Stoppen der Wiedergabe – finden Sie in der Hilfe unter dem Suchbegriff **SoundPlayer-Class**.

Damit ist unser Bildbetrachter fertiggestellt. Einige Erweiterungen warten noch als Einsendeaufgabe auf Sie.

Zusammenfassung

Um die Einträge in einem Listen- oder Kombinationsfeld zu löschen, verwenden Sie die Methode `Items.Clear()`.

Das Ereignis **Enter** tritt ein, wenn ein Steuerelement zum aktiven Steuerelement wird. Achten Sie beim Erstellen der Methode für das Ereignis sorgfältig darauf, dass Sie das richtige Steuerelement ausgewählt haben. Das Ereignis gibt es nämlich für nahezu jedes Steuerelement.

Über die Methode `System.Media.SystemSounds.<Soundname>.Play()` können Sie einen Standard-Sound von Windows wiedergeben. Den Platzhalter `<Soundname>` müssen Sie dabei durch den englischen Namen des gewünschten Sounds ersetzen.

Über die Klasse `System.Media.SoundPlayer` können Sie auch beliebige Dateien im Wave-Format wiedergeben lassen.

Aufgaben zur Selbstüberprüfung

- 4.1 Mit welcher Eigenschaft eines Formulars können Sie die Anzeige des Systemmenüs ein- beziehungsweise ausschalten?

- 4.2 In welcher Gruppe bei den Ereignissen finden Sie das Ereignis **Enter**?

- 4.3 Sie haben in einem Formular über die Eigenschaft **AcceptButton** eine Standard-Schaltfläche definiert. Zeigt diese Eigenschaft auch dann Wirkung, wenn sich der Fokus aktuell auf einer anderen Schaltfläche befindet? Begründen Sie kurz Ihre Antwort.

- 4.4 Sie wollen über die Klasse `System.Media.SoundPlayer` eine Sound-Datei **test.wav** im Ordner `c:\test` wiedergeben. Formulieren Sie bitte die entsprechenden Anweisungen. Das Laden der Datei soll über den Konstruktor der Klasse erfolgen.

Schlussbetrachtung

Kompliment! Sie haben nun Ihr zweites anspruchsvoller Projekt umgesetzt – einen Bildbetrachter mit einer integrierten Bilderschau. Sie wissen jetzt auch, wie Sie neue Formulare erzeugen und anzeigen und wie Sie Anweisungen in einem Programm über einen Timer zeitgesteuert verarbeiten.

Wie Sie selbst gesehen haben, ist es mit einer Windows Forms-Anwendung gar nicht so schwer, leistungsfähige Programme zu entwickeln. Das größte Problem besteht häufig darin, das passende Steuerelement zu finden und die zahlreichen Eigenschaften, Methoden und Ereignisse gezielt einzusetzen. Richtig programmiert mit der Eingabe von Quelltexten haben Sie auch in diesem Studienheft nicht sehr viel.

Ein weiteres Problem beim Erstellen von Windows-Anwendungen ist die Betriebssicherheit. Denn Sie können sich ja nie sicher sein, dass der Anwender tatsächlich alle Schritte in der richtigen Reihenfolge korrekt durchgeführt hat. Gehen Sie deshalb nie einfach davon aus, dass bestimmte Voraussetzungen schon erfüllt sein werden. Sorgen Sie durch umfangreiche Abfragen dafür, dass die Voraussetzungen, die für einen erfolgreichen Programmablauf benötigt werden, auch tatsächlich gegeben sind. Wenn das nicht der Fall ist, brechen Sie die Verarbeitung ab und zeigen Sie dem Anwender zum Beispiel über eine Meldung, was er noch zu tun hat.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Um eine Anwendung direkt nach dem Start zu maximieren, setzen Sie die Eigenschaft **WindowState** auf `Maximized`.

- 1.2 Es gibt zwei Möglichkeiten:

Sie öffnen über die Eigenschaft **TabPage**s den TabPage-Sammlungs-Editor und nehmen dort die Änderungen vor.

Sie wechseln im Designer in das gewünschte Register und klicken dann auf den großen Bereich in der Mitte des Registers. Anschließend legen Sie die Eigenschaften wie gewohnt über das Eigenschaftenfenster fest.

Für die richtige Antwort reicht es aus, wenn Sie eine der beiden Möglichkeiten genannt haben.

- 1.3 Damit ein Steuerelement automatisch an die aktuelle Größe des Formulars angepasst wird, verankern Sie es an allen vier Rändern des Formulars.

Kapitel 2

- 2.1 Um ein Bild für eine **PictureBox** über das Eigenschaftenfenster zu setzen, wählen Sie das gewünschte Bild über die Eigenschaft **Image** aus.

Um ein Bild zur Laufzeit des Programms in einer **PictureBox** anzuzeigen, benutzen Sie die Methode `Load()` der **PictureBox**. Als Argument erwartet die Methode den vollständigen Pfad zur Grafikdatei.

- 2.2 Damit das Bild so groß wie möglich und ohne Verzerrungen angezeigt wird, setzen Sie die Eigenschaft **SizeMode** der **PictureBox** auf `Zoom`.

Bei Veränderungen zur Laufzeit muss vor den Wert noch die Angabe `PictureBoxSizeMode`. gesetzt werden. Andernfalls kennt der Compiler den Wert nicht.

- 2.3 Die Anweisung zum Anzeigen des Öffnendialogs lautet:

```
openFileDialog2.ShowDialog();
```

- 2.4 Die Methode kann wie gewohnt über den Namen aufgerufen werden. Dabei müssen allerdings die beiden Argumente `sender` und `e` übergeben werden. Der vollständige Aufruf könnte zum Beispiel so aussehen:

```
ButtonAuswahl_Click(sender, e);
```

- 2.5 Um zu überprüfen, ob eine Datei existiert, verwenden Sie die Methode `System.IO.File.Exists()`. Als Argument erwartet die Methode den Namen der Datei.

- 2.6 Der vollständige Ausdruck muss lauten:

```
Word-Dateien\*.docx
```

Zuerst wird der Name des Filters angegeben, dann das Zeichen | und abschließend die Dateiliste.

- 2.7 Beim Anklicken der Schaltfläche wird eine neue Instanz des Formulars erzeugt und das Formular angezeigt. Diese beiden Anweisungen könnten zum Beispiel so aussehen:

```
Form2 neuesFormular = new Form2();
neuesFormular.ShowDialog();
```

beziehungsweise

```
neuesFormular.Show();
```

Mit der Methode `ShowDialog()` wird das Formular modal angezeigt, mit der Methode `Show()` nicht modal.

Kapitel 3

- 3.1 Die Eigenschaft **FileName** liefert nur den Namen **einer** ausgewählten Datei. Die Eigenschaft **FileNames** liefert die Namen **aller** ausgewählten Dateien.
- 3.2 Die Verarbeitung kann in einer `foreach`-Schleife erfolgen, die auf jedes Element in dem Array einzeln zugreift. Die Konstruktion mit dem Zwischenspeichern könnte zum Beispiel so aussehen:

```
string[] dateien = openFileDialog1.FileNames;
foreach (string datei in dateien)
    ...
```

Die Bezeichner für die einzelnen Objekte können auch abweichen.

- 3.3 Um den Text in der Titelleiste eines Öffnendialogs zu verändern, setzen Sie die Eigenschaft **Title** des Dialogs auf den gewünschten Wert
- 3.4 Das Ereignis, das von einem Timer ausgelöst wird, heißt **Tick**.
- 3.5 Der Timer wird alle zwei Sekunden ausgelöst.
- 3.6 Nein, der Timer wird nicht ausgelöst. Durch die modale Anzeige wird die weitere Verarbeitung in dem anderen Formular quasi blockiert.

Kapitel 4

- 4.1 Die Eigenschaft heißt **ControlBox**.
- 4.2 Das Ereignis **Enter** befindet sich in der Gruppe **Fokus**.
- 4.3 Nein, wenn eine andere Schaltfläche den Fokus hat, wird beim Drücken der Eingabetaste das Klicken auf diese Schaltfläche simuliert.

4.4 Die Anweisungen könnten zum Beispiel so aussehen:

```
System.Media.SoundPlayer player = new  
System.Media.SoundPlayer("c:\\test\\\\test.wav");  
player.Play();
```

oder

```
System.Media.SoundPlayer player = new  
System.Media.SoundPlayer(@"c:\\test\\test.wav");  
player.Play();
```

Wichtig ist vor allem die korrekte Pfadangabe als Argument beim Erzeugen der Instanz. Die Trennzeichen \ müssen dabei doppelt angegeben werden beziehungsweise das Zeichen @ muss vor die Zeichenkette gestellt werden.

B. Glossar

Aktivierreihenfolge	Die Aktivierreihenfolge in einem Formular bestimmt, in welcher Reihenfolge der Fokus auf die Steuerelemente gesetzt wird.
Eigenschaftenfenster	Das Eigenschaftenfenster ist ein Teil der Oberfläche von Visual Studio 2019. Sie können hier Eigenschaften von Steuerelementen und die Reaktion auf Ereignisse festlegen.
Ereignis	Ein Ereignis ist eine Art Signal, das von bestimmten Aktionen ausgelöst wird. Zu den Aktionen, die ein Ereignis auslösen, gehören zum Beispiel das Bewegen der Maus, das Klicken, das Doppelklicken oder auch eine Tastatureingabe.
Escape-Sequenzen	Escape-Sequenzen bilden Steuerzeichen und Sonderzeichen ab. Eine Escape-Sequenz beginnt immer mit dem Zeichen \.
Feld	Als Feld wird in der Programmiersprache C# ein Attribut einer Klasse bezeichnet.
Fokus	Der Fokus in einem Formular bestimmt, welches Steuerelement aktuell auf Tastatureingaben reagiert.
Form1	Form1 ist das Hauptfenster einer Windows Forms-Anwendung.
Formular	Ein Formular von Visual Studio 2019 entspricht einem Fenster einer Windows Forms-Anwendung.
Garbage Collection	Die <i>Garbage Collection</i> gibt Speicher frei und reorganisiert den Speicher. Sie wird automatisch ausgeführt.
Garbage Collector	Der <i>Garbage Collector</i> ist der Prozess, der die <i>Garbage Collection</i> ausführt.
Instanz	Eine Instanz ist ein Objekt einer Klasse.
Konstruktor	Ein Konstruktor ist eine besondere Methode, die automatisch beim Erzeugen einer Instanz aufgerufen wird.
Modales Fenster	Ein modales Fenster steht immer im Vordergrund und kann auch nicht minimiert werden. Mit der eigentlichen Anwendung kann erst dann weitergearbeitet werden, wenn das modale Fenster wieder geschlossen wird. Typische Beispiele für modale Fenster sind unter anderem die Öffnen- und Speicherndialoge von Windows.

Nicht visuelle Steuerelemente	Nicht visuelle Steuerelemente werden beim Ausführen einer Anwendung nicht direkt im Formular angezeigt. Es handelt sich also nicht um Bedienelemente im eigentlichen Sinne. Typische nicht visuelle Steuerelemente sind zum Beispiel das Steuerelement OpenFileDialog für den Standard-Öffnendialog und das Steuerelement Timer .
Steuerelement	Die Steuerelemente von Visual Studio 2019 stellen vorgefertigte Bedienelemente, Objekte und Methoden zur Verfügung.
Timer	Ein <i>Timer</i> löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis aus, in dem Sie Anweisungen verarbeiten lassen können. Ein <i>Timer</i> ermöglicht damit die zeitgesteuerte Verarbeitung von Anweisungen.
Toolbox	Die Toolbox ist ein Teil der Oberfläche von Visual Studio 2019. Sie ermöglicht das komfortable Einfügen von Steuerelementen.
Windows Forms-Anwendung	Bei einer Windows Forms-Anwendung wird die Oberfläche eines Programms im Wesentlichen aus Formularen und vorgefertigten Steuerelementen zusammengebaut. Die eigentliche Programmierarbeit beschränkt sich vor allem auf die Logik.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019. Ideal für Programmieranfänger.* 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Der TabPage-Sammlungs-Editor	4
Abb. 1.2	Das Formular mit den beiden Registern	5
Abb. 1.3	Die Auswahl der Anker	6
Abb. 1.4	Der erste Test des Programms	7
Abb. 2.1	Die Steuerelemente im ersten Register	10
Abb. 2.2	Die erste angezeigte Grafik	12
Abb. 2.3	Eine Grafik in der optimalen Größe	13
Abb. 2.4	Das Steuerelement OpenFileDialog im Formular	14
Abb. 2.5	Der Dialog Öffnen im Einsatz	16
Abb. 2.6	Der Dialog Neues Element hinzufügen	19
Abb. 2.7	Das neue Formular im Designer	20
Abb. 2.8	Die Anzeige im zweiten Formular	24
Abb. 3.1	Der Rohbau des Registers Bilderschau	29
Abb. 3.2	Das kopierte Steuerelement	30
Abb. 3.3	Das Listenfeld mit einigen Einträgen	32
Abb. 3.4	Der eingefügte Timer	33

E. Tabellenverzeichnis

Tab. 2.1	Beispiele für die Eigenschaft Filter	17
Tab. 2.2	Grundeinstellungen für das neue Formular	20
Tab. 4.1	Die Sounds von Windows	43

F. Codeverzeichnis

Code 2.1	Die Methode für das Anklicken der Schaltfläche Anzeigen.....	11
Code 2.2	Die Anweisungen zum Setzen der EigenschaftSizeMode.....	13
Code 2.3	Die Anweisungen für die Methode OpenFileDialog1_FileOk()	15
Code 2.4	Die geänderte Methode ButtonAnzeigen_Click()	22
Code 2.5	Die Methode zum Laden des Bildes	23
Code 3.1	Die vollständige Methode ButtonAuswaehlen_Click()	30
Code 3.2	Die Anweisungen für das Ereignis FileOk des zweiten Öffnendialogs ..	31
Code 3.3	Das Anzeigen des ersten Bildes.....	34
Code 3.4	Die Anweisungen für das Ereignis Tick des Timers	36
Code 4.1	Die Methode OpenFileDialog2_FileOk() mit dem Leeren des Listenfelds	40
Code 4.2	Die erweiterte Methode ButtonStarten_Click()	41
Code 4.3	Die Methoden TabPageEinzel_Enter() und TabPageSchau_Enter()	42
Code 4.4	Das Abspielen eines Sounds beim Anzeigen eines neuen Bildes	43

G. Sachwortverzeichnis

F

Fenster	
modales	22
Filter	
definieren	17
Formular	
neues erstellen	19

P

PictureBox	10
------------------	----

S

Soundwiedergabe	42
Standard-Dialog	
Öffnen	14
Steuerelement	
nicht visuelles	14
Systemsounds	42

T

Timer	32
Trennzeichen	44

W

Wave-Format	43
-------------------	----

Z

Zeichen @	44
-----------------	----

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP08D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Senden Sie bitte bei allen Aufgaben, bei denen Sie Programme erstellen oder ändern sollen, immer die vollständigen Projekte ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

- Definieren Sie für die beiden Öffnendialoge in unserem Bildbetrachter einzelne Filter für die Grafikformate BMP, GIF und JPG. Die Filter sollen jeweils nur die Dateien des entsprechenden Formats in dem Dialog anzeigen.

Notieren Sie für die Lösung dieser Aufgabe bitte den vollständigen Ausdruck für die Eigenschaft **Filter** der Dialoge. Der Filter für sämtliche Grafikdateien soll dabei erhalten bleiben.

5 Pkt.

- Wenn ein Anwender in der Einzelbildanzeige im Bildbetrachter die Anzeige in einem neuen Fenster wählt, bleibt die Anzeige im Register **Einzelbild** leer beziehungsweise es wird nach wie vor das vorher ausgewählte Bild angezeigt. Sorgen Sie dafür, dass die Anzeige nicht nur im neuen Fenster erfolgt, sondern auch im Register.

Notieren Sie für die Lösung dieser Aufgabe bitte, welche Änderungen an welchen Stellen im Quelltext erforderlich sind.

5 Pkt.

- Lassen Sie im Formular für die Einzelbildanzeige in dem Bildbetrachter neben dem Text **Große Darstellung** auch noch den Pfad und den Namen der aktuell angezeigten Grafikdatei anzeigen.

Für die Datei **test.bmp** im Ordner **c:\test** könnte der Text in der Titelleiste zum Beispiel so aussehen:

Große Darstellung – c:\test\test.bmp

Ändern Sie auch den Text in der Titelleiste des Formulars für die Bilderschau. Dort sollen neben dem Text **Bilderschau** und dem Pfad und Namen der aktuell angezeigten Datei auch noch die Anzahl der Bilder in der Bilderschau und die Nummer des aktuellen Bildes angezeigt werden.

Bei einer Bilderschau mit insgesamt drei Bildern sollte der Text in der Titelleiste des Formulars für das erste Bild ungefähr so aussehen:

Bilderschau – Bild 1 von 3 c:\test\test.bmp

Notieren Sie für die Lösung dieser Aufgabe bitte ebenfalls, welche Änderungen an welchen Stellen im Quelltext erforderlich sind.

30 Pkt.

4. Erweitern Sie den Bildbetrachter so, dass die Anzeigedauer der Bilder in der Bilderschau nicht mehr fest auf 10 Sekunden gesetzt wird, sondern vom Anwender selbst ausgewählt werden kann. Die Angabe soll dabei in Sekunden erfolgen. Bitte beachten Sie dabei, dass der Wert 0 nicht zulässig ist.

Beschreiben Sie für die Lösung dieser Aufgabe auch, welche grundsätzlichen Schritte für diese Erweiterung erforderlich sind.

Ein Tipp zur Lösung:

Sie können für das Einstellen der Anzeigedauer ein Drehfeld verwenden. Sie finden das entsprechende Steuerelement **NumericUpDown** in der Gruppe **Allgemeine Steuerelemente** der Toolbox. Den aktuellen Wert in einem Drehfeld können Sie über die Eigenschaft **Value** abfragen. Allerdings liefert Ihnen die Eigenschaft **Value** einen Wert vom Typ `decimal`, den Sie erst noch konvertieren müssen.

Die untere und obere Grenze für die Werte in einem Drehfeld setzen Sie über die Eigenschaften **Minimum** und **Maximum**. Weitere Details zu dem Steuerelement sehen Sie bitte bei Bedarf in der Hilfe nach.

Wenn Sie kein Drehfeld verwenden wollen, können Sie auch ein einfaches Eingabefeld benutzen.

60 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Fehlersuche und Ausnahmebehandlung

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP09D

Objektorientierte Software-Entwicklung mit C#

Fehlersuche und Ausnahmebehandlung

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Fehlersuche und Ausnahmebehandlung

Inhaltsverzeichnis

Einleitung	1
1 Das Problem	3
Zusammenfassung	6
2 Der integrierte Debugger	7
2.1 Anwendung schrittweise ausführen lassen	8
2.2 Inhalte von Datenobjekten prüfen	11
2.3 Änderungen im Debug-Modus vornehmen	12
2.4 Methoden debuggen	13
Zusammenfassung	16
3 Erweiterte Debugging-Funktionen	18
3.1 Windows Forms-Anwendungen debuggen	18
3.2 Arbeiten mit Haltepunkten	22
3.3 Weitere nützliche Debugger-Funktionen	26
Zusammenfassung	32
4 Exkurs: Vom richtigen Testen	33
4.1 Probleme des Testens	33
4.2 Testmethoden	34
Zusammenfassung	37
5 Ausnahmebehandlung	39
5.1 Exceptions	40
5.2 try und catch	41
5.3 Eigene Exceptions auslösen	47
5.4 Die Anweisung finally	49
5.5 Exceptions filtern	50
5.6 Wann lohnt eine eigene Ausnahmebehandlung?	52
Zusammenfassung	53
Schlussbetrachtung	55

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	56
B.	Glossar	58
C.	Tastenkombinationen zum Arbeiten mit dem Debugger	61
D.	Standard-Exceptions	62
E.	Literaturverzeichnis	63
F.	Abbildungsverzeichnis	64
G.	Tabellenverzeichnis	65
H.	Codeverzeichnis	66
I.	Medienverzeichnis	67
J.	Sachwortverzeichnis	68
K.	Einsendeaufgabe	69

Einleitung

In diesem Studienheft konzentrieren wir uns auf die Fehlersuche und die Behandlung von Ausnahmen. Solche Ausnahmen können zum Beispiel durch Laufzeitfehler wie eine Division durch null auftreten.

Im Einzelnen lernen Sie in diesem Studienheft,

- welche Fehler bei der Programmierung immer wieder auftreten,
- was sich hinter einem *Off-by-one*-Fehler verbirgt,
- wie Sie Fehler mit Testausgaben einkreisen,
- welche Möglichkeiten der integrierte Debugger von Visual Studio bietet,
- wie Sie ein Programm im Debug-Modus schrittweise ausführen lassen,
- wie Sie die Inhalte von Datenobjekten mit dem Debugger prüfen können,
- wie Sie Änderungen im Debug-Modus vornehmen,
- welche Besonderheiten Sie beim Debuggen von Methoden beachten müssen,
- wie Sie Windows Forms-Anwendungen debuggen,
- wie Sie mit Haltepunkten arbeiten,
- wie Sie Werte im Debug-Modus ständig anzeigen lassen,
- wie Sie ein Programm richtig testen,
- was sich hinter der Ausnahmebehandlung verbirgt und
- wie Sie eine eigene Ausnahmebehandlung programmieren.

Christoph Siebeck

1 Das Problem

In diesem Kapitel lernen Sie typische Fehler beim Programmieren kennen. Außerdem erfahren Sie, welche einfachen Mittel zur Fehlersuche Sie einsetzen können.

Wie Sie wahrscheinlich aus eigener Erfahrung wissen, unterlaufen auch dem besten Programmierer immer wieder Fehler – sei es, weil Sie schlicht und einfach unaufmerksam waren oder weil Sie ein komplexes Problem nicht richtig gelöst haben. Die Fehler führen dann in der Regel dazu, dass Ihr Programm nicht genau das macht, was es machen soll – es verhält sich falsch.

Zur Auffrischung:

Grundsätzlich kann man zwischen Syntax- und Semantikfehlern unterscheiden. Syntaxfehler sind Verstöße gegen die Regeln der Programmiersprache – zum Beispiel ein fehlendes Semikolon oder eine falsche Klammer. Semantikfehler dagegen sind Fehler in der Programmlogik.

Syntaxfehler werden vom Compiler bei der Übersetzung des Programms gemeldet. Semantikfehler kann der Compiler in der Regel nicht feststellen.



Schauen wir uns einmal einige sehr einfache Beispiele für Semantikfehler an.

Die folgenden Anweisungen sollen in einer Schleife von 1 bis 10 zählen und den aktuellen Schleifendurchlauf anzeigen:

```
int zaehler = 1;
while (zaehler < 10)
    Console.WriteLine("Schleifendurchlauf {0}", zaehler);
```

Code 1.1: Ein kleiner Fehler mit großer Wirkung



Bevor Sie weiterlesen:

Wo liegt das Problem im vorigen Code? Versuchen Sie es einmal selbst herauszufinden.

Der Fehler ist hier recht offensichtlich. In der Schleife fehlt die Anweisung zum Verändern der Variablen `zaehler`. Damit wird endlos immer der Text

Schleifendurchlauf 1
ausgegeben.



Zur Erinnerung:

Eine Endlosschleife können Sie mit der Tastenkombination **Strg** + **C** abbrechen.

Aber auch wenn Sie die Anweisung zum Verändern des Zählers in den Quelltext aufnehmen, läuft das Programm nicht so wie gewünscht. Die Schleife wird zwar ausgeführt – aber nur bis 9 und nicht wie gewünscht bis 10.

```

int zaehler = 1;
while (zaehler < 10)
{
    Console.WriteLine("Schleifendurchlauf {0}", zaehler);
    zaehler++;
}

```

Code 1.2: Immer noch nicht das gewünschte Ergebnis

Auch hier ist der Fehler schnell gefunden:

Die Abbruchbedingung der Schleife muss nicht `zaehler < 10` lauten, sondern `zaehler <= 10` oder `zaehler < 11`. Dann hat auch alles seine Ordnung.



Fehler, die entstehen, weil ein Wert um 1 zu groß oder um 1 zu klein ist, werden auch **Off-by-one-Fehler** genannt. Diese „Um eins daneben“- oder „Einen entfernt“-Fehler haben in der Regel eine kleine Ursache und eine große Wirkung – zum Beispiel eben einen Schleifendurchlauf zu viel oder zu wenig.

Nun gibt es aber zahlreiche Fehler, die nicht so offensichtlich sind – zum Beispiel in verschachtelten Verzweigungen. Auch dazu ein Beispiel: Im folgenden Fragment sollen abhängig vom Wert einer Eingabe entweder 10, 20 oder 30 addiert werden.

```

int zahl;
Console.Write("Bitte geben Sie eine ganze Zahl ein: ");
zahl = Convert.ToInt32(Console.ReadLine());
if (zahl == 5)
    zahl = zahl + 10;
if (zahl <= 5)
    zahl = zahl + 20;
if (zahl > 5)
    zahl = zahl + 30;
Console.WriteLine("Zahl hat den Wert {0}.", zahl);

```

Code 1.3: Noch ein Fehler



Bevor Sie weiterlesen:

Versuchen Sie, auch im vorigen Code den Fehler erst einmal selbst zu finden.

Bei diesem Code müssen Sie schon etwas genauer hinsehen, um die Fehler zu finden. Denn das Programm funktioniert nur dann richtig, wenn ein Wert größer als 5 eingegeben wird. Bei der Eingabe einer Zahl kleiner als 5 werden zuerst 20 zum eingegebenen Wert addiert und dann noch einmal 30, da der Wert nach der ersten Addition ja größer ist als 5. Wenn die Zahl gleich 5 ist, werden erst 10 addiert und anschließend 30.

Solche Schwierigkeiten lassen sich durch geschickte Programmierung vermeiden. Im vorigen Code könnten Sie zum Beispiel `else`-Zweige einbauen und damit sicherstellen, dass die verschiedenen Anweisungen nur dann ausgeführt werden, wenn eine Bedingung nicht zutrifft.

Nun gibt es aber auch Probleme, die sich nicht durch geschickte Programmierung vermeiden lassen. Dazu gehört zum Beispiel der Aufruf einer Methode mit vertauschten Argumenten. Schauen wir uns auch dazu ein ganz einfaches Beispiel an: Im folgenden Programmfragment wird eine Methode zur Subtraktion aufgerufen. Gerechnet werden soll $20 - 10$.

```
//die Methode zur Subtraktion
static int Subtraktion(int wert1, int wert2)
{
    return (wert1 - wert2);
}

static void Main(string[] args)
{
    int zahl;
    //eigentlich soll 20 - 10 gerechnet werden
    zahl = Subtraktion(10, 20);
    Console.WriteLine("Zahl hat den Wert {0}.", zahl);
}
```

Code 1.4: Vertauschte Argumente beim Aufruf einer Methode

Da die Argumente beim Aufruf der Methode in einer falschen Reihenfolge angegeben werden, wird nicht $20 - 10$ gerechnet, sondern $10 - 20$.

Hier könnten Ihnen Testmeldungen weiterhelfen, die zum Beispiel in der Methode die Werte der Argumente ausgeben. Das könnte dann so aussehen:

```
static int Subtraktion(int wert1, int wert2)
{
    Console.WriteLine("Wert1 hat den Wert {0}.", wert1);
    Console.WriteLine("Wert2 hat den Wert {0}.", wert2);
    return (wert1 - wert2);
}
```

Testmeldungen geben Ihnen zwar in vielen Fällen erste Hinweise auf eine mögliche Fehlerquelle, brauchbar sind sie allerdings nicht immer. Zum einen ist der Aufwand für das Einfügen recht hoch und zum anderen liefern sie auch nur sehr beschränkte Informationen. Und: Testmeldungen werden auch gerne im Programm „vergessen“. Schon fast ein Klassiker sind Ausgaben wie „Hallo“ oder „Huhu“, die urplötzlich in einer Anwendung erscheinen.

Tipp:

Um solch unangenehme Überraschungen zu vermeiden, können Sie für Testausgaben auch die Methoden `Write()` und `WriteLine()` der Klasse `System.Diagnostics.Debug` verwenden. Die Meldungen dieser Methoden werden nur beim Debuggen von Programmen in einem speziellen Ausgabebereich unten im Fenster von Visual Studio angezeigt. Wenn Sie das nächste Kapitel zum integrierten Debugger durchgearbeitet haben, können Sie die Anweisungen ja einmal mit den beiden vorigen Codes testen.

Sehr viel effektiver als die Ausgabe von Testmeldungen ist die Fehlersuche mit einem speziellen Werkzeug von Visual Studio – dem integrierten Debugger. Damit werden wir uns im nächsten Kapitel beschäftigen.

Zusammenfassung

Fehler unterlaufen bei der Programmierung immer wieder.

Grundsätzlich wird zwischen Syntax- und Semantikfehlern unterschieden. Syntaxfehler findet der Compiler bei der Übersetzung, Semantikfehler in der Regel nicht.

Eine Möglichkeit, einen Fehler einzukreisen, bieten Testausgaben.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Was verstehen Sie unter einem Off-by-one-Fehler?

- 1.2 Sie wollen den aktuellen Wert von zwei numerischen Variablen vergleichen und vergessen eines der beiden Gleichheitszeichen im Ausdruck. Handelt es sich dabei um einen Syntax- oder um einen Semantikfehler? Begründen Sie bitte Ihre Antwort.

2 Der integrierte Debugger

In diesem Kapitel beschäftigen wir uns mit dem integrierten Debugger von Visual Studio.

Ein **Debugger** ist ein Werkzeug zur Fehlersuche. Wörtlich übersetzt bedeutet Debugger so viel wie „Entwanzer“. Dieser etwas seltsam anmutende Name leitet sich von einem Insekt ab, das in den Anfangszeiten der Informationstechnik einen Kurzschluss in einem Rechner verursachte. Seitdem werden Fehler in einem Computerprogramm auch *Bug* (engl. „Insekt, Wanze“) genannt. Das Beseitigen von Fehlern heißt dann *debuggen*.



Der Debugger bietet Ihnen unter anderem folgende Möglichkeiten:

- Sie können ein Programm Anweisung für Anweisung ausführen lassen.
- Sie können die aktuellen Inhalte von Datenobjekten wie Variablen oder Feldern anzeigen lassen.
- Sie können das Programm bis zu einer bestimmten Stelle ausführen lassen.
- Sie können gezielt Haltepunkte in einem Programm setzen.

Schauen wir uns die verschiedenen Möglichkeiten der Reihe nach an. Dazu verwenden wir zunächst ein einfaches Konsolenprogramm.

Hinweis:

Mit dem Debuggen von Windows Forms-Anwendungen beschäftigen wir uns im nächsten Kapitel.

Legen Sie bitte ein neues Projekt für eine Konsolenanwendung an. Geben Sie dann den Quelltext aus dem Code 1.2 als Methode `Main()` ein. Übernehmen Sie dabei bitte auch den Fehler in der Bedingung der Schleife.

Tipp:

Sie finden das Projekt auch im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Debug01**.

2.1 Anwendung schrittweise ausführen lassen

Schauen wir uns nun an, wie Sie eine Anwendung schrittweise ausführen lassen. Dazu verwenden Sie die Funktionen **Einzelschritt** und **Prozedurschritt** im Menü **Debuggen** beziehungsweise die Symbole **Einzelschritt** und **Prozedurschritt** in der Symbolleiste **Debuggen**.

Hinweis:

Die Symbolleiste **Debuggen** wird automatisch eingeblendet, wenn Sie das Debuggen starten.

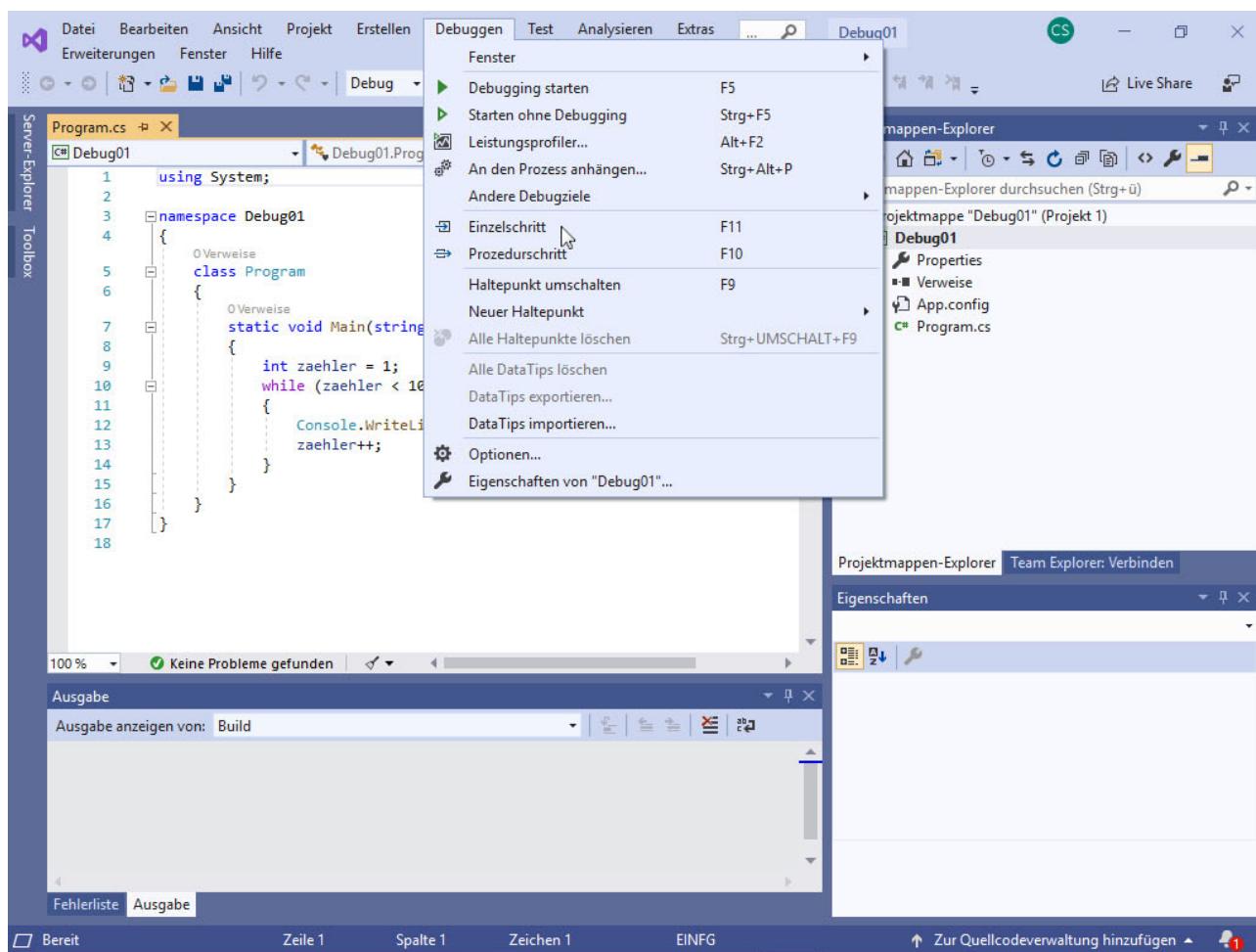


Abb. 2.1: Die Funktionen zum Debuggen im Menü **Debuggen** (in der Mitte des Menüs am Mauszeiger)

Die beiden Funktionen unterscheiden sich vor allem in der Ausführung von Methoden. Mit dem Befehl **Prozedurschritt** wird die Methode komplett ausgeführt, bei dem Befehl **Einzelschritt** dagegen Anweisung für Anweisung. Den Unterschied werden wir uns gleich noch genauer ansehen.

Starten Sie jetzt bitte die Funktion **Einzelschritt**. Klicken Sie auf den entsprechenden Eintrag im Menü **Debuggen**.

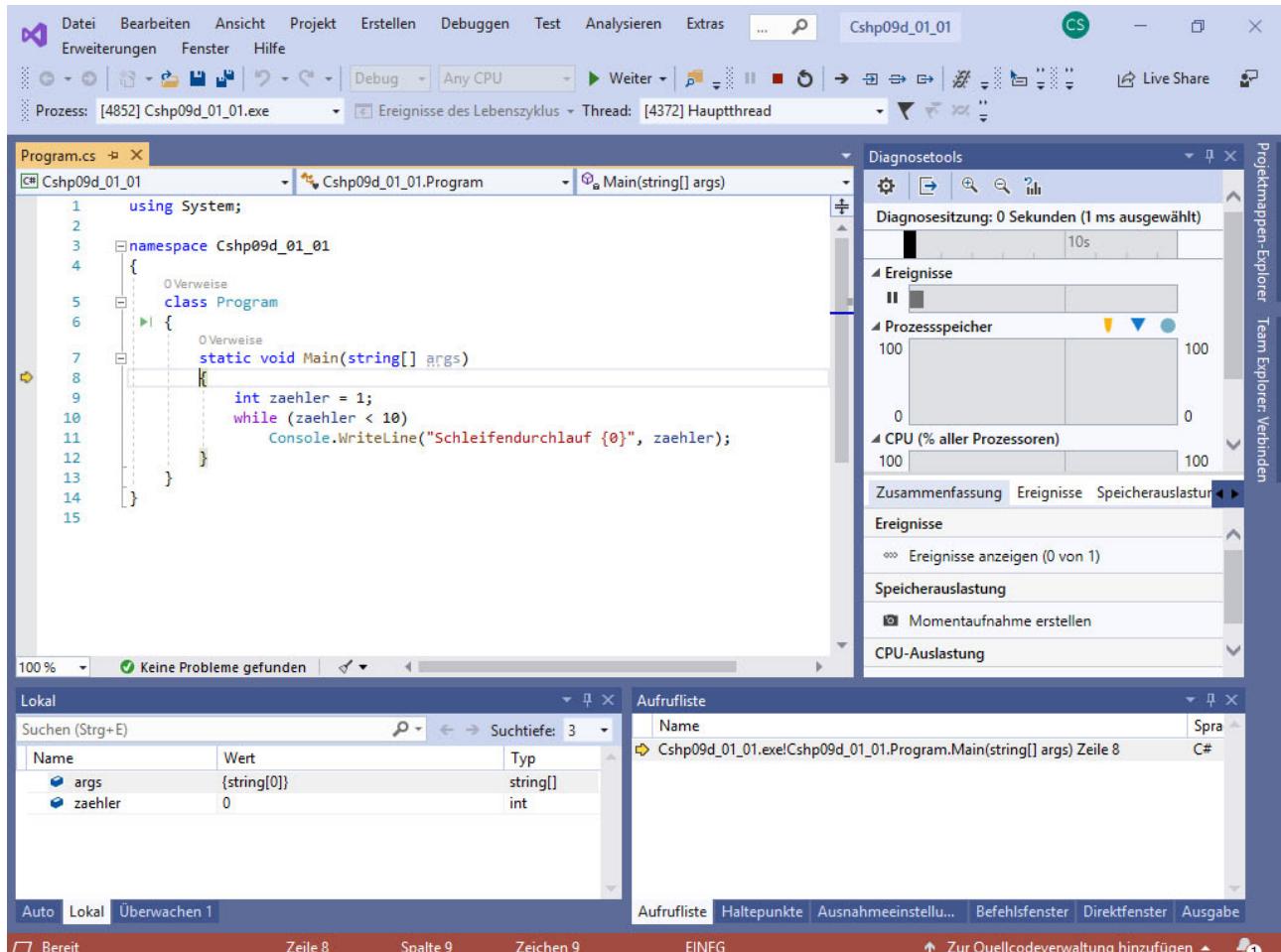


Abb. 2.2: Das Programm beim Debuggen

Das Projekt wird zunächst kompiliert. Danach wird die Ausführung gestartet und direkt wieder angehalten. Das erkennen Sie daran, dass im Hintergrund jetzt das Konsolenfenster sichtbar ist. Außerdem wird die Statusleiste rotbraun dargestellt. Im Fenster von Visual Studio finden Sie unten zusätzlich die Bereiche **Auto** sowie **Aufrufliste** und oben die Symbolleiste **Debuggen**. Rechts im Fenster werden die Diagnosetools angezeigt.

Im Quelltext selber ist die Zeile mit der ersten öffnenden geschweiften Klammer der Methode `Main()` mit einem kleinen gelben Pfeil links vor der Zeile markiert und zusätzlich mit einem Rahmen versehen. Diese Zeile wird beim nächsten Aufruf der Funktion **Einzelsschritt** ausgeführt.

Rufen Sie die Funktion **Einzelsschritt** jetzt bitte noch einmal auf.

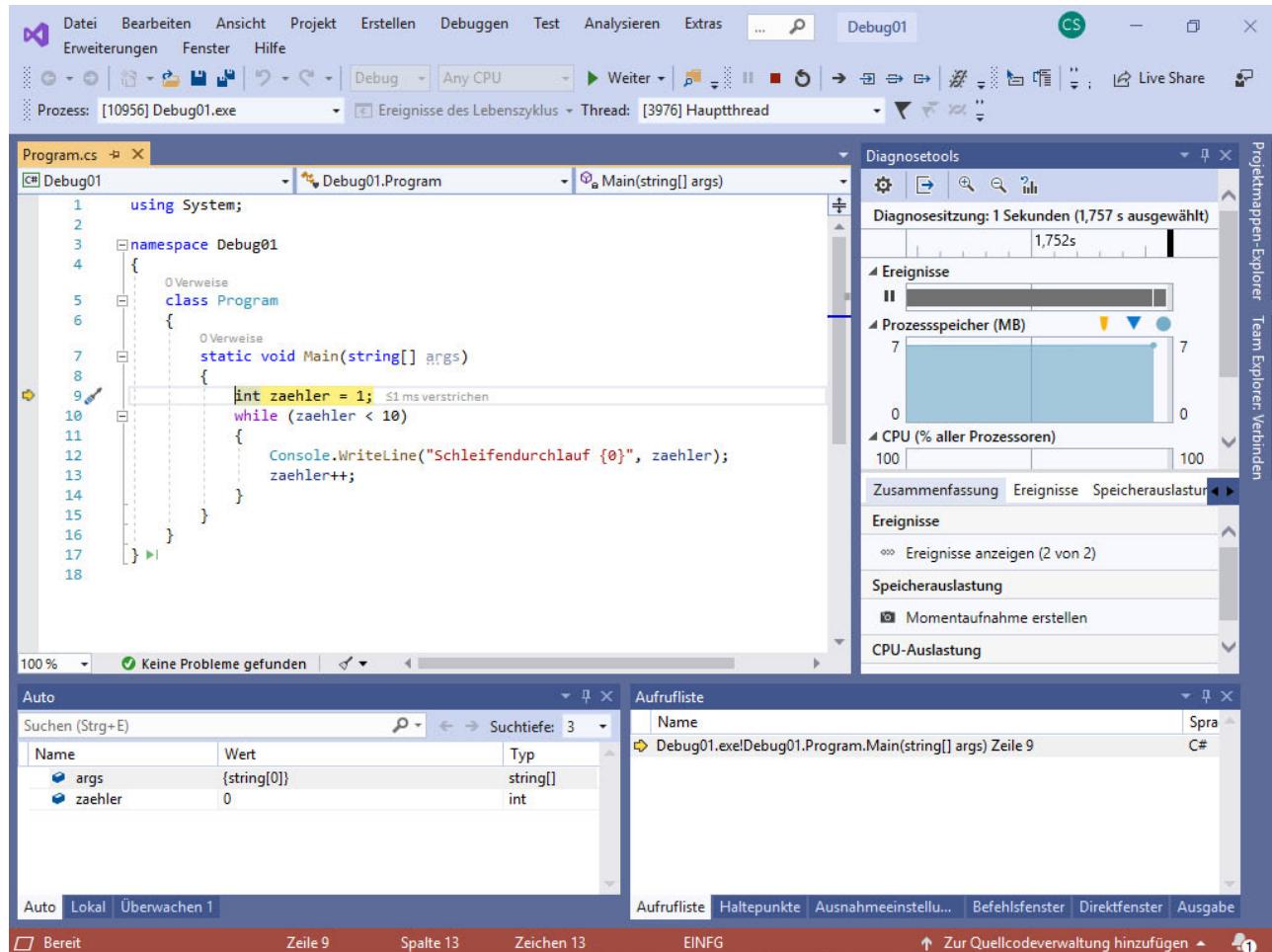


Abb. 2.3: Die nächste Anweisung im Debugger

Wie Sie sehen, sind der gelbe Pfeil und die Markierung verschoben worden. Die Anweisung wird jetzt auch gelb hinterlegt. Wiederholen Sie die Einzelschritte bitte so lange, bis die Ausgabeanweisung markiert wird. Lassen Sie dann auch diese Anweisung ausführen.

Tipp:

Falls das Konsolenfenster bei Ihnen verdeckt wird, können Sie sehr schnell mit der Tastenkombination **Alt** + **Esc** in die Konsole wechseln. Wenn Sie im Konsolefenster noch einmal die Tastenkombination **Alt** + **Esc** drücken, gelangen Sie wieder in den Editor von Visual Studio. Abhängig von den laufenden Anwendungen müssen Sie unter Umständen die Taste **Esc** mehrfach drücken.

Der Aufruf der Anweisungen über das Menü **Debuggen** kann recht lästig werden. Verwenden Sie daher für das Ausführen der Funktion **Einzelschritt** am besten das Symbol **Einzelschritt** oder die Funktionstaste **F11**. Dann müssen Sie nicht immer wieder das Menü **Debuggen** öffnen.

2.2 Inhalte von Datenobjekten prüfen

Den Wert von Datenobjekten können Sie sich jetzt sehr einfach im Debugger anzeigen lassen. Stellen Sie dazu den Mauszeiger auf ein beliebiges Vorkommen des Objekts im Quelltext und warten Sie einen kurzen Moment ab. Visual Studio zeigt Ihnen dann in einem speziellen Bildschirmtipp – dem **DataTip**¹ – den aktuellen Wert an.

Probieren Sie das bitte aus. Lassen Sie sich den aktuellen Wert der Variablen `zaehler` anzeigen.

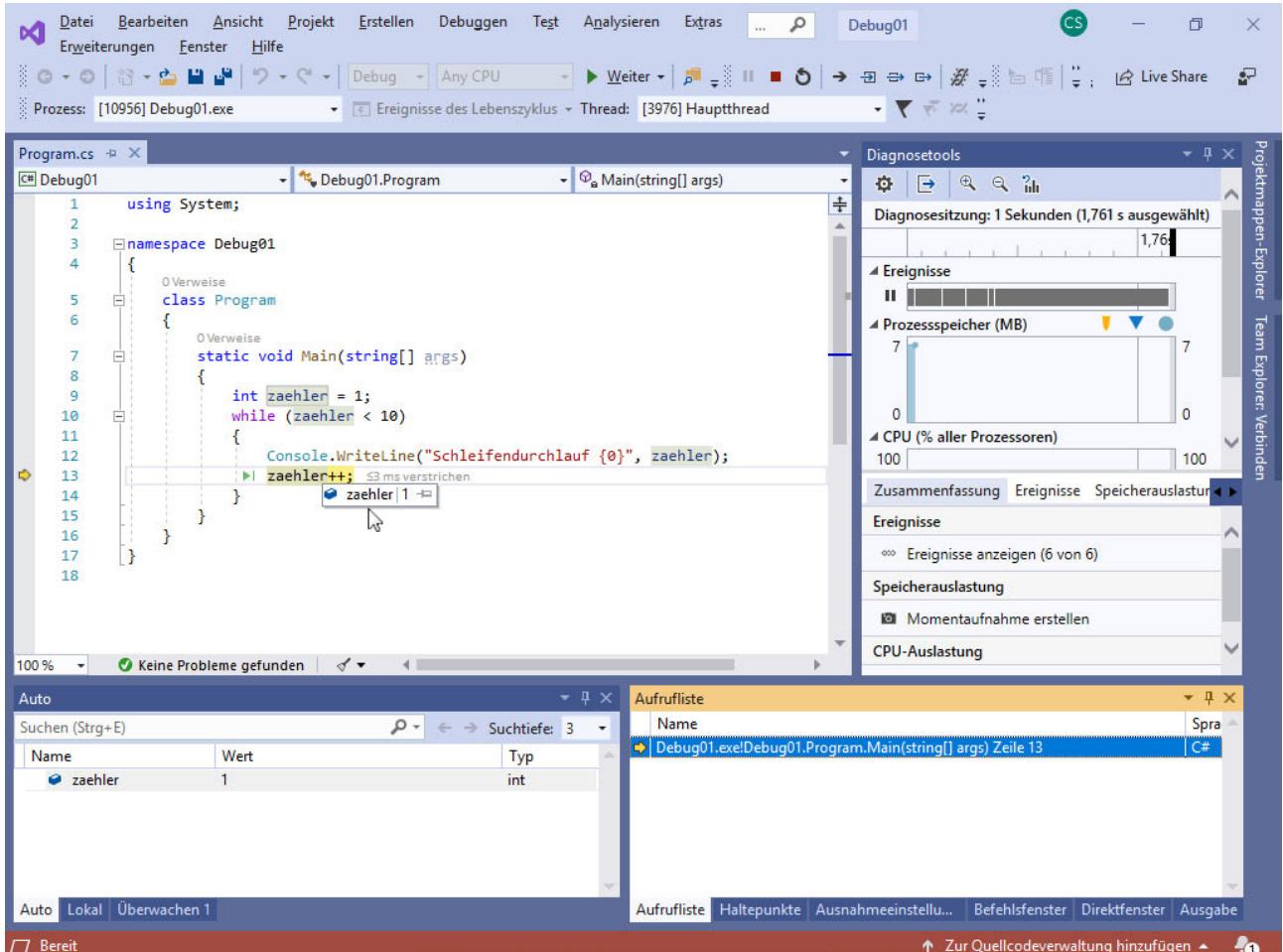


Abb. 2.4: Der Wert von `zaehler` in einem DataTip (in der Mitte der Abbildung am Mauszeiger)

Hinweis:

Falls die Anzeige bei Ihnen nicht funktioniert, überprüfen Sie bitte, ob Sie den Mauszeiger genau auf das Wort `zaehler` gestellt haben.

1. Wörtlich übersetzt bedeutet *data tip* so viel wie „Datentipp“.

Die Anzeige klappt auch für viele andere Elemente in einem Quelltext. Probieren Sie das einfach einmal aus. Stellen Sie den Mauszeiger zum Beispiel auf den Eintrag `args` in der Zeile

```
static void Main(string[] args)
```

Im DataTip sehen Sie dann die übergebenen Argumente. In unserem Fall ist die Liste allerdings leer.

Hinweis:

Der Debugger kennt noch weitere Möglichkeiten, Werte anzuzeigen und auch zu verändern. Damit werden wir uns im nächsten Kapitel ausführlicher beschäftigen.

Lassen Sie jetzt bitte die Anweisung zum Erhöhen von `zaehler` ausführen. Drücken Sie die Funktionstaste **F11** oder klicken Sie auf das Symbol **Einzelschritt** .

Lassen Sie dann noch einmal den Wert von `zaehler` anzeigen. Jetzt sollte im DataTip der Wert 2 erscheinen.

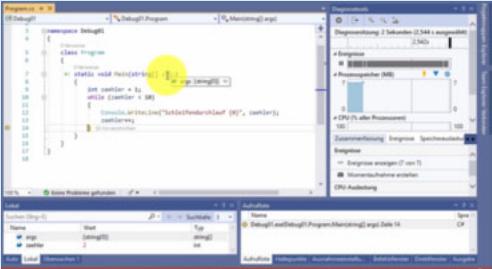
2.3 Änderungen im Debug-Modus vornehmen

Im nächsten Schritt können wir den Fehler in der Bedingung der Schleife korrigieren. Dazu brechen Sie die Ausführung des Programms mit der Funktion **Debuggen beenden** im Menü **Debuggen** ab. Alternativ können Sie auch das Symbol **Debuggen beenden**  in der Symbolleiste **Debuggen** oder die Tastenkombination **Shift + F5** verwenden.

Nach dem Beenden des Programms können Sie die gewünschten Änderungen im Quelltext vornehmen und das Programm danach noch einmal testen. Sie können die Änderungen aber auch direkt beim Debuggen durchführen. Dazu geben Sie die Änderungen wie gewohnt im Quelltext ein. Visual Studio übernimmt die neuen Anweisungen dann automatisch in das laufende Programm.

Korrigieren Sie jetzt bitte die `while`-Anweisung. Führen Sie das Programm dann zeilenweise bis zum Ende aus. Danach sollte der Editor wieder in der gewohnten Darstellung angezeigt werden.





In diesem Video stellen wird Ihnen das Debuggen von Konsolenanwendungen vor.

www.dfz.media/4wh5yh

Video 2.1: Erste Schritte mit dem Debugger

2.4 Methoden debuggen

Schauen wir uns jetzt noch an, wie Sie Methoden schrittweise debuggen. Legen Sie bitte ein neues Projekt für eine Konsolenanwendung an und geben Sie dort den Quelltext aus dem Code 1.4 ein. Alternativ können Sie auch das Projekt **Debug02** aus dem heftbezogenen Download-Bereich Ihrer Online-Lernplattform benutzen.

Lassen Sie dann den Quelltext schrittweise ausführen, bis die Zeile mit dem Aufruf der Methode `Subtraktion()` markiert ist.

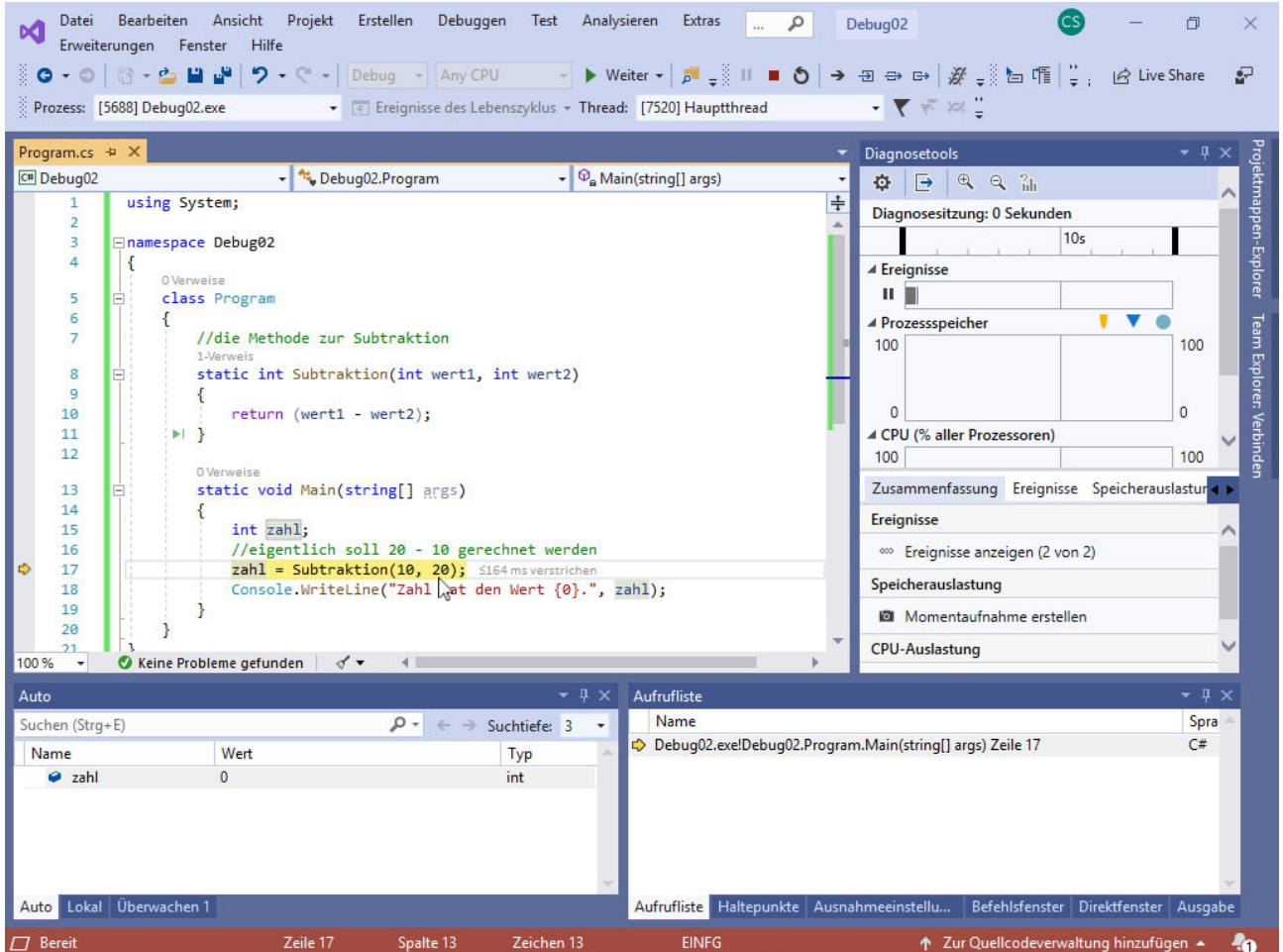


Abb. 2.5: Der markierte Aufruf der Methode `Subtraktion()` (in der Mitte der Abbildung am Mauszeiger)

Um jetzt auch die Anweisungen der Methode `Subtraktion()` schrittweise auszuführen, rufen Sie einfach noch einmal die Funktion **Einzelschritt** auf. Der Debugger springt dann in die Methode und markiert die erste Zeile.

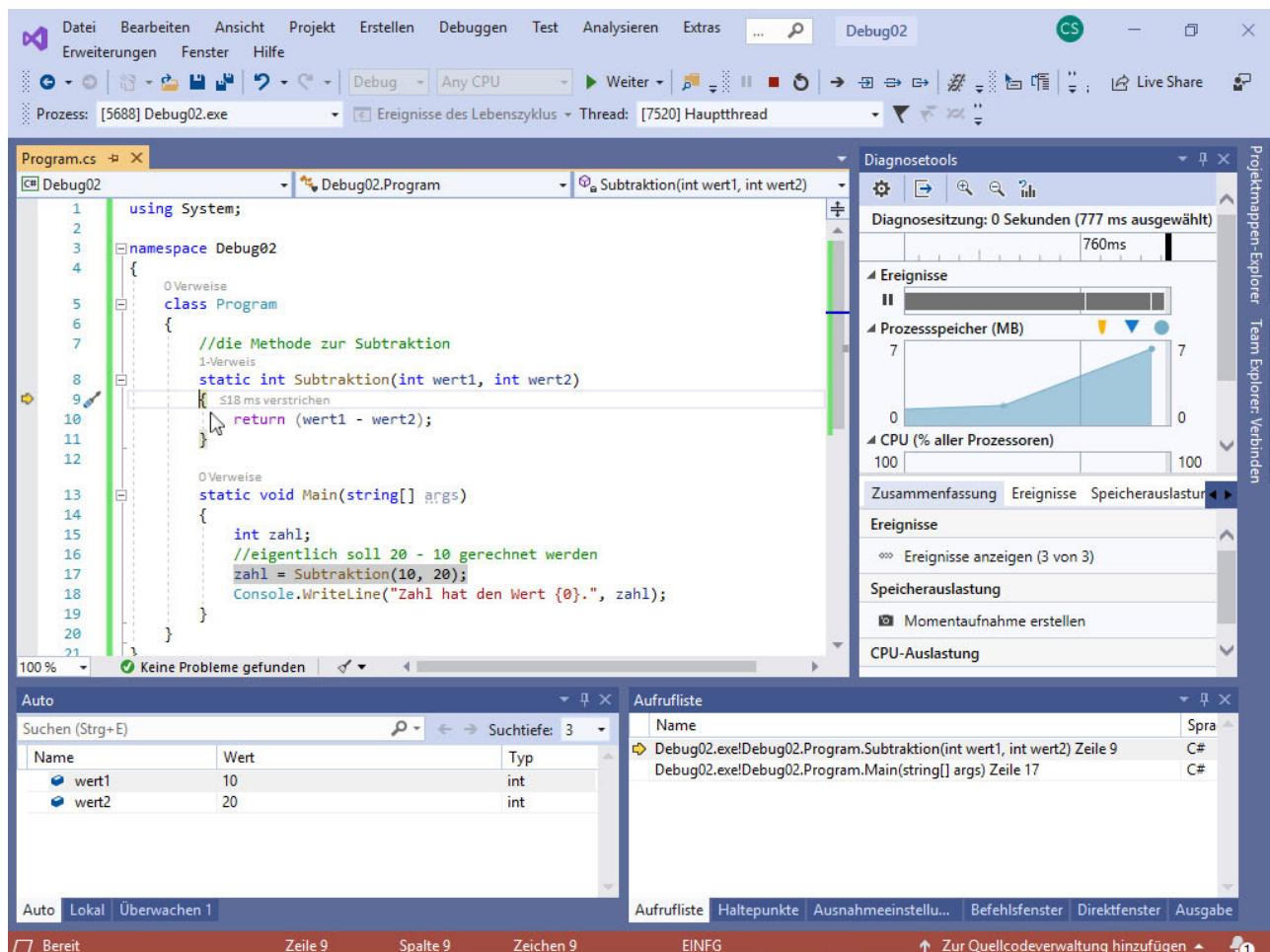


Abb. 2.6: Die markierte Zeile in der Methode (in der Mitte der Abbildung am Mauszeiger)

Diese Zeilen können Sie nun ebenfalls ausführen lassen, bis Sie das Ende der Methode erreichen und die Zeile mit der schließenden Klammer } markiert wird. Beim nächsten Einzelschritt wechselt der Debugger wieder zurück in die Funktion `Main()` und führt das Programm weiter aus.

Probieren Sie das bitte einmal aus. Lassen Sie die Anweisungen in der Methode schrittweise ausführen, bis der Debugger wieder in die Methode `Main()` wechselt.

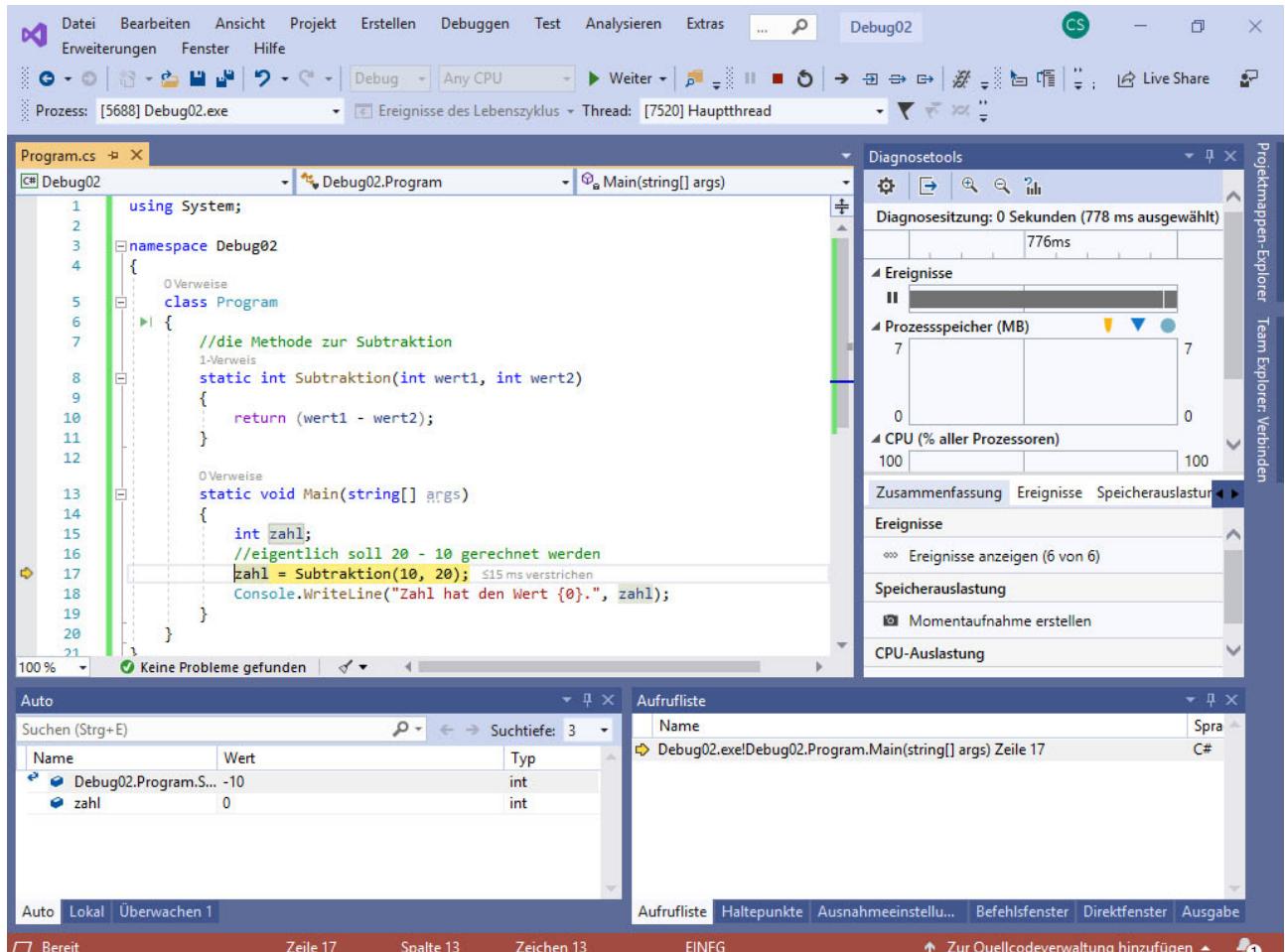


Abb. 2.7: Die Fortsetzung des Programms in der Methode Main()

Der gelbe Pfeil steht jetzt immer noch vor der Zeile mit dem Aufruf der Methode `Subtraktion()`. Das liegt daran, dass nun zwar die Methode ausgeführt wurde, aber noch keine Zuweisung des Ergebnisses an die Variable `zahl` erfolgt ist. Das geschieht erst, wenn Sie noch einmal die Funktion **Einzelschritt** ausführen lassen. Und erst nach diesem zweiten Einzelschritt wird auch der Rückgabewert der Methode korrekt angezeigt.

Neben dem schrittweisen Ausführen können Sie eine Methode auch komplett im Debugger ausführen lassen. Dazu benutzen Sie die Funktion **Prozedurschritt**. Sie starten sie entweder über den entsprechenden Eintrag im Menü **Debuggen**, über das Symbol **Prozedurschritt** in der Symbolleiste **Debuggen** oder mit der Funktionstaste **F10**. Dann wird die gesamte Zeile mit dem Aufruf der Methode sozusagen in einem Rutsch verarbeitet.

Probieren Sie den Unterschied zwischen einem Einzelschritt und einem Prozedurschritt bitte aus. Brechen Sie das Debuggen für die Anwendung ab. Lassen Sie dann das Programm schrittweise ausführen, bis die Zeile mit dem Aufruf der Methode `Subtraktion()` markiert ist und rufen Sie die Funktion **Prozedurschritt** auf. Sie werden sehen, der Debugger führt die komplette Methode aus, weist das Ergebnis zu und markiert die nächste Zeile in der Methode `Main()`.

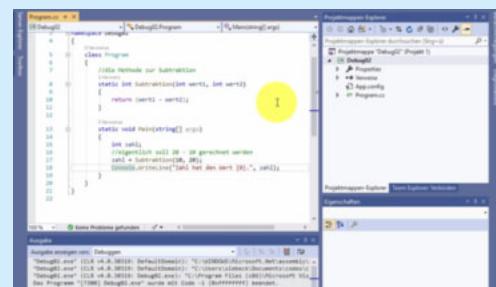


Merken Sie sich:

Wenn Sie die Anweisungen in einer Methode schrittweise ausführen lassen wollen, müssen Sie beim Aufruf der Methode die Funktion **Einzelschritt** benutzen. Wollen Sie dagegen die gesamte Methode ausführen lassen, benutzen Sie die Funktion **Prozedurschritt**.



dfz.media/kdc360



In diesem Video zeigen wir Ihnen, wie Sie Methoden debuggen.

www.dfz.media/kdc360

Video 2.2: Methoden debuggen

Hinweis:

In der Mitte der Symbolleiste von Visual Studio finden Sie ein Kombinationsfeld, über das Sie zwischen den Einstellungen **Debug** und **Release** umschalten können. Diese Einstellungen beziehen sich auf die Projektmappenkonfiguration und legen unter anderem fest, ob besondere Informationen zur Fehlersuche gespeichert werden. Dazu benutzen Sie die Standardeinstellungen **Debug**. Bei der Einstellung **Release** werden diese Informationen nicht gespeichert und der Code zusätzlich optimiert. Diese Einstellung sollten Sie daher für die endgültige Version eines Projekts verwenden.

So viel zum Debuggen von Konsolenanwendungen. Im nächsten Kapitel werden wir uns mit dem Debuggen von Windows Forms-Anwendungen beschäftigen.

Zusammenfassung

Mit dem integrierten Debugger können Sie ein Programm schrittweise ausführen lassen und sich dabei die Werte von Datenobjekten ansehen.

Sie können Änderungen an einem Quelltext direkt im Debug-Modus vornehmen.

Aufgaben zur Selbstüberprüfung

- 2.1 Was unterscheidet die Funktionen **Einzelschritt** und **Prozedurschritt** beim Debuggen?

- 2.2 Wie können Sie sich beim Debuggen möglichst einfach den aktuellen Wert einer Variablen anzeigen lassen?

- 2.3 Sie untersuchen einen Quelltext im Debugger und wollen nun eine Methode zeilenweise ausführen lassen. Welche Funktionstaste drücken Sie an der Aufrufstelle der Methode?

3 Erweiterte Debugging-Funktionen

In diesem Kapitel zeigen wir Ihnen, wie Sie Windows Forms-Anwendungen debuggen, und stellen Ihnen einige erweiterte Debugger-Funktionen vor.

3.1 Windows Forms-Anwendungen debuggen

Beginnen wir mit dem Debuggen von Windows Forms-Anwendungen. Als Beispiel benutzen wir das Programm für einen sehr einfachen Taschenrechner, das Sie bereits erstellt haben, und bauen absichtlich einen Fehler in die Anwendung ein. Sie finden das entsprechende Projekt im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **TaschenrechnerFehler**.

Lassen Sie das Projekt bitte ausführen und probieren Sie ein paar Beispielrechnungen aus. Sie werden sehen, bei der Addition und der Subtraktion stimmen die Ergebnisse nicht.



Bevor Sie weiterlesen:

Stellen Sie einmal selbst Vermutungen an, wo der Fehler liegen könnte. Testen Sie dazu ein paar Berechnungen mit der laufenden Anwendung.

Ganz offensichtlich stimmt etwas mit der Auswertung der Rechenoperation nicht. Bei der Addition wird subtrahiert und bei der Subtraktion dagegen ist das Ergebnis immer Null. Nur die Ergebnisse für die Multiplikation und die Division stimmen.

Gehen wir dem Geheimnis mit dem Debugger auf den Grund. Starten Sie den Debugger mit der Funktion **Einzelschritt**.

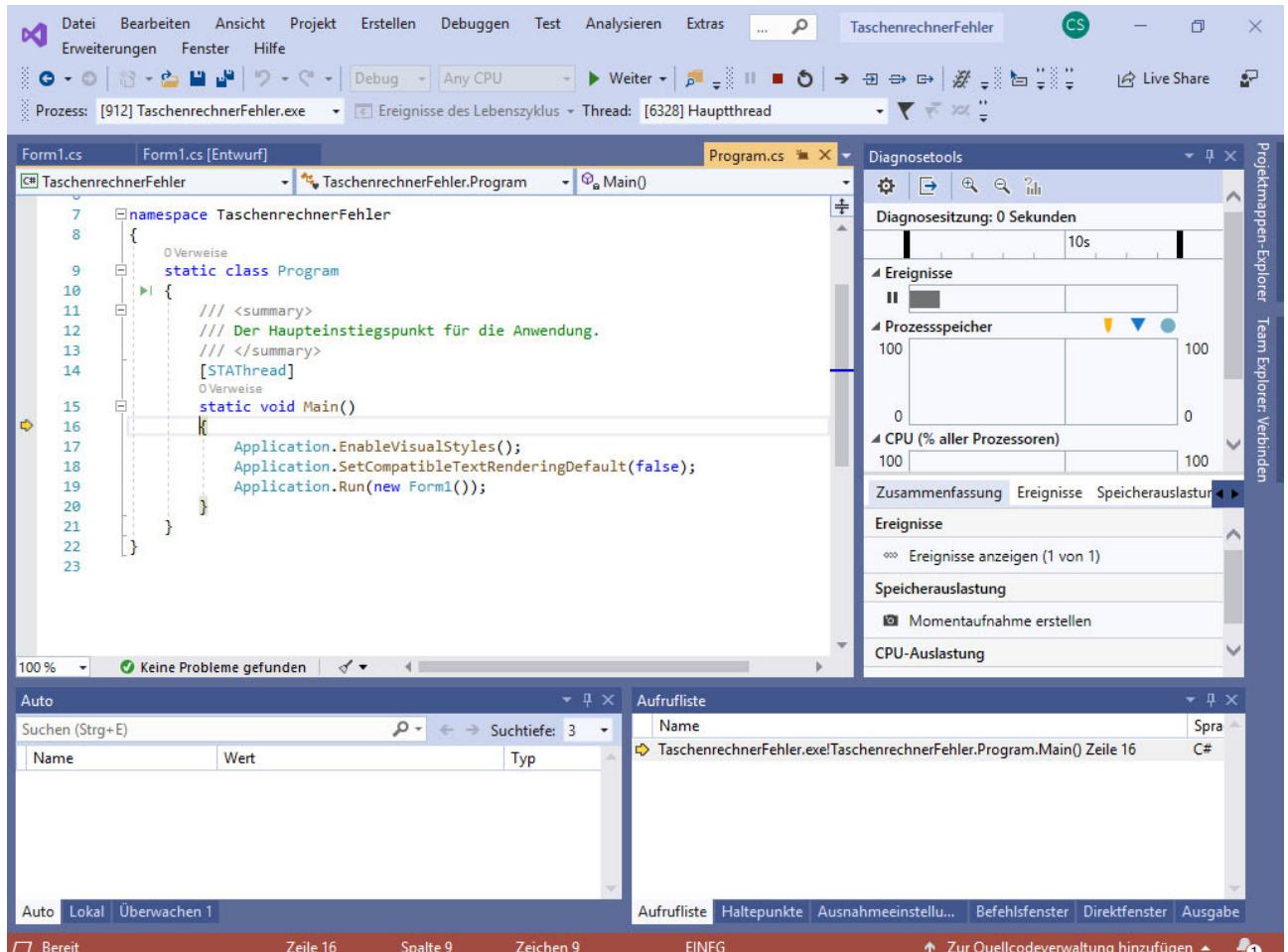


Abb. 3.1: Die erste Anweisung einer Windows Forms-Anwendung im Debugger

Jetzt erscheint allerdings nicht die erste Anweisung unseres Programms, sondern der Quelltext **Program.cs** des Projekts. Dieser Quelltext bildet den Rahmen der eigentlichen Anwendung. Hier werden die visuellen Effekte aktiviert, das Formular **Form1** erzeugt und die Anwendung schließlich ausgeführt. Das übernehmen die Zeilen

```

Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new Form1());
    
```

Was sich genau hinter den beiden ersten Anweisungen verbirgt, wollen wir uns hier nicht weiter ansehen. Interessant für das Debuggen ist vor allem die Anweisung

```
Application.Run(new Form1());
```

Hier wird eine Methode aufgerufen, die die Anwendung ausführt. Als Argument wird dabei eine neue Instanz der Klasse **Form1** übergeben. Wenn wir diese Methode mit der Funktion **Einzelsschritt** debuggen, erscheint auch der eigentliche Quelltext des Taschenrechners.

Probieren Sie das bitte aus. Rufen Sie so oft die Anweisung **Einzelschritt** auf, bis die Zeile

```
Application.Run(new Form1());  
ausgeführt wird.
```

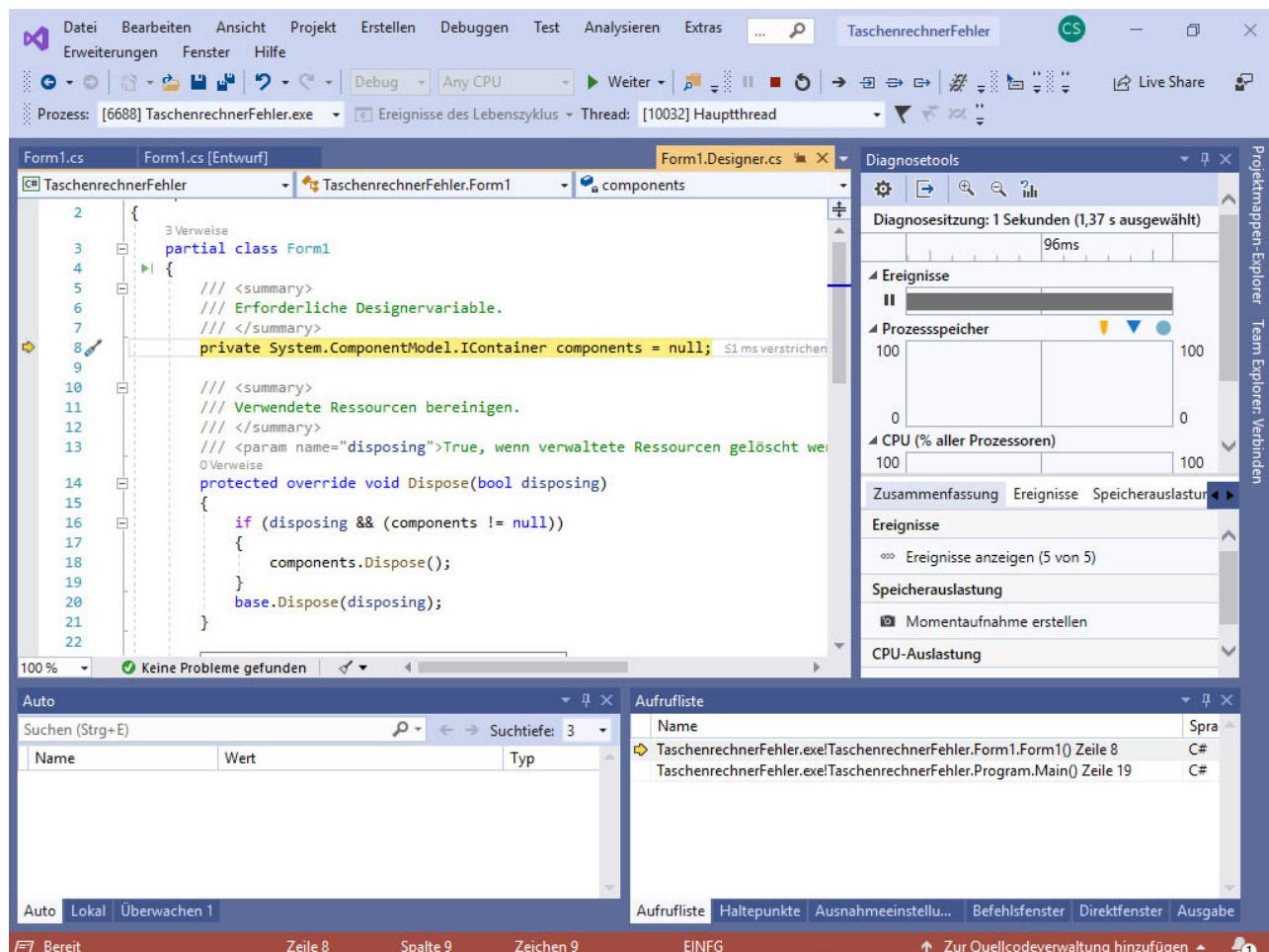


Abb. 3.2: Die erste Anweisung von Form1 im Debugger

Jetzt wird zunächst einmal ein Feld vereinbart. Die nächste Anweisung führt dann den Konstruktor von `Form1` aus. Er initialisiert unter anderem die verschiedenen Steuer-elemente im Formular über die Methode `InitializeComponent()`. Da in dieser Methode nichts passiert, was direkt mit dem Problem in der Anwendung zusammenhängen könnte, lassen Sie diese Methode bitte als Prozedurschritt ausführen. Verlassen Sie anschließend den Konstruktor über die Funktion **Einzelschritt**.

Der Debugger wechselt dann wieder zurück in den Quelltext `Program.cs`. Da gerade ja lediglich der Konstruktor der Klasse `Form1` ausgeführt wurde, steht die Markierung für die nächste Anweisung immer noch vor der Zeile

```
Application.Run(new Form1());
```

Beim nächsten Einzel- beziehungsweise Prozedurschritt würde nun auch endlich die Anwendung ausgeführt.

Probieren Sie auch das bitte aus. Geben Sie zwei Zahlen in den Taschenrechner ein und klicken Sie auf die Schaltfläche **Berechnen**.

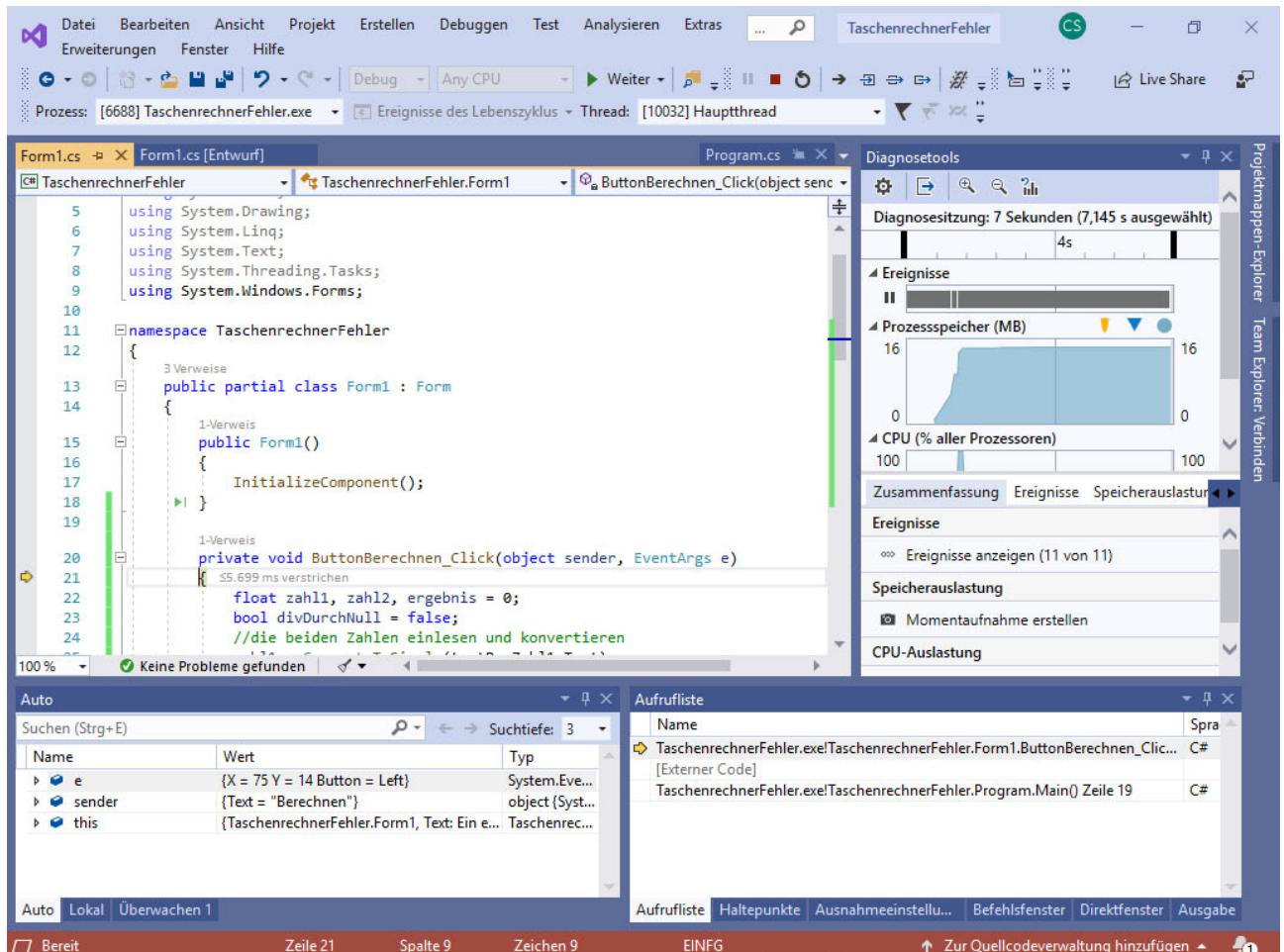


Abb. 3.3: Der Debugger mit der Methode `ButtonBerechnen_Click()`

Lassen Sie dann die Anweisungen weiter ausführen, bis die `if`-Anweisungen zur Auswertung der Rechenoperationen erscheinen. Hier wird jetzt auch das Problem deutlich: Mit den Zeilen

```
if (radioButtonAddition.Checked == true)
    ergebnis = zahl1 + zahl2;
if (radioButtonAddition.Checked == true)
    ergebnis = zahl1 - zahl2;
```

wird bei der Rechenart Addition sowohl addiert als auch subtrahiert. Die Abfrage für die Subtraktion dagegen fehlt ganz. Die zweite `if`-Anweisung muss also korrekt lauten:

```
if (radioButtonSubtraktion.Checked == true)
    ergebnis = zahl1 - zahl2;
```

Korrigieren Sie den Fehler und lassen Sie das Programm neu ausführen. Jetzt sollte es auch wieder korrekt funktionieren.

3.2 Arbeiten mit Haltepunkten

Neben dem einzelnen Ausführen aller Anweisungen in einer Anwendung können Sie den Debugger auch anweisen, das Programm bis zu einer bestimmten Zeile im Quelltext auszuführen und dann die Ausführung anzuhalten. Dazu verwenden Sie die Funktion **Ausführen bis Cursor**. Sie versteckt sich im Kontextmenü einer Quelltextzeile.

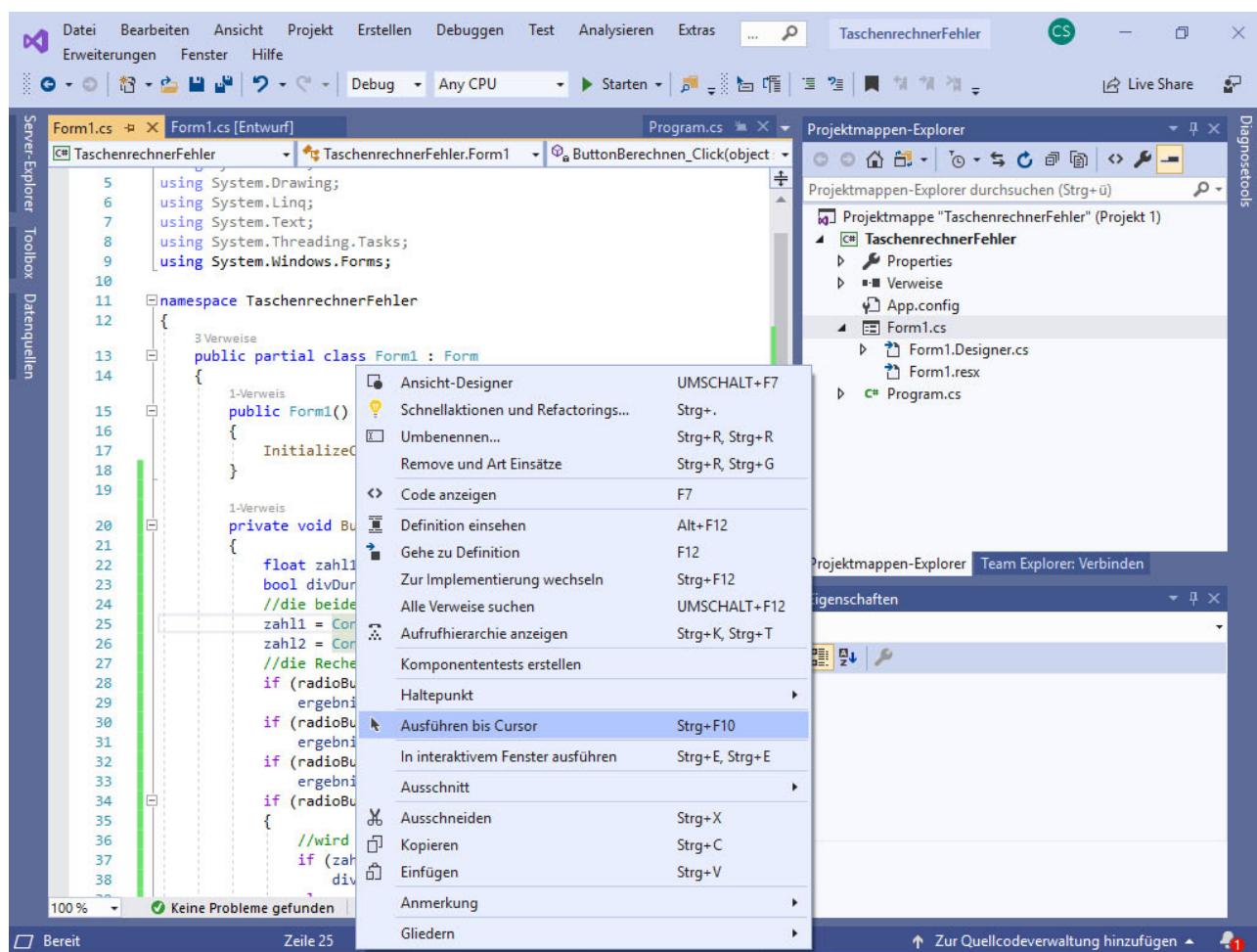


Abb. 3.4: Die Funktion **Ausführen bis Cursor** im Kontextmenü einer Quelltextzeile

Probieren Sie die Funktion jetzt einmal aus. Wechseln Sie – falls erforderlich – über das entsprechende Register oben im Editor in die Datei **Form1.cs**. Verschieben Sie dann den Quelltext, bis die Methode `ButtonBerechnen_Click()` angezeigt wird. Klicken Sie anschließend mit der rechten Maustaste an eine beliebige Stelle in der ersten Anweisung der Methode und wählen Sie im Kontextmenü der Zeile den Eintrag **Ausführen bis Cursor**.

Geben Sie anschließend im Formular zwei Werte ein und klicken Sie auf die Schaltfläche **Berechnen**.

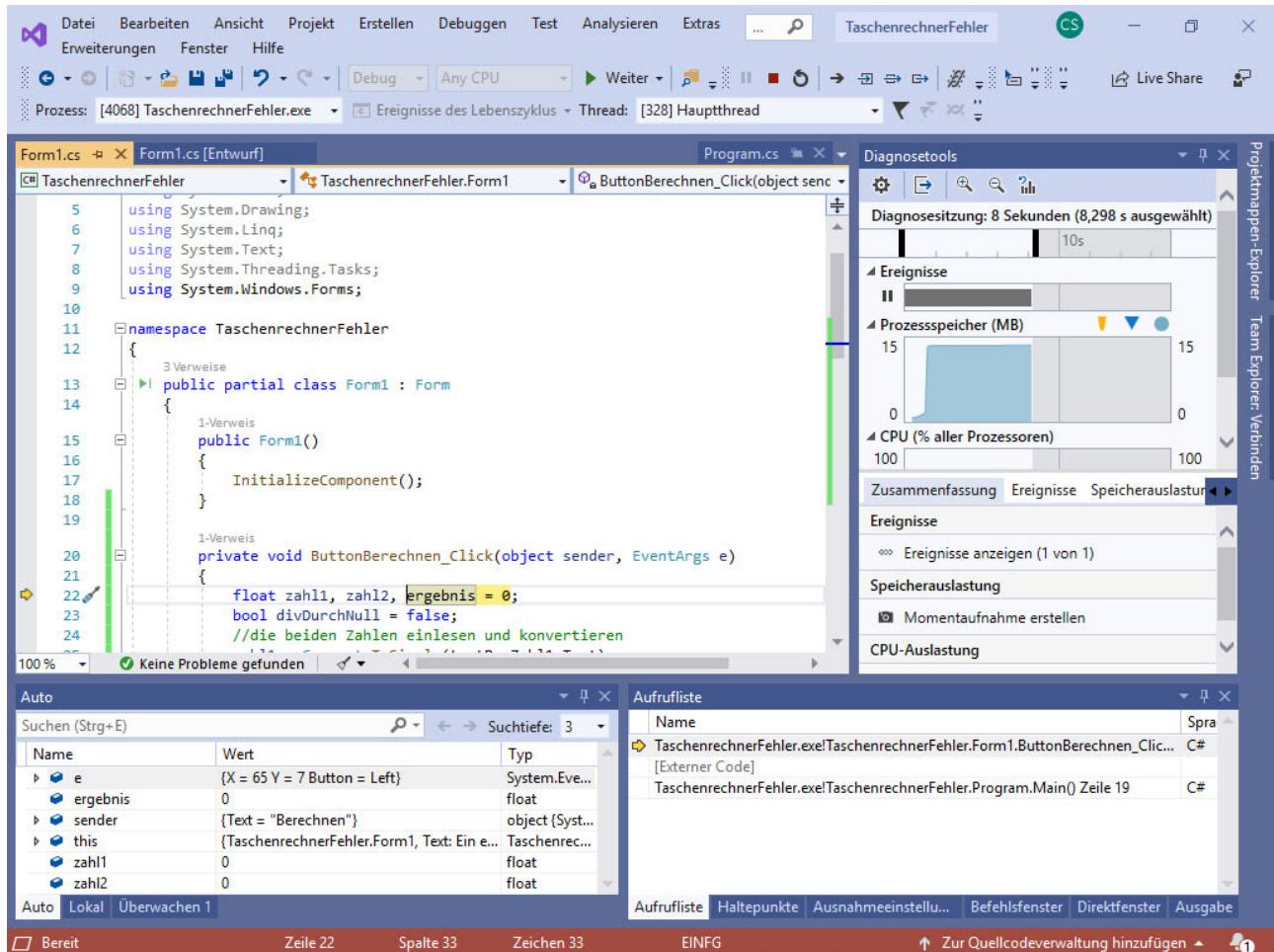


Abb. 3.5: Das angehaltene Programm

Die Ausführung des Programms wird jetzt vor der Zeile mit der Vereinbarung der Variablen unterbrochen. Visual Studio wechselt dabei automatisch in den Debug-Modus. Sie können das Programm jetzt also wie gewohnt weiter in Einzelschritten oder Prozedurschritten testen.

Hinweis:

Unter Umständen bleibt das Formular der Anwendung nach dem Anhalten des Programms weiter im Vordergrund stehen. Sie müssen dann selbst in den Editor von Visual Studio wechseln, um die Anwendung weiter zu debuggen.

Neben der Ausführung bis zu einer bestimmten Position im Quelltext können Sie zusätzlich auch **Haltepunkte** – im Fachjargon **Breakpoints** genannt – festlegen. An diesen Haltepunkten wird die Ausführung des Programms immer wieder unterbrochen. Das ist zum Beispiel dann interessant, wenn Sie das Verhalten in einem bestimmten Teil einer Verzweigung testen möchten.

Am einfachsten setzen Sie einen Haltepunkt, indem Sie die Einfügemarke in die gewünschte Zeile stellen und dann die Funktionstaste **F9** drücken. Alternativ können Sie auch mit der Maus in die Spalte ganz links vor der Zeile klicken.

Tipp:

Wenn Sie lieber mit Menüs arbeiten, finden Sie die Funktion zum Setzen eines Haltepunkts im Menü **Debuggen** unter **Haltepunkt umschalten** oder im Kontextmenü einer Zeile unter **Haltepunkt/Haltepunkt einfügen**.

Nach dem Setzen eines Haltepunkts markiert der Editor die entsprechende Zeile mit einer Art rotem Kreis links vor der Anweisung und hinterlegt außerdem die Anweisung farbig.

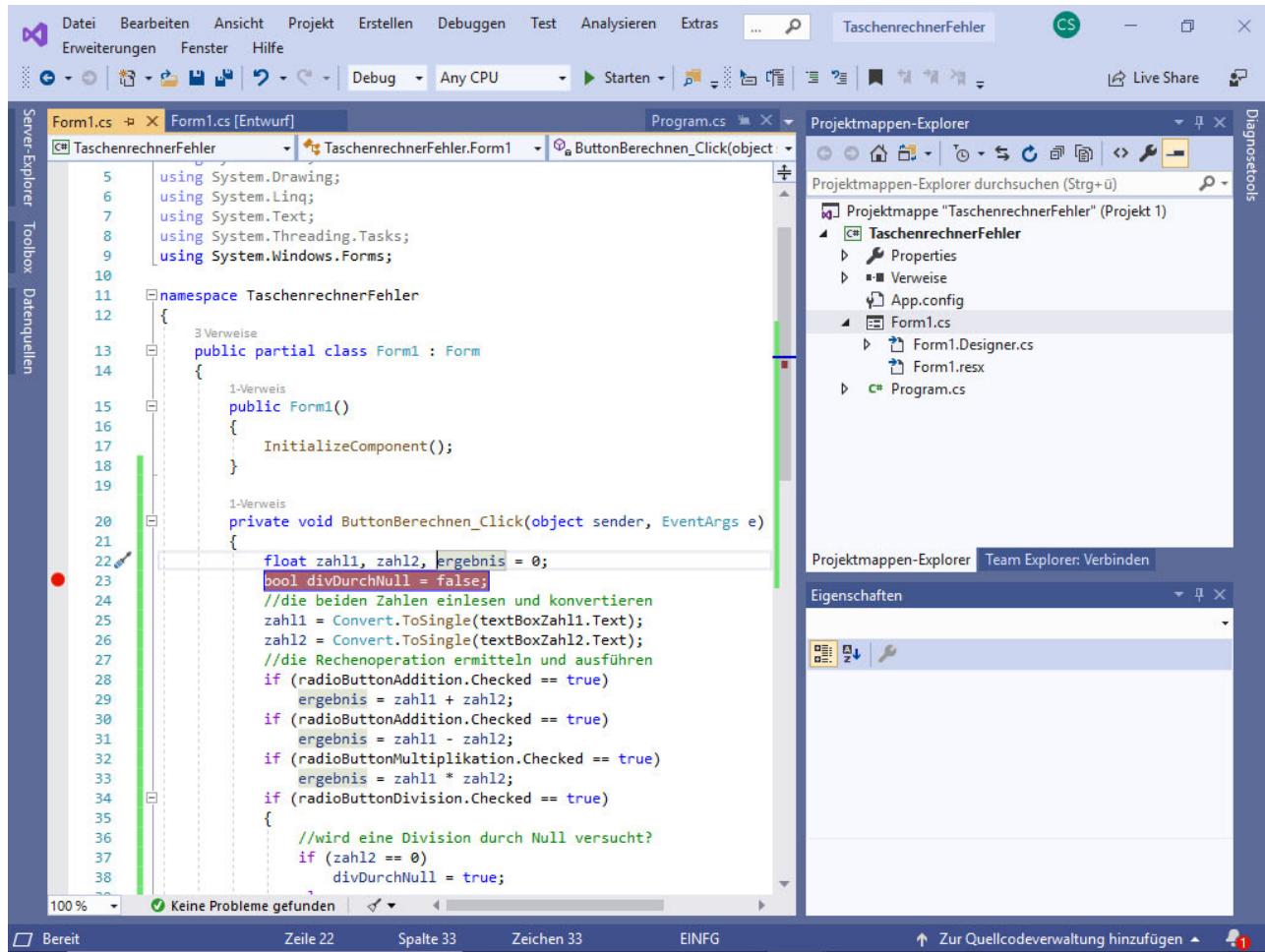


Abb. 3.6: Ein gesetzter Haltepunkt

Probieren Sie das jetzt bitte aus. Setzen Sie einen Haltepunkt an einer beliebigen Stelle in der Methode `ButtonBerechnen_Click()` – zum Beispiel in einer der `if`-Anweisungen.

Nachdem Sie den Haltepunkt gesetzt haben, starten Sie das Programm mit der Funktion **Debugging starten** im Menü **Debuggen**. Alternativ können Sie auch auf das Symbol **Starten** klicken oder die Funktionstaste **F5** drücken.

Bitte beachten Sie:

Wenn Sie das Programm mit der Funktion **Starten ohne Debugging** oder über die Tastenkombination **Strg + F5** ausführen lassen, werden Haltepunkte schlicht und einfach ignoriert.



Sie werden sehen, das Programm wird so lange ausgeführt, bis der Haltepunkt erreicht ist. Danach wechselt Visual Studio automatisch in den Editor.

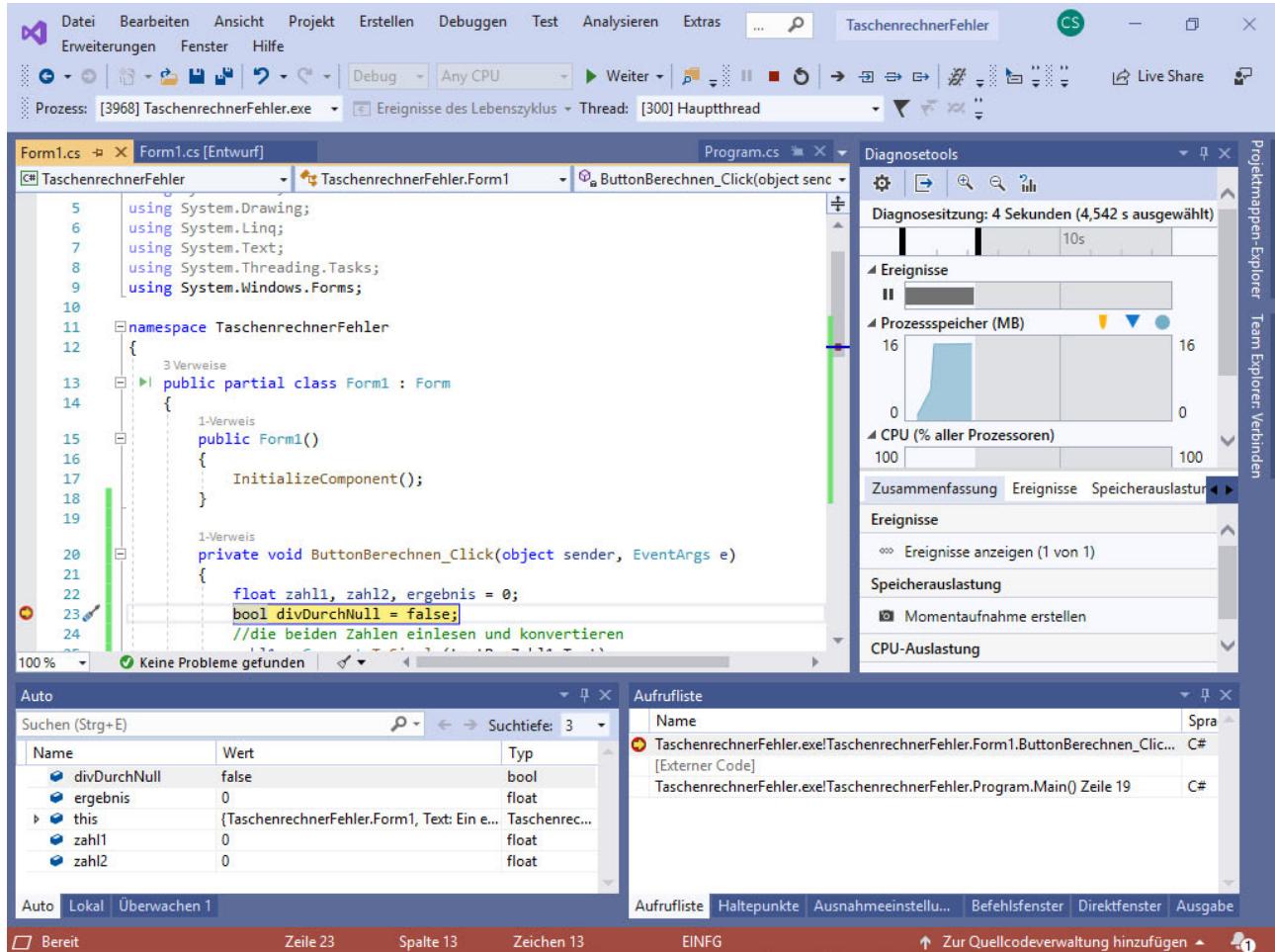


Abb. 3.7: Der erreichte Haltepunkt

Sie können das Programm jetzt entweder schrittweise prüfen oder mit der Funktion **Weiter** im Menü **Debuggen** beziehungsweise über das Symbol weiter ausführen lassen. Dabei läuft das Programm so lange, bis der Haltepunkt erneut erreicht wird.

Um einen gesetzten Haltepunkt wieder zu löschen, stellen Sie die Einfügemarke in die entsprechende Zeile des Quelltextes. Drücken Sie dann noch einmal die Funktionstaste **F9** oder klicken Sie mit der Maus ganz links in die Spalte vor der Zeile. Alternativ können Sie auch die Funktion **Haltepunkt umschalten** im Menü **Debuggen** beziehungsweise die Funktion **Haltepunkt/Haltepunkt umschalten** im Kontextmenü der Zeile aufrufen.

Tipp:

Sie können einen Haltepunkt auch deaktivieren. Dazu benutzen Sie die Funktion **Haltepunkt deaktivieren** im Kontextmenü des Symbols für den Haltepunkt oder klicken Sie auf das Symbol **Haltepunkt deaktivieren**, das erscheint, wenn Sie den Mauszeiger auf das Symbol für den Haltepunkt stellen. Ein deaktivierter Haltepunkt wird nicht gelöscht, sondern lediglich abgeschaltet. Sie können ihn jederzeit wieder mit der Funktion **Haltepunkt aktivieren** im Kontextmenü des Symbols oder über das Symbol **Haltepunkt aktivieren** erneut aktivieren. Ob ein Haltepunkt im Moment aktiv ist oder nicht, erkennen Sie an dem Symbol. Für einen aktiven Haltepunkt wird ein gefüllter Kreis angezeigt, für einen deaktivierten Haltepunkt lediglich ein nicht gefüllter Kreis.



In diesem Video zeigen wir Ihnen, wie Sie Haltepunkte setzen und löschen.

www.dfz.media/3jhbu0

Video 3.1: Haltepunkte setzen und löschen

Sie können Haltepunkte auch mit Bedingungen verknüpfen oder erst nach einer bestimmten Trefferanzahl auslösen lassen. Mehr zu diesen und noch weiteren Funktionen finden Sie in der Hilfe im Dokument **Debuggen in Visual Studio**.

So viel zu den Haltepunkten.

3.3 Weitere nützliche Debugger-Funktionen

Sehen wir uns nun an, wie Sie Ausdrücke dauerhaft in einem eigenen Fensterbereich anzeigen lassen können. Das ist zum Beispiel dann interessant, wenn Sie den Wert einer Variablen oder eines Feldes während des gesamten Programmablaufs beobachten wollen. Sie müssen dann den aktuellen Wert nicht immer wieder durch das Positionieren des Mauszeigers auf ein Vorkommen im Quelltext abrufen.

Die Werte von lokalen Datenobjekten werden links unten im Register **Lokal** oder im Register **Auto** angezeigt. Allerdings erscheinen diese Register nur im Debug-Modus.

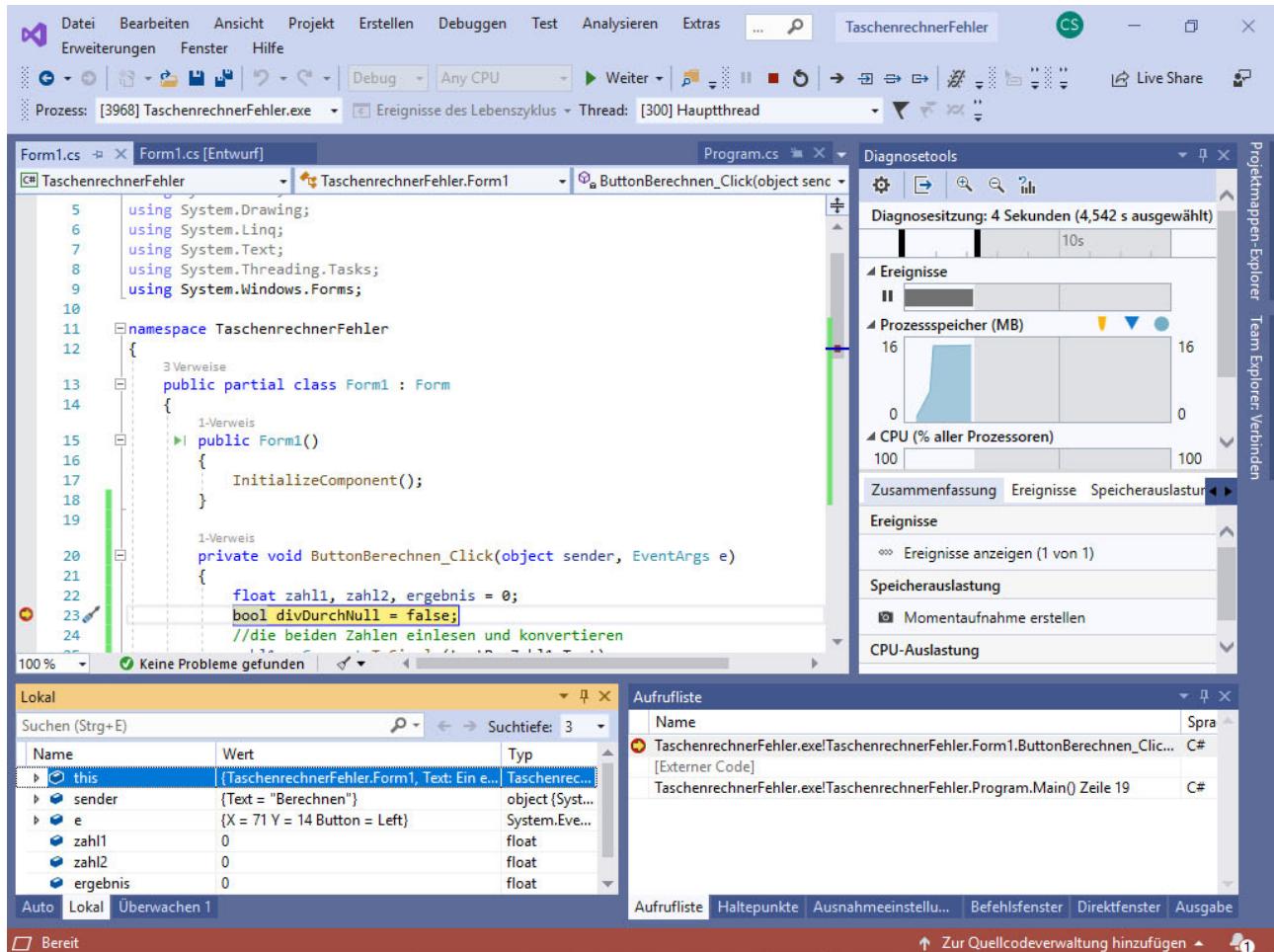


Abb. 3.8: Das Register **Lokal** (unten links in der Abbildung)

Hinweis:

Im Register **Auto** werden vor allem die Datenobjekte aus der aktuellen und der vorigen Codezeile angezeigt. Im Register **Lokal** dagegen erscheinen alle lokalen Datenobjekte.

Probieren Sie das einfach einmal aus. Setzen Sie in die erste Zeile der Methode `ButtonBerechnen_Click()` einen Haltepunkt. Führen Sie das Programm dann bis zu diesem Haltepunkt aus und wechseln Sie in das Register **Lokal** links unten im Fenster von Visual Studio.

Neben den Werten der vier lokalen Variablen werden in dem Register dann außerdem die Werte der Datenobjekte `this`, `sender` und `e` angezeigt. `this` steht dabei für die aktuelle Instanz der Klasse `Form1`. Die Werte für `sender` und `e` werden automatisch an die Methode übergeben.

Zur Auffrischung:

Der Parameter `sender` steht für das Steuerelement, das das Ereignis ausgelöst hat, und der Parameter `e` enthält weitere Informationen – zum Beispiel, welche Maustaste gedrückt wurde.



Durch einen Mausklick auf das Plussymbol vor einem Eintrag können Sie auch noch weitere Informationen anzeigen lassen – zum Beispiel zum Mausereignis.

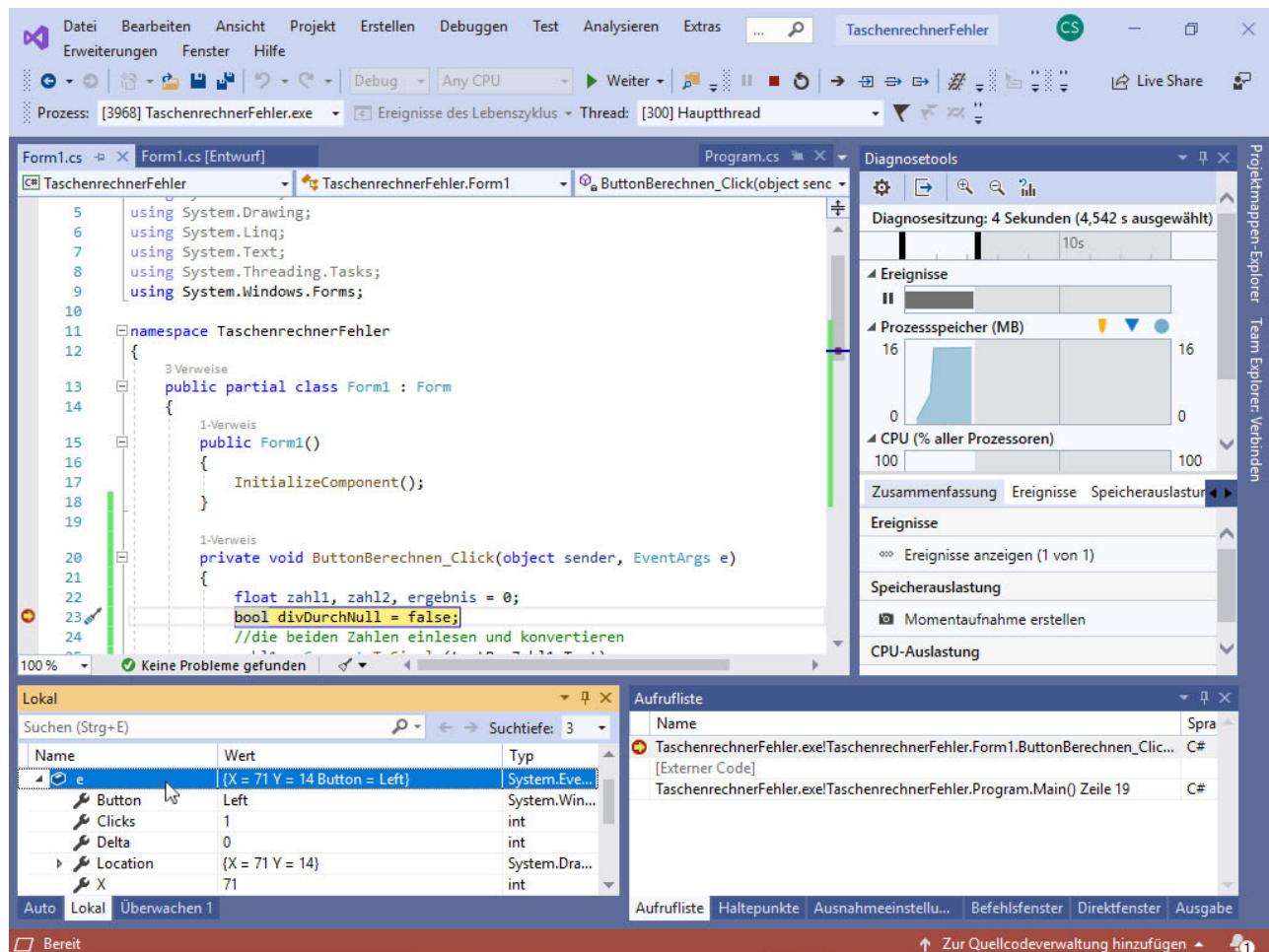


Abb. 3.9: Die Anzeige weiterer Informationen für das Datenobjekt e (unten in der Abbildung am Mauszeiger)

Über die Funktion **Überwachung hinzufügen** im Kontextmenü eines Bezeichners können Sie den Wert des Datenobjekts auch im Register **Überwachen1** anzeigen lassen. Sie finden es rechts neben dem Register **Lokal**.

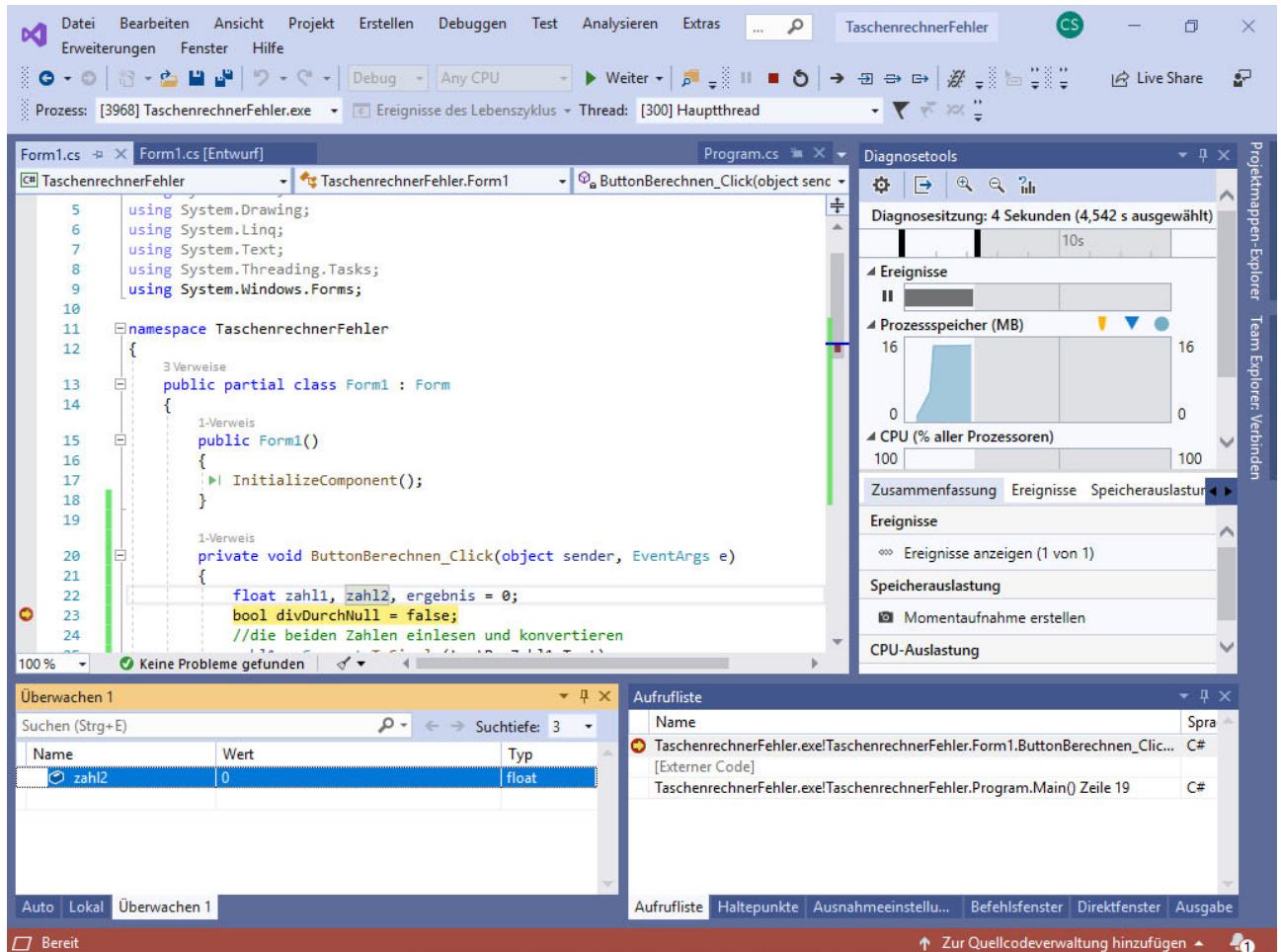


Abb. 3.10: Die Anzeige eines Wertes im Register **Überwachen 1** (unten links in der Abbildung)

Neben dem Überwachen eines Ausdrucks können Sie einem Ausdruck auch beim laufenden Programm selbst einen Wert zuweisen. Damit lässt sich zum Beispiel überprüfen, wie die Anwendung reagiert, wenn ein Datenobjekt einen bestimmten Wert hat.

Schauen wir uns auch dazu ein Beispiel an:

Lassen Sie den Taschenrechner bitte schrittweise weiter ausführen, bis die `if`-Anweisungen erreicht werden. An dieser Stelle haben die beiden Variablen `zahl1` und `zahl2` den Wert, den Sie in die entsprechenden Eingabefelder eingetragen haben. Den Wert der Variablen `zahl2` wollen wir jetzt einmal über den Debugger verändern.

Wechseln Sie in das Register **Lokal** unten links im Fenster von Visual Studio. Doppelklicken Sie dann auf den angezeigten Wert der Variablen `zahl2`.

Bitte beachten Sie:

Sie müssen in die Spalte **Wert** des Datenobjekts klicken. Ein Doppelklick in eine der anderen Spalten funktioniert nicht.



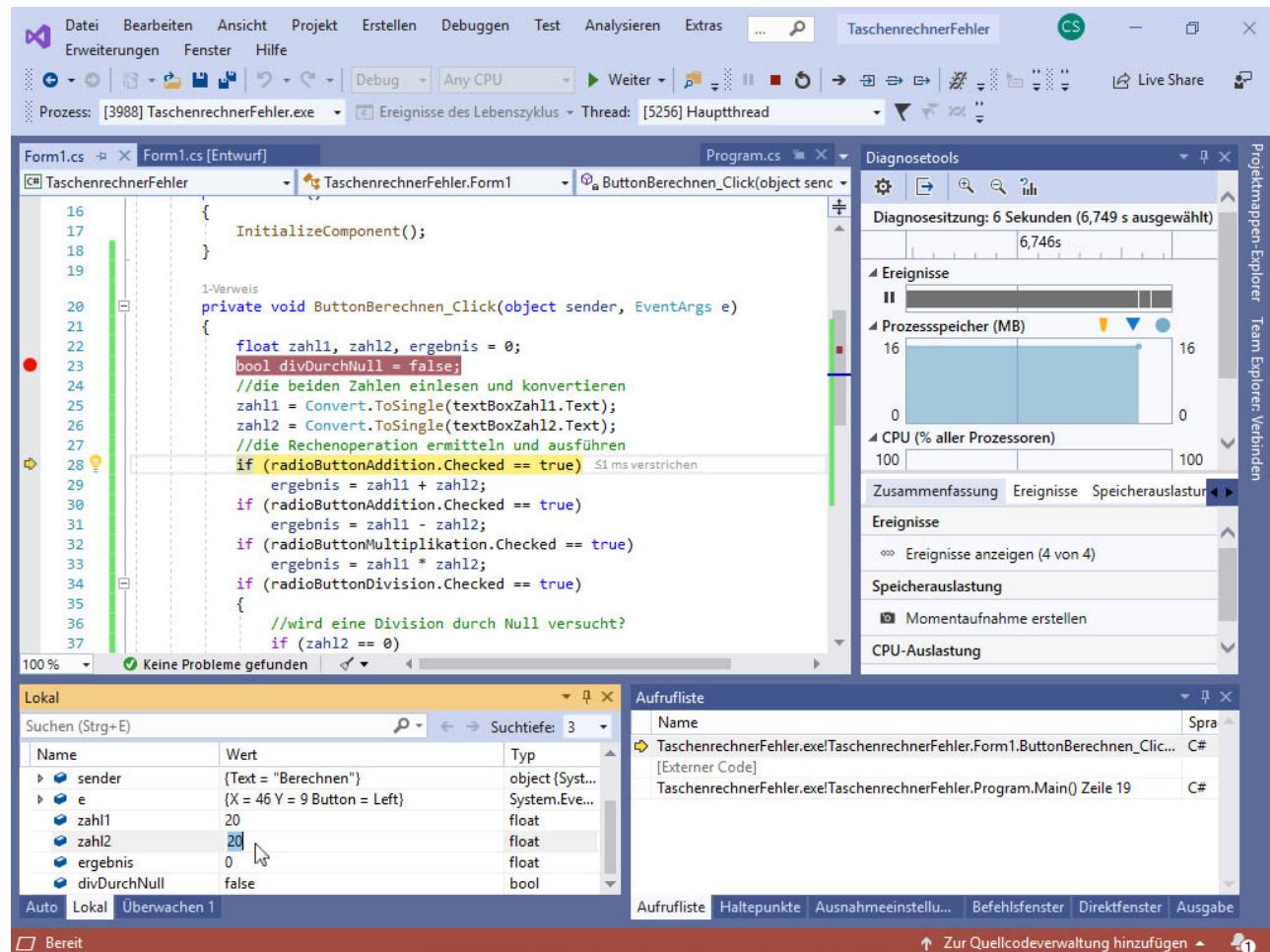


Abb. 3.11: Der Wert von zahl2 ist zum Ändern markiert (unten in der Abbildung am Mauszeiger)

Visual Studio markiert jetzt nur noch den Wert und zeigt Ihnen durch die blinkende Einfügemarkie am Ende der Markierung an, dass Sie den Wert überschreiben können. Geben Sie nun einen beliebigen neuen Wert für zahl2 ein und drücken Sie dann die Eingabetaste.

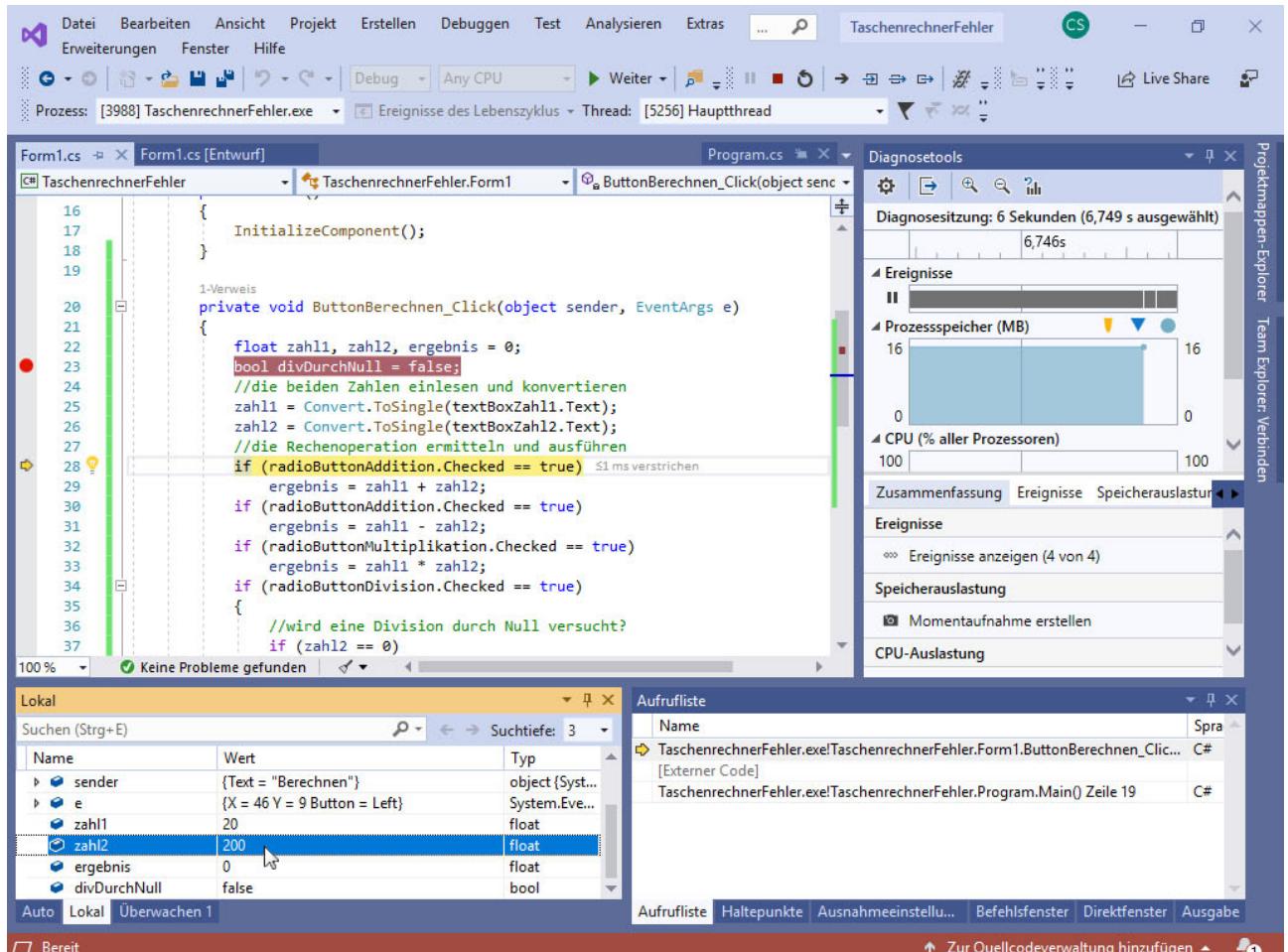
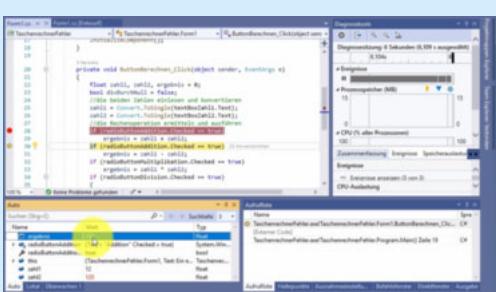


Abb. 3.12: Der geänderte Wert für zahl2 (unten in der Abbildung am Mauszeiger)

Lassen Sie anschließend das Programm weiter schrittweise ausführen. Sie werden sehen, Visual Studio rechnet mit dem Wert weiter, den Sie im Register **Lokal** eingegeben haben.

Tipp:

Das Ändern eines Wertes funktioniert auch direkt über den DataTip. Lassen Sie dazu zuerst den DataTip des Datenobjekts anzeigen und klicken Sie dann mit der Maus an eine beliebige Stelle im DataTip. Danach können Sie den Wert direkt im DataTip überschreiben.



In diesem Video stellen wir Ihnen weitere nützliche Debuggerfunktionen vor.



www.dfz.media/06egg9

Video 3.2: Nützliche Debuggerfunktionen

Zusammenfassung

Mit Haltepunkten können Sie ein Programm bis zu einer bestimmten Stelle ausführen lassen.

Im Register **Lokal** werden die Werte von lokalen Datenobjekten automatisch angezeigt. Sie können auch nahezu beliebige weitere Datenobjekte überwachen lassen.

Den Wert eines überwachten Datenobjekts können Sie in den Registern oder direkt über den DataTip ändern.

Aufgaben zur Selbstüberprüfung

- 3.1 Was unterscheidet einen Haltepunkt von der Funktion **Ausführen bis Cursor**?

- 3.2 Wie können Sie den Wert eines Datenobjekts über den Debugger ändern?

4 Exkurs: Vom richtigen Testen

In diesem Kapitel unternehmen wir einen Ausflug in das richtige Testen von Software.

Selbst wenn Sie hoch konzentriert und sehr sorgfältig arbeiten, werden Ihre Programme fast immer Fehler haben. Das können „Kleinigkeiten“ wie Berechnungsfehler sein oder auch „Katastrophen“ wie ein Absturz in einer bestimmten Situation, an die Sie beim Programmieren nicht gedacht haben. Vollständig vermeiden lassen sich solche Fehler nicht. Sie müssen sie aber finden und natürlich auch korrigieren. Dazu testen Sie Ihre Programme – Sie prüfen die Qualität.

Bei einem Test wird ein Produkt analysiert, mit eindeutigen Vorgaben ausprobiert und sein Verhalten beobachtet.



4.1 Probleme des Testens

Eine Qualitätsprüfung für Software ist allerdings alles andere als einfach – auch wenn es auf den ersten Blick vielleicht nicht so wirkt. Das erste Problem:

Sie müssen beim Test gezielt nach Fehlern suchen und nicht nachweisen, dass Ihr Programm keine Fehler hat. Der Nachweis der Fehlerfreiheit mag bei sehr einfachen Programmen vielleicht noch funktionieren, bei komplexerer Software ist er schlicht und einfach unmöglich. Das liegt allein schon daran, dass es nahezu unendlich viele Varianten für den Ablauf eines Programms gibt.

Nehmen wir noch einmal den sehr simplen Taschenrechner aus dem letzten Kapitel als Beispiel. Direkt nach dem Start des Programms gibt es bereits fünf verschiedene Möglichkeiten, wie das Programm weiter ablaufen könnte:

- 1) Der Anwender klickt direkt auf die Schaltfläche **Berechnen**.
- 2) Der Anwender gibt einen Wert in das erste Feld ein.
- 3) Der Anwender gibt einen Wert in das zweite Feld ein.
- 4) Der Anwender wählt eine andere Rechenart aus.
- 5) Der Anwender beendet das Programm sofort wieder.

Für die ersten vier Möglichkeiten gibt es dann wieder zahlreiche Varianten für den weiteren Programmablauf.

Das zweite Problem: Qualitätsprüfung ist ein eher destruktiver Prozess.

Destruktiv ist das Gegenteil von konstruktiv und bedeutet so viel wie „zerstörerisch“.



Ein guter Prüfer ist hoch motiviert, in einem Produkt Fehler zu finden. Dazu muss er davon überzeugt sein, dass das Produkt fehlerhaft ist – er muss bereit sein, die mangelhafte Qualität zu beweisen. Einem Programmierer fällt solch eine negative Einstellung seiner Software gegenüber natürlich sehr schwer. Er ist davon überzeugt, dass seine Arbeit

brauchbar ist, das Programm korrekt arbeitet und Fehler vor allem durch den „falschen Einsatz“ verursacht werden – also zum Beispiel durch den Anwender („Da hätten Sie jetzt nicht klicken dürfen.“).

Häufig kann ein Programmierer selbst einen Fehler auch gar nicht finden, da er ja davon ausgeht, korrekte Annahmen und Vorgaben zu verwenden. Umstände, die den Fehler auftreten lassen, wird er daher wahrscheinlich auch bei einer Prüfung erst gar nicht berücksichtigen.

Als Programmierer sollten Sie Ihre Werke daher nicht nur allein prüfen. Versuchen Sie – wann immer möglich –, Dritte zu beteiligen oder Testmethoden einzusetzen. Einige dieser Testmethoden wollen wir uns jetzt einmal ansehen.

4.2 Testmethoden

Beginnen wir mit dem „heißen Testen“.

Beim „heißen Testen“ werden so lange willkürlich Daten eingegeben beziehungsweise das Programm wird so lange ausgeführt, bis ein Fehler auftritt – oder eben nicht. In unserem Taschenrechner könnte ein „heißer Test“ so aussehen:

Es werden zwei Zahlen eingegeben und für jede Rechenart einmal durchprobiert. Damit die Prüfung des Ergebnisses möglichst einfach ist, werden nur kleine Zahlen verwendet – zum Beispiel 2 und 3. Danach folgt ein Test mit der Null als zweiter Zahl, um zu überprüfen, ob die Division durch Null abgefangen wird. Ein letzter Test wird dann vielleicht noch mit negativen Zahlen durchgeführt.

Wenn alle Tests die gewünschten Ergebnisse liefern, wird das Programm als fehlerfrei betrachtet.

Besonders brauchbar ist das „heiße Testen“ allerdings in der Regel nicht. Es eignet sich höchstens für einen ersten groben Test direkt nach der Programmierung, aber nicht für eine gründliche Prüfung. Durch das unsystematische Vorgehen werden häufig Fehler übersehen. Außerdem ist oft nicht klar, wodurch ein gefundener Fehler denn nun überhaupt genau entstanden ist.

Sehr viel systematischer gehen funktionsorientierte Testverfahren – auch *Blackbox*-Tests² genannt – vor. Hier wird ein Testobjekt – zum Beispiel ein Programm oder ein Teil eines Programms – mit Daten „gefüttert“ und dann geprüft, ob die Ausgabedaten mit den Daten übereinstimmen, die das Testobjekt eigentlich liefern müsste.

Wie das Testobjekt die Ausgabedaten erzeugt, spielt für den Test zunächst einmal überhaupt keine Rolle. Interessant ist nur das Ergebnis – also die Ausgaben.



Abb. 4.1: Grundprinzip eines funktionsorientierten Tests

2. *Blackbox* bedeutet übersetzt so viel wie „schwarze Schachtel“.

Anders als beim „heißen Testen“ werden für funktionsorientierte Tests **vor** dem eigentlichen Test **Testfälle** erstellt.

Ein Testfall beschreibt die verwendeten Eingabedaten und die erwarteten Ausgaben.



Für die Entwicklung von Testfällen gelten dabei folgende Grundregeln:

- Es müssen sowohl Eingaben als auch Ausgaben abgedeckt werden. Das heißt, Sie müssen festlegen, welche Eingabe zu welcher Ausgabe führt beziehungsweise führen sollte.
- Es müssen sowohl gültige als auch ungültige oder unerwartete Eingaben benutzt werden. Für unseren Taschenrechner müssten Sie also nicht nur Testfälle mit Zahlen erstellen, sondern zum Beispiel auch Testfälle mit Buchstaben. Außerdem benötigen Sie Testfälle mit Zahlen, die Nachkommastellen enthalten.

Die Testfälle für die Addition könnten zum Beispiel so aussehen:

Tab. 4.1: Mögliche Testfälle für die Addition

Testfall	Eingabedaten	erwartete Ausgabe
1	0 und 1	1
2	1 und 0	1
3	A und 0	keine, da A ein ungültiger Datentyp ist beziehungsweise Fehlermeldung
4	0,1 und 1,0	1,1
5	1 und 1	2
6	1 und -1	0
...

Bevor Sie weiterlesen:

Welches Problem sehen Sie beim Erstellen der Testfälle? Überlegen Sie einfach einmal, wie viele Testfälle wir benötigen, um wirklich alle denkbaren Eingaben für die Addition zu testen.



In unserem Beispiel gibt es überhaupt keine definierte Grenze. Wir bräuchten unendlich viele Testfälle, da es ja unendlich viele Zahlen und damit unendlich viele Varianten für unser Programm gibt.

Aber auch bei Tests, die scheinbar nur eine begrenzte Anzahl von Eingaben benötigen, kann die Anzahl der Testfälle schnell enorme Ausmaße annehmen. Wenn Sie beispielsweise die Eingabe einer maximal zweistelligen positiven Zahl durch Testfälle abdecken wollen, benötigen Sie bereits 100 unterschiedliche Varianten – alle Zahlen von 1 bis 99 und die Zahl 0.

Noch größer wird die Anzahl der Testfälle, wenn Sie Buchstaben verarbeiten wollen. Bereits bei der Eingabe von maximal drei Zeichen müssten Sie 26^3 Testfälle bilden – einen für jeden vorhandenen Buchstaben im Alphabet und das für drei Zeichen. Insgesamt sind das also 17 576 Varianten. Dabei sind die Umlaute und das scharfe s sowie Groß- und Kleinschreibung noch nicht einmal berücksichtigt.



Selbst bei einem sehr kleinen Programm lässt sich kaum jede mögliche Variante durch einen eigenen Testfall abdecken.

Deshalb werden für die Testfälle **Äquivalenzklassen** gebildet.



Äquivalenzklassen fassen sehr ähnliche Testfälle, die zu demselben Ergebnis führen, zusammen. Äquivalenz bedeutet so viel wie „Gleichwertigkeit“.

Jede Äquivalenzklasse wird möglichst eindeutig beschrieben und mit Beispieleingabedaten versehen. Für die Addition in unserem Taschenrechner könnten die Äquivalenzklassen zum Beispiel so aussehen:

Tab. 4.2: Äquivalenzklassen für die Addition

Klasse	Beschreibung	Beispieldaten	erwartete Ausgabe
1	Zwei positive Zahlen	10 und 20	30
2	Eine positive und eine negative Zahl	-10 und 20	10
3	Zwei negative Zahlen	-10 und -20	-30
4	Ein oder beide Werte sind ungültig	A und 0 oder A und B	keine beziehungsweise Fehlermeldung

Damit hätten wir nahezu alle möglichen Eingabevarianten mit lediglich vier Testfällen abgedeckt.

So viel zum funktionsorientierten Test.

Eine weitere Variante ist die testgetriebene Entwicklung. Hier werden die Tests vor oder während der eigentlichen Programmierung erstellt. Das Programm wird dann in mehreren Schritten immer wieder getestet und geändert, bis es die Tests besteht.

Neben dynamischen Testverfahren, bei denen das Programm tatsächlich ausgeführt wird, gibt es auch noch **statische Testverfahren**. Hier werden vor allem Dokumente wie zum Beispiel Quelltexte „auf dem Trockenen“ geprüft. Die meisten statischen Testverfahren werden in Teamsitzungen durchgeführt, um einen möglichst objektiven Blick auf das Testobjekt sicherzustellen.

Es gibt aber auch einen statischen Test, den Sie gut im Alleingang durchführen können – den **Schreibtischtest**. Er ist so etwas wie ein Debug-Durchgang „per Hand“: Der Quelltext wird ausgedruckt und dann Zeile für Zeile auf dem Trockenen durchgespielt.

Zwar liefern Schreibtischtests allein keine echten Aussagen, ob ein Programm nun korrekt funktioniert oder nicht. Sie haben aber einen großen Vorteil: Allein dadurch, dass der Code ausgedruckt ist, nehmen Sie oft eine distanziertere Haltung ein, die die Fehler-suche vereinfacht. Denn nicht selten „verbeissen“ sich Programmierer bei der Lösung eines Problems in einen Weg und übersehen Alternativen.

Tipp:

Wenn Sie bei der Suche nach einem Fehler gar nicht weiterkommen, lassen Sie das Programm – wenn möglich – einige Stunden oder Tage einfach liegen. Versuchen Sie, in der Zwischenzeit abzuschalten, und nehmen Sie dann einen neuen Anlauf. Sie werden sehen, Sie gehen oft mit ganz neuen Ideen an die Lösung des Problems heran.

So viel zum richtigen Testen.

Hinweis:

Der Test von Software ist eine Wissenschaft für sich. Weiterführende Informationen zu dem Thema finden Sie in dem Buch von Vigenschow im Literaturverzeichnis.

Sie finden rechts in der Menüleiste von Visual Studio einen Menüeintrag **Test**. Über dieses Menü können Sie verschiedene Funktionen für manuelle und auch automatisierte Tests aufrufen.

Zusammenfassung

Bei einem Test wird ein Programm analysiert, mit eindeutigen Vorgaben ausprobiert und sein Verhalten beobachtet.

Beim „heißen Testen“ werden unsystematisch Daten eingegeben, bis ein Fehler auftritt oder eben nicht.

Funktionsorientierte Testverfahren arbeiten mit Testfällen und Äquivalenzklassen.

Bei statischen Testverfahren werden vor allem Dokumente wie Quelltexte „auf dem Trockenen“ geprüft.

Aufgaben zur Selbstüberprüfung

- 4.1 Welche Aufgabe sollte ein Test haben: den Nachweis der Fehlerfreiheit oder das Aufspüren von Fehlern? Begründen Sie bitte Ihre Antwort.

- 4.2 Was ist ein Testfall?

- 4.3 Was verstehen Sie unter einer Äquivalenzklasse?

5 Ausnahmebehandlung

In diesem Kapitel zeigen wir Ihnen, wie Sie Fehler beziehungsweise die Auswirkungen von Fehlern in Ihren Programmen abfangen können.

Selbst wenn Ihre Codes an sich keine Fehler enthalten, kann es trotzdem vorkommen, dass Ihr Programm nicht korrekt arbeitet – zum Beispiel, weil ein Anwender einen Wert eingibt, den er eigentlich nicht eingegeben dürfen. Das Programm gerät dann in einen unerwarteten Zustand, der zu einem undefinierten Verhalten oder zu einem Absturz führen kann.

Ein Klassiker für solch ein undefiniertes Verhalten ist die Division durch null. Dazu ein einfaches Konsolenprogramm als Beispiel:

```
static void Main(string[] args)
{
    int zahl1, zahl2, ergebnis;
    Console.WriteLine("Zahl 1:");
    zahl1 = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Zahl 2:");
    zahl2 = Convert.ToInt32(Console.ReadLine());
    ergebnis = zahl1 / zahl2;
    Console.WriteLine("Das Ergebnis ist {0}", ergebnis);
}
```

Code 5.1: Eine drohende Division durch null

Gibt der Anwender für die zweite Zahl eine Null ein, wird das Programm mit einer Fehlermeldung abgebrochen.

Eine Möglichkeit, solche Fehler abzufangen, wäre es, vor der Verarbeitung zu überprüfen, ob alle Werte gültig sind. In unserem Beispiel könnte das durch eine `if`-Anweisung erfolgen:

```
if (zahl2 != 0)
    ergebnis = zahl1/zahl2;
else
    //eine Meldung ausgeben
```

Durch solche Abfragen wird ein Programm allerdings sehr schnell recht unübersichtlich. Außerdem können sich auch bei den Abfragen selbst wieder Fehler einschleichen – zum Beispiel, wenn Sie `zahl2` versehentlich auf gleich null überprüfen.

Und das größte Problem: Es lassen sich nicht alle möglichen Fehler durch einfache Abfragen abfangen. Im vorigen Code werden ja zum Beispiel die beiden Zahlen durch die Anweisungen

```
zahl1 = Convert.ToInt32(Console.ReadLine());
zahl2 = Convert.ToInt32(Console.ReadLine());
```

eingeleSEN und in einen `Int32`-Typ konvertiert. Das klappt auch wunderbar, solange der Anwender tatsächlich nur eine ganze Zahl eingibt. Alle anderen Eingaben – zum Beispiel eine Zahl mit Nachkommastellen oder ein Buchstabe – führen zu einem Abbruch des Programms.



Zur Erinnerung:

Wir haben Ihnen schon ganz kurz gezeigt, wie Sie solche Konvertierungsfehler über eine eigene Ausnahmebehandlung abfangen können. Was sich genau dahinter verbirgt, werden wir uns gleich im Detail ansehen.

Ähnliches geschieht auch bei unserer Windows Forms-Anwendung für den Taschenrechner. Wenn Sie hier zum Beispiel einen Buchstaben statt einer Zahl eingeben, läuft das Programm ebenfalls auf einen Fehler hinaus.

Sie könnten jetzt versuchen, eine Methode zu programmieren, die vor der Umwandlung überprüft, ob sich möglicherweise in einem der beiden Eingabefelder ungültige Zeichen befinden. Dazu wäre allerdings erheblicher Aufwand nötig – der auch völlig überflüssig ist.

5.1 Exceptions

Denn ein ungültiger Wert in einem der beiden Eingabefelder führt nicht zu einem echten Absturz des Programms, sondern löst eine **Exception** – eine Ausnahme – aus. Das sehen Sie auch, wenn Sie im Taschenrechner einmal einen Buchstaben eingeben und die Meldung dann aufmerksam studieren.

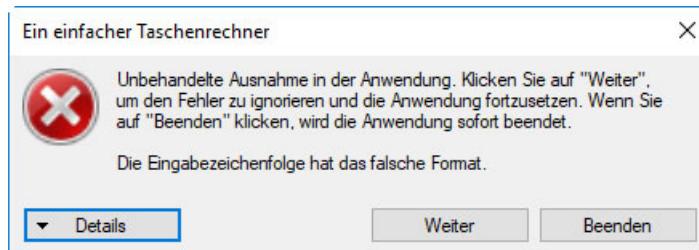


Abb. 5.1: Die Meldung bei einem ungültigen Wert

Solche Exceptions werden auch für viele andere Fehler ausgelöst, die zur Laufzeit eines Programms entstehen können – zum Beispiel

- bei unzulässigen Speicherzugriffen,
- bei Konvertierungsfehlern,
- bei Ein- und Ausgabefehlern in Dateien oder
- bei einer Division durch null.

Bei einer Division durch null wird aber nur dann eine Ausnahme ausgelöst, wenn Sie mit ganzzahligen Typen arbeiten. Bei anderen Typen liefert eine Division durch null entweder das Ergebnis `NaN` für **Not a number** („keine Zahl“), positiv unendlich oder negativ unendlich.



Fehler, die bei der Ausführung eines Programms auftreten, werden auch **Laufzeitfehler** genannt.

In der Standardeinstellung werden nahezu alle Exceptions automatisch abgefangen und auch verarbeitet. Dabei können Sie in der Regel auswählen, ob Sie das Programm beenden möchten oder trotz des Fehlers weiterarbeiten wollen.

Wenn Sie das Programm in der Entwicklungsumgebung ausführen, hängt die Reaktion auf eine Exception vom Startmodus ab. Beim Starten ohne Debuggen erscheint auch die Meldung, die in der Anwendung erscheint. Wenn Sie das Programm dagegen im Debugger starten, sieht die Meldung ein wenig anders aus.



Abb. 5.2: Die Reaktion auf eine Ausnahme im Debug-Modus

Neben der Standardreaktion können Sie auch eine eigene Reaktion auf Exceptions festlegen – Sie lassen die Ausnahmen behandeln.

Im Fachjargon wird die Ausnahmebehandlung auch **Exception Handling** genannt.



5.2 try und catch

Dazu benutzen Sie die Anweisungen `try` und `catch`.

Bei `try` geben Sie die Anweisungen an, die auf eine Exception überprüft werden sollen. Dabei können Sie auch mehrere Anweisungen verwenden.

Hinter `catch` folgen dann die Anweisungen, die ausgeführt werden sollen, wenn eine Exception ausgelöst wurde.

Den Unterschied zwischen `try` und `catch` können Sie sich auch leicht an der Übersetzung merken: `try` bedeutet so viel wie „versuche“ und `catch` so viel wie „fange“.

Bei `catch` werden die **Exception Handler** festgelegt. Übersetzt bedeutet Exception Handler so viel wie „Ausnahmebehandler“.



Auf welche Exception `catch` reagieren soll, wird in runden Klammern hinter `catch` angegeben. Falls sämtliche Ausnahmen bearbeitet werden sollen, geben Sie nur das Schlüsselwort ohne Klammern an.

Allgemein sieht die `try-catch`-Konstruktion so aus:

```
try
{
    Anweisung 1;
    Anweisung 2;
```

```

    ...
}
catch (Exception)
{
    Anweisung 1;
    Anweisung 2;
    ...
}

```



Bitte beachten Sie:

Die geschweiften Klammern sind auch dann erforderlich, wenn Sie nur eine einzige Anweisung in einem der Blöcke verwenden.

Wenn Sie so wollen, entspricht die `try-catch`-Konstruktion einer `if`-Konstruktion. Wenn im `try`-Block eine Ausnahme ausgelöst wurde, werden die Anweisungen im `catch`-Block ausgeführt. Andernfalls werden die Anweisungen im `catch`-Block übersprungen.

Schauen wir uns `try` und `catch` nun an einigen praktischen Beispielen an.



Bitte beachten Sie:

Wenn Sie ein Konsolenprogramm im Debug-Modus ausführen lassen, wird das Fenster direkt nach dem Ende des Programms wieder geschlossen. Starten Sie das folgende Programm daher bitte über die Funktion **Starten ohne Debugging** oder mit der Tastenkombination **Strg + F5**. Andernfalls sind die Meldungen der Ausnahmebehandlung kaum zu sehen.

Beginnen wir mit dem Abfangen der Division durch null im Code 5.2. Mit einer `try-catch`-Konstruktion sieht dieser Code so aus. Die neu eingefügten Teile haben wir fett markiert:

```

static void Main(string[] args)
{
    int zahl1, zahl2, ergebnis;
    Console.WriteLine("Zahl 1:");
    zahl1 = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Zahl 2:");
    zahl2 = Convert.ToInt32(Console.ReadLine());
    //die Division wird versucht
    try
    {
        ergebnis = zahl1 / zahl2;
    }
    //wenn es nicht geklappt hat
    catch
    {
        Console.WriteLine("Es ist ein Fehler aufgetreten.");
        Console.WriteLine("Ergebnis erhält den Wert 0.");
    }
}

```

```
    ergebnis = 0;
}
Console.WriteLine("Das Ergebnis ist {0}", ergebnis);
}
```

Code 5.2: try-catch-Konstruktion

Im `try`-Block werden die beiden Werte dividiert. Wenn dabei eine Ausnahme auftritt, werden im `catch`-Block zwei Meldungen ausgegeben und die Variable `ergebnis` auf null gesetzt.

Danach wird dann der Wert der Variablen `ergebnis` ausgegeben. Die entsprechende Ausgabeanweisung befindet sich außerhalb des `catch`-Blocks. Das heißt, sie wird in jedem Fall ausgeführt.

Testen Sie den vorigen Code jetzt einmal. Lassen Sie das Programm übersetzen und ausführen. Geben Sie dann für die zweite Zahl einmal den Wert 0 ein. Sie werden sehen, die Ausnahme wird durch den `catch`-Block abgefangen.

Hinweis:

Wenn das Konsolenfenster nach der Eingabe der 0 sehr schnell wieder geschlossen wird, haben Sie das Programm wahrscheinlich im Debug-Modus ausgeführt. Starten Sie es dann bitte noch einmal mit der Funktion **Starten ohne Debuggen** und testen Sie es erneut.

Bitte beachten Sie, dass die Anweisungen im `try`-Block nach dem Auftreten einer Ausnahme nicht weiter ausgeführt werden. Das können Sie ganz einfach selbst ausprobieren, wenn Sie im vorigen Code nach der Anweisung zur Division noch eine weitere Anweisung einfügen – zum Beispiel eine Ausgabe. Diese Anweisung wird nur dann ausgeführt, wenn **keine** Ausnahme bei der Division auftritt. Andernfalls wird die Anweisung schlicht und einfach übersprungen.

Schauen wir uns jetzt noch an, wie Sie gezielt auf unterschiedliche Ausnahmen reagieren. Dazu benutzen wir eine einfache Windows Forms-Anwendung als Beispiel.

Erstellen Sie bitte ein neues Projekt für eine Windows Forms-Anwendung. Fügen Sie in das Formular ein Steuerelement **TextBox** und eine Schaltfläche ein. Legen Sie dann für das Ereignis **Click** der Schaltfläche folgende Anweisungen fest:

```
int ergebnis, zahl2, zahl1 = 10;
//Konvertierung und Berechnung werden versucht
try
{
    zahl2 = Convert.ToInt32(textBox1.Text);
    ergebnis = zahl1/zahl2;
    MessageBox.Show("Das Ergebnis ist " + ergebnis, "Hurra");
}
catch
{
    MessageBox.Show("Da ist etwas schief gegangen", "Oh nein");
}
```

Code 5.3: Ausnahmebehandlung für eine Division

Provokieren Sie dann eine Division durch Null. Geben Sie dazu die Zahl 0 in das Feld ein und klicken Sie anschließend auf die Schaltfläche. Die Exception wird sauber durch den `catch`-Block abgefangen.

Geben Sie dann einmal einen Buchstaben in das Feld ein und klicken Sie erneut auf die Schaltfläche. Auch diese Exception wird durch das Programm abgefangen.

Im vorigen Code können wir allerdings die beiden unterschiedlichen Exceptions nicht auseinanderhalten, da wir ja bei `catch` sämtliche Exceptions in einem gemeinsamen Block verarbeiten. Sie können aber in Klammern auch den Namen der Exception-Klasse angeben, auf die `catch` reagieren soll. Bei einem Konvertierungsfehler durch ein ungültiges Format wird zum Beispiel eine Ausnahme der Exception-Klasse `FormatException` ausgelöst. Der entsprechende `catch`-Aufruf würde dann so aussehen:

```
//bei einem ungültigen Format
catch (FormatException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Bei der Konvertierung ist
    etwas schief gegangen.", "Oh nein");
}
```

Code 5.4: Gezielte Reaktion auf eine Exception-Klasse

In dem `catch`-Block aus dem vorigen Code wird allerdings nur noch auf Exceptions der Klasse `FormatException` reagiert. Andere Ausnahmen werden nicht mehr behandelt. Sie müssten dann für alle weiteren Exceptions einen eigenen `catch`-Block erstellen und – um ganz auf „Nummer Sicher“ zu gehen – noch einen weiteren `catch`-Block, der sämtliche Exceptions abdeckt.

Für unser Beispiel könnten diese Erweiterungen so aussehen:

```
//bei einem ungültigen Format
catch (FormatException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Bei der Konvertierung ist
    etwas schief gegangen.", "Oh nein");
}
//bei einer Division durch null
catch (DivideByZeroException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Eine Division durch null ist nicht
    definiert.", "Oh nein");
}
//bei allen anderen Ausnahmen
catch
{
    MessageBox.Show("Ein anderes Problem", "Oh nein");
}
```

Code 5.5: Reaktion auf verschiedene Ausnahmen

Hinweis:

Das komplette Beispiel finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **Exception01**.

Im zweiten `catch`-Block reagieren wir auf eine Ausnahme der Klasse `DivideByZeroException`. Sie wird bei einer Division durch null ausgelöst.

Zur Erinnerung:

Die Ausnahme `DivideByZeroException` wird nur dann bei einer Division durch null ausgelöst, wenn Sie ganzzahlige Typen verwenden.



Im dritten `catch`-Block schließlich fangen wir alle anderen Ausnahmen ab.

Bitte beachten Sie:

Wenn eine Ausnahme in einem `try`-Block auftritt, wird nur der nächste passende `catch`-Block ausgeführt. Alle anderen `catch`-Blöcke werden ignoriert. Der `catch`-Block, der sämtliche Ausnahmen behandelt, muss daher immer am Ende stehen. Falls das nicht der Fall ist, erscheint auch eine Fehlermeldung beim Übersetzen.



Probieren Sie den geänderten Code jetzt einmal aus. Sie werden sehen, je nach Ausnahme erscheint auch eine passende Meldung.

Hinweis:

Eine Tabelle mit einigen Standard-Exception-Klassen finden Sie im Anhang D dieses Studienhefts.

Schauen wir uns nun noch an, wie Sie geschachtelte `try`-Blöcke erstellen – also einen `try`-Block in einen anderen `try`-Block setzen. Ein Beispiel finden Sie im folgenden Code. Hier werden die Anweisungen zum Konvertieren und zur Division in zwei verschachtelten `try`-Blöcken ausgeführt.

```
private void Button1_Click(object sender, EventArgs e)
{
    int ergebnis, zahl2, zahl1 = 10;
    //Konvertierung im äußeren try-Block
    try
    {
        zahl2 = Convert.ToInt32(textBox1.Text);
        //die Berechnungen im inneren try-Block
        try
        {
            ergebnis = zahl1 / zahl2;
            MessageBox.Show("Das Ergebnis ist " + ergebnis, "Hurra");
        }
        //catch für den inneren Block
        catch (DivideByZeroException)
        {
```

```

        //bitte in einer Zeile eingeben
        MessageBox.Show("Eine Division durch
        null ist nicht definiert.", "Oh nein");
    }
}
//catch für den äußeren Block
catch (FormatException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Bei der Konvertierung ist
    etwas schief gegangen.", "Oh nein");
}
//bei allen anderen Ausnahmen
catch
{
    MessageBox.Show("Ein anderes Problem", "Oh nein");
}
}

```

Code 5.6: Geschachtelte try-Blöcke

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Exception02**.

Bei geschachtelten `try`-Blöcken müssen Sie zwei Besonderheiten beachten:

- 1) Die `catch`-Anweisungen für einen `try`-Block müssen unmittelbar auf den jeweiligen Block folgen. Dabei kann es für einen `try`-Block auch mehrere `catch`-Anweisungen geben. Im vorigen Code gehören die beiden letzten `catch`-Anweisungen zum Beispiel zum äußeren `try`-Block.
- 2) Ein `try`-Block, der innen liegt, kann auch `catch`-Anweisungen der äußeren Blöcke ausführen – und zwar dann, wenn seine eigenen `catch`-Anweisungen keine passende Ausnahmebehandlung bieten. Ein `try`-Block, der außen liegt, kann allerdings keine `catch`-Anweisung eines inneren Blocks aufrufen.

Probieren Sie das selbst aus. Vertauschen Sie im vorigen Code einmal die beiden Anweisungen `catch (FormatException)` und `catch (DivideByZeroException)`. Wenn Sie dann zum Beispiel einen Buchstaben in das Feld im Formular eingeben, wird die Anweisung für `catch` ohne weitere Angaben ausgeführt. Das liegt daran, dass sich die Ausnahmebehandlung für `FormatException` jetzt im inneren `try`-Block befindet – also vom äußeren `try`-Block nicht verarbeitet wird.

Bei einer Division durch null dagegen werden die Anweisungen für `catch (DivideByZeroException)` ausgeführt, obwohl sich diese `catch`-Anweisung nicht im inneren `try`-Block befindet.

5.3 Eigene Exceptions auslösen

Neben den Standard-Exceptions von C# können Sie auch eigene Exceptions auslösen. Dazu erzeugen Sie eine neue Instanz der Basisklasse `Exception` und lösen die Exception dann mit dem Schlüsselwort `throw`³ aus. Das hört sich sehr viel komplizierter an, als es tatsächlich ist. Ein Beispiel:

```
throw new Exception("Da ist etwas schiefgegangen");
```

lässt eine Exception über eine neue Instanz der Basisklasse `Exception` aus und liefert eine Zeichenkette für eine Meldung.

Die Auswertung der Exception erfolgt wie gewohnt mit einer `catch`-Anweisung. In den Klammern geben Sie dabei den Typ `Exception` an. Über die Eigenschaft `Message` können Sie auch den Text ausgeben, den die Exception geliefert hat. Die `catch`-Anweisung könnte zum Beispiel so aussehen:

```
catch (Exception fehler)
{
    MessageBox.Show(fehler.Message, "Oh nein");
}
```

Bitte beachten Sie:

Diese `catch`-Anweisung verarbeitet alle Exceptions, die von der Basisklasse `Exception` abgeleitet werden – also auch die Standard-Exceptions.



Schauen wir uns den Einsatz von selbst ausgelösten Exceptions jetzt an einem vollständigen Beispiel an. Im folgenden Code wird beim Anklicken einer Schaltfläche zuerst der Wert in einem Eingabefeld in eine Zahl umgewandelt. Danach wird eine Methode `Check()` für die Zahl aufgerufen. Diese Methode prüft, ob die Zahl kleiner als 5 oder größer als 10 ist. Für diese beiden Fälle wird über `throw` eine Exception ausgelöst. Die Auswertung der Exceptions erfolgt dann im inneren `try`-Block in der Methode `button1_Click()`.

```
//eine Methode zum Erzeugen der Exceptions
void Check(int zahl)
{
    //wenn zahl größer ist als 10, wird eine Exception ausgelöst
    if (zahl > 10)
        throw new Exception("Die Zahl ist zu groß.");
    //wenn zahl kleiner ist als 5 ebenfalls
    if (zahl < 5)
        throw new Exception("Die Zahl ist zu klein.");
}

private void Button1_Click(object sender, EventArgs e)
{
    int zahl1;
```

3. `throw` bedeutet übersetzt so viel wie „werfe“.

```

//die Konvertierung im äußeren try-Block
try
{
    zahl1 = Convert.ToInt32(textBox1.Text);
    //die Methode Check() im inneren try-Block aufrufen
    try
    {
        Check(zahl1);
        MessageBox.Show("Ihre Eingabe war " + zahl1, "Meldung");
    }
    //die Exceptions aus der Methode Check() verarbeiten
    catch (Exception fehler)
    {
        MessageBox.Show(fehler.Message, "Oh nein");
    }
}
//catch für den äußeren Block
catch (FormatException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Bei der Konvertierung ist
    etwas schief gegangen.", "Oh nein");
}
}

```

Code 5.7: Selbst ausgelöste Exceptions

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Exception03**.

Probieren Sie den vorigen Code einmal in der Anwendung mit einem Eingabefeld und einer Schaltfläche aus. Sie werden sehen, bei Werten kleiner als 5 und größer als 10 löst die Methode `Check()` die entsprechenden Exceptions aus.

Experimentieren Sie auch ein wenig mit dem Code. Lösen Sie zum Beispiel den inneren `try`-Block einmal auf und setzen Sie die Anweisungen in den äußeren `try`-Block. Ersetzen Sie dann die `catch`-Anweisung vom äußeren `try`-Block durch die `catch`-Anweisung vom inneren `try`-Block. Die geänderte Methode `Button1_Click()` könnte danach so aussehen:

```

private void Button1_Click(object sender, EventArgs e)
{
    int zahl1;
    //jetzt erfolgt alles in einem try-Block
    try
    {
        zahl1 = Convert.ToInt32(textBox1.Text);
        Check(zahl1);
        MessageBox.Show("Ihre Eingabe war " + zahl1, "Meldung");
    }
    //die Exceptions aus der Methode Check() verarbeiten
    catch (Exception fehler)
    {

```

```
        MessageBox.Show(fehler.Message, "Oh nein");  
    }  
}
```

Code 5.8: Der aufgelöste innere try-Block

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Exception04**.

Sie werden sehen, die `catch`-Anweisung wird auch bei einem Konvertierungs-Fehler ausgelöst. Hier wird dann eine Meldung angezeigt, dass die Eingabezeichenfolge das falsche Format hat.

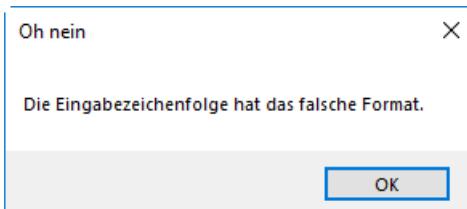


Abb. 5.3: Die Meldung für den Konvertierungsfehler

5.4 Die Anweisung finally

Kommen wir nun zur Anweisung `finally`⁴. Mit dieser Anweisung können Sie für `try` einen Block definieren, der unabhängig davon ausgeführt wird, ob eine Exception auftritt oder nicht.

Allgemein dargestellt sieht die `try-finally`-Konstruktion so aus:

```
try  
{  
    Anweisung 1;  
    Anweisung 2;  
    ...  
}  
catch (Exception)  
{  
    Anweisung 1;  
    Anweisung 2;  
    ...  
}  
finally  
{  
    Anweisung 1;  
    Anweisung 2;  
    ...  
}
```

4. `finally` bedeutet übersetzt so viel wie „endgültig“ oder „schließlich“.

Ein Beispiel für den Einsatz finden Sie im folgenden Code:

```
private void Button1_Click(object sender, EventArgs e)
{
    //eine Ausnahme provozieren
    try
    {
        int[] test = new int[1000000000];
        test[0] = 10;
    }
    //die Ausnahme behandeln
    catch (Exception fehler)
    {
        MessageBox.Show(fehler.Message, "Oh nein");
    }
    finally
    {
        MessageBox.Show("Diese Meldung erscheint immer.", "finally");
    }
}
```

Code 5.9: Die finally-Anweisung

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Exception05**.

Im `try`-Block wird versucht, Speicher für ein riesiges Array zu reservieren, und so eine Ausnahme `OutOfMemoryException` ausgelöst. Die Anweisung bei `finally` wird dabei in jedem Fall ausgeführt – gleichgültig, ob die Reservierung gelingt oder nicht. Das können Sie ganz einfach ausprobieren, indem Sie den Wert für die Elemente des Arrays zum Beispiel auf 10 verändern. Ein solch kleines Array sollte sich in jedem Fall erstellen lassen.

In dem Beispiel von oben ergibt die `finally`-Anweisung so allerdings nur wenig Sinn, da Sie den Text ja auch einfach direkt nach dem `catch`-Block ausgeben könnten. Es gibt aber durchaus Situationen, in denen `finally`-Blöcke wichtig sein können – zum Beispiel bei der Arbeit mit Dateien oder Datenbanken. Hier kann im `finally`-Block dafür gesorgt werden, dass die Datei beziehungsweise Datenbank korrekt geschlossen wird.

5.5 Exceptions filtern

Schauen wir uns jetzt noch an, wie Sie Exceptions filtern. Das ist zum Beispiel dann interessant, wenn Sie bei Exceptions vom selben Typ auf unterschiedliche Bedingungen reagieren möchten.

Um eine Exception zu filtern, geben Sie hinter dem `catch`-Ausdruck das Schlüsselwort `when` und die Bedingung an. Die Bedingung setzen Sie dabei in runde Klammern. Allgemein sieht eine gefilterte Exception also so aus:

```
catch (Ausnahme) when (Bedingung)
```

Nehmen wir ein praktisches Beispiel. Im folgenden Code wird eine Exception ausgelöst und über die Nachricht gefiltert.

```
private void Button1_Click(object sender, EventArgs e)
{
    //eine Ausnahme auslösen
    try
    {
        throw new Exception("Test");
    }
    //die Ausnahme behandeln und filtern
    catch (Exception fehler) when (fehler.Message == "Test")
    {
        MessageBox.Show("Gefiltert", "Oh nein");
    }
    //die Ausnahme ohne Filter behandeln
    catch (Exception fehler)
    {
        MessageBox.Show("Ungefiltert" + fehler.Message, "Oh nein");
    }
}
```

Code 5.10: Eine gefilterte Ausnahme

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Exception06**.

Die Anweisung

```
catch (Exception fehler) when (fehler.Message == "Test")
```

sorgt dafür, dass die folgende Meldung nur dann erscheint, wenn die Nachricht der Ausnahme den Wert „Test“ hat.

In der Anweisung

```
catch (Exception fehler)
```

dagegen haben wir keinen Filter definiert. Dieser Block wird also in allen anderen Fällen ausgeführt.

Wenn ein `catch`-Block mit einem Filter ausgeführt wurde, werden keine weiteren Blöcke für diese Ausnahme mehr ausgeführt.



5.6 Wann lohnt eine eigene Ausnahmebehandlung?

Wie Sie in den letzten Abschnitten gesehen haben, ist die Ausnahmebehandlung selbst bei recht einfachen Programmen mit einem Aufwand verbunden. Außerdem müssen Sie vor allem bei geschachtelten `try`-Blöcken sehr sorgfältig arbeiten, damit Sie auch wirklich die Ausnahmen mit dem richtigen Exception Handler an der passenden Stelle verarbeiten.

Bevor Sie jetzt aber beschließen, auf die Ausnahmebehandlung lieber zu verzichten, eine gute Nachricht: Nahezu alle Programme, die Sie mit Visual Studio erstellen, arbeiten automatisch mit einer Ausnahmebehandlung, die fast alle Ausnahmen abfängt und verarbeitet. Dabei wird das Programm nach Möglichkeit nicht beendet, sondern wieder in den Zustand versetzt, den es vor der Ausnahme hatte. Wenn Sie so wollen, wird das Programm „zurückgespult“.

Probieren Sie das einfach einmal aus. Kommentieren Sie zum Beispiel einmal im Code 5.3 sämtliche Anweisungen zur Ausnahmebehandlung aus. Die Anweisungen `MessageBox.Show()` können Sie dabei zu Testzwecken weiter ausführen lassen. Starten Sie dann das Programm und provozieren Sie sowohl einen Konvertierungsfehler als auch eine Division durch null. Sie werden sehen: Wenn Sie das Programm nicht im Debug-Modus laufen lassen, erscheint eine Standardfehlermeldung und das Programm wird nach dem Auftreten der Exception wieder „zurückgespult“, sobald Sie auf die Schaltfläche **Weiter** klicken. So erscheinen die Meldungen von `MessageBox.Show()` zum Beispiel nur dann, wenn keine Exception aufgetreten ist.

Allein aus Gründen der Betriebssicherheit müssen Sie also selbst keine Ausnahmebehandlung in Ihren Programmen implementieren. Sie können aber über eigene Exception Handler die Bedienung Ihrer Programme vereinfachen und auch detailliertere Fehlermeldungen anzeigen lassen. So können Sie zum Beispiel den Fokus in einem Exception Handler wieder auf das Element zurücksetzen, das den Fehler verursacht hat. Damit werden Sie sich in einem Teil der Einsendeaufgaben auseinandersetzen.

Auch für die Übersetzung von Fehlermeldungen wie „Authentication failed“ eignen sich eigene Exception Handler ideal. Hier könnten Sie dem Anwender im Klartext mitteilen, dass eine Anmeldung fehlgeschlagen ist und was er tun muss.

Für eine eigene Ausnahmebehandlung in Ihren Programmen müssen Sie aber lediglich die Exception Handler für die Ausnahmen erstellen, auf die Ihr Programm individuell reagieren soll. Sie überschreiben dann die Standard Exception Handler. Die gesamte restliche Ausnahmebehandlung können Sie Visual Studio überlassen.



Bitte beachten Sie unbedingt:

Wenn Sie einen Standard Exception Handler überschreiben, wird die Methode, in der die Ausnahme aufgetreten ist, weiter bearbeitet und **nicht** mehr automatisch beendet. Das heißt also, alle Anweisungen, die nach dem Exception Handler stehen, werden auch bei einer Ausnahme weiter ausgeführt. Sie sollten daher die Methode im Exception Handler gegebenenfalls über `return` verlassen.

Abschließend noch zwei Hinweise zur Ausnahmebehandlung:

Einige Steuerelemente verfügen über eigene Ereignisse, die bei einer Ausnahme eintreten. Hier können Sie dann einen „Exception Handler“ wie gewohnt über die entsprechende Methode programmieren.

Behalten Sie bitte auch im Hinterkopf, dass sich Fehler wie eine Division durch null durch sorgfältige Programmierung und ausgiebige Tests in vielen Fällen vermeiden lassen. Die Ausnahmebehandlung von Visual Studio ist also kein Freibrief für eilige Programmierer.

So viel zur Ausnahmebehandlung.

Zusammenfassung

Bei Laufzeitfehlern wird das Programm unterbrochen. Sie können dann in der Regel auswählen, ob Sie das Programm fortsetzen oder beenden möchten.

Sie können selbst festlegen, wie das Programm auf Ausnahmen reagieren soll. Dazu geben Sie zuerst im `try`-Block die Anweisungen ein, die auf eine Ausnahme überprüft werden sollen. Anschließend folgen im `catch`-Block die Anweisungen, die beim Auftreten einer Ausnahme ausgeführt werden sollen.

Im `catch`-Block können Sie auch gezielt auf bestimmte Ausnahmen reagieren.

Mit dem Schlüsselwort `throw` können Sie eigene Ausnahmen auslösen.

Mit einem `finally`-Block können Sie Anweisungen für einen `try`-Block definieren, die immer ausgeführt werden – also auch dann, wenn keine Ausnahme auftritt.

Alle Programme, die Sie mit Visual Studio erstellen, verfügen über eine eigene Ausnahmebehandlung.

Aufgaben zur Selbstüberprüfung

- 5.1 Was ist eine Exception? Wann wird sie ausgelöst? Nennen Sie mindestens zwei Beispiele.

- 5.2 Wann müssen die Anweisungen im `try`- und `catch`-Block in geschweifte Klammern gesetzt werden?

- 5.3 Was sind die Exception Handler? Wo werden die Exception Handler angegeben?

- 5.4 Wie reagieren Sie in einem `catch`-Block gezielt auf eine bestimmte Ausnahme?

- 5.5 Welche Besonderheiten müssen Sie bei der Schachtelung von `try`-Blöcken beachten?

Schlussbetrachtung

In diesem Studienheft haben wir uns mit der Fehlersuche und der Ausnahmebehandlung beschäftigt. Sie wissen jetzt, wie Sie den integrierten Debugger von Visual Studio einsetzen. Außerdem haben Sie mit der Ausnahmebehandlung eine Möglichkeit kennengelernt, selbst auf Laufzeitfehler in Ihren Programmen zu reagieren.

Betrachten Sie die Ausnahmebehandlung aber bitte nicht als Freibrief. Kein Anwender wird begeistert sein, wenn mehrfach schwere Fehlermeldungen in einem Programm erscheinen, die dazu möglicherweise noch recht unverständlich sind. Setzen Sie die Ausnahmebehandlung gezielt ein, um Fehler abzufangen und dem Anwender detailliert mitzuteilen, was er falsch gemacht hat und wie er den Fehler beheben kann.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Ein Off-by-one-Fehler tritt auf, weil ein Wert entweder um eins zu groß oder zu klein ist.
- 1.2 Eigentlich handelt es sich um einen Syntaxfehler. Der Vergleichsoperator besteht aus zwei Gleichheitszeichen. Da ein einfaches Gleichheitszeichen eine Zuweisung ausführt, entsteht durch den Syntaxfehler aber auch ein Semantikfehler.

Kapitel 2

- 2.1 Der Unterschied macht sich vor allem beim Aufruf von Methoden bemerkbar. Die Anweisung **Prozedurschritt** führt die gesamte Methode auf einmal aus. Die Anweisung **Einzelschritt** dagegen wechselt in die Methode und führt dort alle Anweisungen einzeln aus.
- 2.2 Sie stellen den Mauszeiger auf ein Vorkommen der Variablen im Quelltext und warten einen Moment ab. In einem speziellen Bildschirmtipp – dem DataTip – wird dann der aktuelle Wert angezeigt.
- 2.3 Sie drücken die Funktionstaste **F11**.

Kapitel 3

- 3.1 Ein Haltepunkt wird immer dann ausgelöst, wenn das Programm bei der Ausführung an die entsprechende Stelle im Quelltext gelangt. Bei der Funktion **Ausführen bis Cursor** wird das Programm nur einmal an der entsprechenden Stelle angehalten.
- 3.2 Um den Wert eines Datenobjekts über den Debugger zu ändern, gibt es zwei Möglichkeiten:
 - Sie lassen den DataTip anzeigen. Nach dem Anklicken des DataTips können Sie den Wert überschreiben.
 - Sie doppelklicken im Register **Lokal** auf die Spalte **Wert** des Datenobjekts und geben dann den neuen Wert ein.

Für die richtige Lösung reicht es aus, wenn Sie eine der beiden Möglichkeiten angegeben haben.

Kapitel 4

- 4.1 Ein Test soll Fehler finden. Der Nachweis der Fehlerfreiheit ist bei komplexeren Programmen unmöglich.
- 4.2 Ein Testfall beschreibt bei einem funktionsorientierten Testverfahren die verwendeten Eingabedaten und die zu erwartenden Ausgabedaten.
- 4.3 Äquivalenzklassen fassen sehr ähnliche Testfälle, die zu demselben Ergebnis führen, zusammen.

Kapitel 5

- 5.1 Eine Exception ist eine Ausnahme. Sie wird bei bestimmten Laufzeitfehlern eines Programms ausgelöst – zum Beispiel bei einer Division durch null mit ganzzahligen Typen, bei unzulässigen Speicherzugriffen, bei Konvertierungsfehlern oder bei Ein- und Ausgabefehlern in Dateien.
- 5.2 Die Anweisungen im `try`- und `catch`-Block müssen immer in geschweifte Klammern gesetzt werden.
- 5.3 Die Exception Handler definieren die verschiedenen Ausnahmen, auf die das Programm reagieren soll. Sie werden in den `catch`-Blöcken definiert.
- 5.4 Sie geben die Klasse der Ausnahme in den runden Klammern hinter `catch` an.
- 5.5 Bei geschachtelten `try`-Blöcken müssen Sie zwei Besonderheiten beachten:
 - Die `catch`-Anweisungen für einen `try`-Block müssen unmittelbar auf den `try`-Block folgen.
 - Ein `try`-Block, der innen liegt, kann auch `catch`-Anweisungen der äußeren Blöcke aufrufen. Ein `try`-Block, der außen liegt, kann aber keine `catch`-Anweisungen eines inneren Blocks aufrufen.

B. Glossar

Äquivalenz	Äquivalenz bedeutet so viel wie „Gleichwertigkeit“.
Äquivalenzklassen	Äquivalenzklassen werden beim Test von Software verwendet. Sie fassen sehr ähnliche Testfälle, die zu demselben Ergebnis führen, zusammen.
Ausnahme	Ausnahme ist die deutsche Übersetzung für <i>Exception</i> .
Ausnahmebehandlung	Durch die Ausnahmebehandlung können Sie in einem Programm auf Laufzeitfehler reagieren. Die Ausnahmebehandlung wird auch <i>Exception Handling</i> genannt.
Blackbox-Test	<i>Blackbox-Test</i> ist der englische Ausdruck für funktionsorientierte Testverfahren.
Breakpoint	<i>Breakpoint</i> ist der englische Ausdruck für Haltepunkt.
Bug	Ein <i>Bug</i> (engl. „Insekt, Wanze“) ist ein Fehler in einem Computerprogramm. Dieser Name leitet sich von einem Insekt ab, das in den Anfangszeiten der Informations-technologie einen Kurzschluss in einem Rechner verursachte.
Bug fixing	<i>Bug fixing</i> ist ein anderer Ausdruck für die Korrektur eines Fehlers.
Debuggen	<i>Debuggen</i> ist der Fachbegriff für das Beseitigen von Fehlern in einem Computerprogramm.
Debugger	Ein <i>Debugger</i> ist ein Werkzeug zur Fehlersuche. Wörtlich übersetzt bedeutet <i>Debugger</i> so viel wie „Entwandler“.
Exception	Eine <i>Exception</i> – eine Ausnahme – wird durch nicht definierte Zustände bei der Ausführung eines Programms ausgelöst. Dazu gehören zum Beispiel Divisionen durch null bei ganzen Zahlen oder fehlgeschlagene Speicher-reservierungen.
Exception Handler	Der <i>Exception Handler</i> besteht aus den Anweisungen, die beim Auftreten einer Ausnahme ausgeführt werden sollen.
Exception Handling	<i>Exception Handling</i> ist die englische Bezeichnung für Ausnahmebehandlung.
Exception-Klasse	Die <i>Exception</i> -Klassen fassen bestimmte Ausnahmen zusammen – zum Beispiel Fehler beim Konvertieren von Daten.

Funktionsorientierte Testverfahren	Funktionsorientierte Testverfahren sind eine Methode für die Qualitätsprüfung von Software. Ein Testobjekt wird mit Daten „gefüttert“. Die erzeugten Ausgabedaten werden dann mit den Daten verglichen, die das Testobjekt bei korrekter Funktion liefern müsste. Funktionsorientierte Testverfahren werden auch <i>Black-box</i> -Tests genannt.
Haltepunkt	Ein Haltepunkt wird beim Debuggen von Programmen eingesetzt. Das Programm wird nur bis zum Haltepunkt ausgeführt und dann unterbrochen. Haltepunkte werden auch <i>Breakpoints</i> genannt.
Heißes Testen	Das „heiße“ Testen ist eine Methode für die Qualitätsprüfung von Software. Es werden so lange willkürlich Daten eingegeben beziehungsweise das Programm wird so lange ausgeführt, bis ein Fehler auftritt – oder eben nicht. „Heiße“ Tests eignen sich lediglich für erste grobe Funktionsprüfungen. Für eine echte Qualitätsprüfung sind sie nicht zu gebrauchen.
Off-by-one-Fehler	<i>Off-by-one</i> -Fehler entstehen, weil ein Wert um 1 zu groß oder um 1 zu klein ist. Dadurch wird zum Beispiel ein Schleifendurchlauf zu viel oder zu wenig durchgeführt. <i>Off by one</i> bedeutet übersetzt so viel wie „Um eins daneben“ oder „Einen entfernt“.
Schreibtischtest	Der Schreibtischtest ist ein statisches Testverfahren für Software. Ein Quelltext wird ausgedruckt und dann Zeile für Zeile auf dem Trockenen durchgespielt.
Semantik	Die Semantik beschreibt bei einem Quelltext die Funktionalität der Anwendung.
Statische Testverfahren	Statische Testverfahren werden beim Test von Software eingesetzt. Hier werden vor allem Dokumente wie zum Beispiel Quelltexte auf dem Trockenen geprüft. Die meisten statischen Testverfahren werden in Teamsitzungen durchgeführt, um einen möglichst objektiven Blick auf das Testobjekt sicherzustellen.
Syntax	Die Syntax bezeichnet die Regeln, die durch die Programmiersprache vorgegeben werden.
Syntaxfehler	Syntaxfehler sind Verstöße gegen die Regeln der Programmiersprache – zum Beispiel ein fehlendes Semikolon oder eine falsche Klammer. Syntaxfehler werden vom Compiler bei der Übersetzung des Programms gemeldet.

Test	Bei einem Test wird ein Produkt analysiert, mit eindeutigen Vorgaben ausprobiert und sein Verhalten beobachtet.
Testfall	Ein Testfall beschreibt beim Test von Software die verwendeten Eingabedaten und die erwarteten Ausgaben. Testfälle werden zum Beispiel bei funktionsorientierten Testverfahren verwendet.
Typecasting	<i>Typecasting</i> bezeichnet das Umwandeln eines Wertes in einen anderen Datentyp.

C. Tastenkombinationen zum Arbeiten mit dem Debugger

Tab. C.1: Tastenkombinationen zum Arbeiten mit dem Debugger

Taste/ Tastenkombination	Wirkung
[F5]	Das Programm wird im Debug-Modus gestartet beziehungsweise die Ausführung wird fortgesetzt. Wenn ein Haltepunkt definiert ist, wird die Ausführung an dem Haltepunkt unterbrochen.
[Strg] + [F5]	Das Programm wird ohne Debugger gestartet. Dabei werden gesetzte Haltepunkte ignoriert.
[↑] + [F5]	Das Debuggen wird abgebrochen.
[Strg] + [↑] + [F5]	Das Programm wird neu gestartet.
[F9]	Die aktuelle Zeile im Quelltext wird als Haltepunkt markiert beziehungsweise der Haltepunkt in der aktuellen Zeile wird wieder ausgeschaltet.
[Strg] + [F10]	Das Programm wird bis zu der Zeile ausgeführt, in der sich die Einfügemarke befindet.
[F11]	Die nächste Anweisung wird als Einzelschritt ausgeführt.
[F10]	Die nächste Anweisung wird als Prozedurschritt ausgeführt. Wenn es sich um eine Methode handelt, wird die gesamte Methode komplett ausgeführt.
[↑] + [F11]	Das Programm wird bis zu der Stelle ausgeführt, an der der Rückgabewert der aktuellen Methode zugewiesen wird.

D. Standard-Exceptions

Tab. D.1: Standard-Exceptions

Exception-Klasse	Wirkung
System.AccessViolationException	Die Ausnahme AccessViolationException wird bei einem ungültigen Speicherzugriff ausgelöst.
System.ArgumentNullException	Die Ausnahme ArgumentNullException wird bei Methoden ausgelöst, wenn ein Argument mit einem Nullwert übergeben wird, der von der Methode nicht verarbeitet werden kann.
System.DivideByZeroException	Die Ausnahme DivideByZeroException wird bei einer Division durch null ausgelöst, wenn Sie ganzzahlige Typen verwenden. Bitte beachten Sie: Bei Typen wie float oder double führt eine Division durch null nicht zu einer Ausnahme.
System.IO.DriveNotFoundException	Die Ausnahme DriveNotFoundException wird ausgelöst, wenn auf ein ungültiges Laufwerk zugegriffen wird.
System.Exception	Die Klasse Exception ist die Basisklasse für alle Ausnahmen.
System.IO.FileNotFoundException	Die Ausnahme FileNotFoundException wird ausgelöst, wenn auf eine nicht vorhandene Datei zugegriffen wird.
System.FormatException	Die Ausnahme FormatException wird ausgelöst, wenn das Argument beim Aufruf einer Methode nicht mit dem festgelegten Parameter der Methode übereinstimmt.
System.IndexOutOfRangeException	Die Ausnahme IndexOutOfRangeException wird bei einem Zugriff auf ein ungültiges Element eines Arrays ausgelöst.
System.InvalidCastException	Die Ausnahme InvalidCastException wird bei einem ungültigen Typecasting ausgelöst. Zur Auffrischung: Beim Typecasting wird ein Wert in einen anderen Datentyp umgewandelt.
System.InvalidOperationException	Die Ausnahme InvalidOperationException wird bei Methoden ausgelöst, wenn eine ungültige Operation ausgeführt wurde.
System.NullReferenceException	Die Ausnahme NullReferenceException wird beim Zugriff auf ein Objekt mit einem Nullwert ausgelöst.
System.SystemException	Die Klasse SystemException ist die Basisklasse für alle Ausnahmen, die von der Laufzeitumgebung ausgelöst werden.

E. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in Visual C# 2019. Ideal für Programmieranfänger geeignet.* 6. Aufl., Bonn: Rheinwerk.

Speziell mit dem Thema „Testen von Software“ beschäftigt sich das folgende Buch:

Vigenschow, U. (2010). *Testen von Software und Embedded Systems. Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen.* 2. überarbeitete und aktualisierte Aufl., Heidelberg: dpunkt Verlag.

F. Abbildungsverzeichnis

Abb. 2.1	Die Funktionen zum Debuggen im Menü Debuggen	8
Abb. 2.2	Das Programm beim Debuggen	9
Abb. 2.3	Die nächste Anweisung im Debugger	10
Abb. 2.4	Der Wert von zaehler in einem DataTip	11
Abb. 2.5	Der markierte Aufruf der Methode Subtraktion()	13
Abb. 2.6	Die markierte Zeile in der Methode	14
Abb. 2.7	Die Fortsetzung des Programms in der Methode Main()	15
Abb. 3.1	Die erste Anweisung einer Windows Forms-Anwendung im Debugger	19
Abb. 3.2	Die erste Anweisung von Form1 im Debugger	20
Abb. 3.3	Der Debugger mit der Methode ButtonBerechnen_Click()	21
Abb. 3.4	Die Funktion Ausführen bis Cursor im Kontextmenü einer Quelltextzeile	22
Abb. 3.5	Das angehaltene Programm	23
Abb. 3.6	Ein gesetzter Haltepunkt	24
Abb. 3.7	Der erreichte Haltepunkt	25
Abb. 3.8	Das Register Lokal	27
Abb. 3.9	Die Anzeige weiterer Informationen für das Datenobjekt e	28
Abb. 3.10	Die Anzeige eines Wertes im Register Überwachen 1	29
Abb. 3.11	Der Wert von zahl2 ist zum Ändern markiert	30
Abb. 3.12	Der geänderte Wert für zahl2	31
Abb. 4.1	Grundprinzip eines funktionsorientierten Tests	34
Abb. 5.1	Die Meldung bei einem ungültigen Wert	40
Abb. 5.2	Die Reaktion auf eine Ausnahme im Debug-Modus	41
Abb. 5.3	Die Meldung für den Konvertierungsfehler	49

G. Tabellenverzeichnis

Tab. 4.1	Mögliche Testfälle für die Addition	35
Tab. 4.2	Äquivalenzklassen für die Addition	36
Tab. C.1	Tastenkombinationen zum Arbeiten mit dem Debugger	61
Tab. D.1	Standard-Exceptions	62

H. Codeverzeichnis

Code 1.1	Ein kleiner Fehler mit großer Wirkung	3
Code 1.2	Immer noch nicht das gewünschte Ergebnis	4
Code 1.3	Noch ein Fehler	4
Code 1.4	Vertauschte Argumente beim Aufruf einer Methode	5
Code 5.1	Eine drohende Division durch null	39
Code 5.2	try-catch-Konstruktion	43
Code 5.3	Ausnahmebehandlung für eine Division.....	43
Code 5.4	Gezielte Reaktion auf eine Exception-Klasse	44
Code 5.5	Reaktion auf verschiedene Ausnahmen	44
Code 5.6	Geschachtelte try-Blöcke	46
Code 5.7	Selbst ausgelöste Exceptions	48
Code 5.8	Der aufgelöste innere try-Block	49
Code 5.9	Die finally-Anweisung	50
Code 5.10	Eine gefilterte Ausnahme	51

I. Medienverzeichnis

Video 2.1	Erste Schritte mit dem Debugger	12
Video 2.2	Methoden debuggen	16
Video 3.1	Haltepunkte setzen und löschen	26
Video 3.2	Nützliche Debuggerfunktionen	31

J. Sachwortverzeichnis

A	O
Anwendung	Off-by-one-Fehler 4
schrittweise ausführen 8	
Äquivalenzklasse 36	
Ausnahmebehandlung 41	
B	P
Blackbox-Test 34	Prozedurschritt 15
Breakpoint 23	
C	S
Cursor	Schreibtischtest 36
Ausführen bis 22	Semantikfehler 3
D	Syntaxfehler 3
DataTip 11	
Datenobjekt	
Inhalte von, prüfen 11	
Debugger 7	
Debug-Modus	
Änderungen vornehmen 12	
E	T
Einzelschritt 8	Test 33
Entwicklung	heißer 34
testgetriebene 36	Testfall 35
Exception 40	Testmeldung 5
eigene auslösen 47	Testmethode 34
Handler 41	Testobjekt 34
Handling 41	Testverfahren
Klasse 44	funktionsorientiertes 34
Standard Handler 52	statisches 36
H	U
Haltepunkt 23	Überwachung
Arbeiten mit 22	hinzufügen 28
L	W
Laufzeitfehler 40	Windows Forms-Anwendung
M	debuggen 18
Methode	
debuggen 13	

K. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP09D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

1. Beschreiben Sie bitte, wie Sie einen Haltepunkt für den Debugger in einem Programm setzen.

5 Pkt.

2. Formulieren Sie bitte `catch`-Anweisungen für die Verarbeitung folgender Ausnahmen:

- a) Zugriff auf ein ungültiges Laufwerk,
- b) eine Division durch null bei ganzzahligen Typen,
- c) ein ungültiges Typecasting,
- d) ein Zugriff auf eine nicht vorhandene Datei,
- e) für sämtliche Ausnahmen, die auftreten können.

Sie müssen dabei jeweils nur die Zeile mit der `catch`-Anweisung angeben.

für jede richtige Anweisung 1 Pkt.

3. Sie vereinbaren für einen `try`-Block mehrere Exception Handler. In welcher Reihenfolge werden diese Exception Handler bearbeitet? An welcher Position muss sich der Exception Handler befinden, der alle denkbaren Ausnahmen gleichzeitig behandelt?

5 Pkt.

4. Wie werden Ausnahmen in Ihren Programmen behandelt, wenn Sie keine eigene Ausnahmebehandlung programmieren?

5 Pkt.

5. Programmieren Sie bitte für den Taschenrechner eine Ausnahmebehandlung, die Konvertierungsfehler bei den beiden Eingabefeldern für die Zahlen abfängt. Erstellen Sie dazu eine Methode, die sich prinzipiell für jede TextBox verwenden lässt. Wenn bei der Konvertierung eine Ausnahme auftritt, soll eine Meldung für den Anwender erscheinen, die ihn auf den Fehler hinweist und den ungültigen Wert noch einmal anzeigt. Außerdem soll der Eingabefokus auf das Feld gesetzt werden, das die Ausnahme ausgelöst hat. Die Ausgabe der Meldung und auch das Setzen des Fokus sollen in der Methode erfolgen.

Berücksichtigen Sie bei der Lösung bitte auch, dass der Taschenrechner keine Berechnung durchführen darf, wenn eine Ausnahme aufgetreten ist. Dazu können Sie zum Beispiel in der Methode für die Überprüfung eine eigene Ausnahme auslösen, die Sie dann in der Methode mit den Berechnungen auswerten.

Einige Hinweise zur Lösung:

Den Fokus können Sie mit der Anweisung `Name.Select()` setzen. `Name` steht dabei für das Steuerelement, das den Fokus erhalten soll.

Um ein Steuerelement komplett an eine Methode zu übergeben, vereinbaren Sie den Typ des Steuerelements als Parameter. Der Typ für eine TextBox ist zum Beispiel `TextBox`. Als Argument übergeben Sie dann den Namen des Steuerelements.

Schicken Sie für die Lösung bitte das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

80 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Grundlagen der Grafikprogrammierung

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP10D

Objektorientierte Software-Entwicklung mit C#

Grundlagen der Grafikprogrammierung

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Grundlagen der Grafikprogrammierung

Inhaltsverzeichnis

Einleitung	1
1 Die GDI+ API und die Klasse Graphics	3
1.1 Der Zugriff auf die Zeichenfläche	3
1.2 Das Beschaffen der Zeichenfläche	12
Zusammenfassung	15
2 Geometrische Figuren	17
2.1 Linien	17
2.2 Kreise und Ellipsen	20
2.3 Polygone	22
2.4 Weitere Figuren	24
Zusammenfassung	25
3 Gestaltung	27
3.1 Der Stift und der Pinsel	27
3.2 Verläufe	33
3.3 Texte in Grafiken	34
Zusammenfassung	36
4 Eine kleine Spielerei	38
4.1 Einige Vorüberlegungen	39
4.2 Das Formular	39
4.3 Das Zeichnen der Figuren	43
4.4 Die Farbauswahl	48
4.5 Die Listen für den Linienstil und das Hintergrundmuster	51
Zusammenfassung	58
Schlussbetrachtung	60

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	61
B.	Glossar	63
C.	Literaturverzeichnis	64
D.	Abbildungsverzeichnis	65
E.	Tabellenverzeichnis	66
F.	Codeverzeichnis	67
G.	Sachwortverzeichnis	68
H.	Einsendeaufgabe	69

Einleitung

In diesem Studienheft beschäftigen wir uns mit den Grundlagen der Grafikprogrammierung. Sie erfahren unter anderem, wie Sie Zeichnungen mit verschiedenen grafischen Objekten erstellen und wie Sie die Eigenschaften für diese Objekte setzen. Außerdem lernen Sie, wie Sie Listenfelder zur Laufzeit mit Grafiken versehen.

Im Einzelnen lernen Sie in diesem Studienheft,

- was sich hinter der GDI+ API und der Klasse `Graphics` verbirgt,
- wie Sie Rechtecke, Kreise, Linien und andere Figuren zeichnen,
- wie Sie die Eigenschaften dieser Figuren verändern,
- wie Sie Texte direkt in der Zeichenfläche ausgeben,
- wie Sie Farben über einen Dialog auswählen,
- wie Sie Drehfelder und Schieberegler in einem Formular verwenden und
- wie Sie in der Liste eines Listenfeldes zeichnen.

Am Ende dieses Studienhefts werden Sie eine Anwendung programmieren, in der der Anwender selbst Einstellungen für unterschiedliche Figuren vornehmen kann, die in dem Formular gezeichnet werden sollen.

Hinweis:

In diesem Studienheft zeigen wir Ihnen vor allem Ausschnitte aus Quelltexten, die wesentliche Techniken erklären. Vollständige Beispiele finden Sie in den Projekten im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. In den Projekten werden auch einige Techniken, die wir hier im Studienheft nur kurz beschreiben, praktisch umgesetzt. Ein Blick in die Quelltexte lohnt sich in jedem Fall zur Vertiefung.

Die Codes im Download-Bereich sind ausführlich kommentiert. Sie sollten daher keine Schwierigkeiten mit dem Verständnis haben.

Christoph Siebeck

1 Die GDI+ API und die Klasse Graphics

In diesem Kapitel lernen Sie die GDI+ API und die Klasse `Graphics` kennen.

Das Erzeugen von Grafiken in einer Windows Forms-Anwendung erfolgt im Wesentlichen über eine spezielle Schnittstelle – das *Graphics Device Interface + Application Programming Interface*¹ (kurz GDI+ API oder auch einfach nur GDI+ genannt). Diese Schnittstelle stellt unterschiedlichste Klassen für die Ausgabe von nahezu beliebigen Informationen auf Bildschirmen und Druckern zur Verfügung. Sie als Programmierer müssen „eigentlich“ nichts weiter machen, als die passenden Methoden der jeweiligen Klassen aufzurufen. Um den gesamten Rest – also zum Beispiel die Aufbereitung für ein bestimmtes Gerät – kümmert sich dann GDI+.

GDI+ ist eine geräteunabhängige Programmierschnittstelle für die Ausgabe von Informationen auf grafikfähigen Geräten. Die Schnittstelle wird vor allem in Windows Forms-Anwendungen eingesetzt.



Ein wesentliches Kernstück der GDI+ ist die Klasse `Graphics`. Sie stellt Ihnen zahlreiche Eigenschaften und Methoden für grafische Objekte zur Verfügung – zum Beispiel:

- das Zeichnen verschiedener geometrischer Figuren,
- das Ändern von Linienbreiten und Linienfarben,
- das Füllen von grafischen Objekten,
- das Darstellen von Texten und
- noch vieles mehr.

Außerdem kapselt die Klasse `Graphics` die Zeichenfläche für die Steuerelemente. So können Sie zum Beispiel im Formular selbst, auf Schaltflächen, in Tabellen und so weiter zeichnen.

Bei den Eigenschaften eines Steuerelements werden Sie die Zeichenfläche allerdings vergeblich suchen. Denn sie steht Ihnen nur zur Laufzeit einer Anwendung zur Verfügung. Das heißt, nahezu alle Aktionen, die mit dem Zeichnen zusammenhängen, müssen im Quelltext des Programms kodiert werden.

1.1 Der Zugriff auf die Zeichenfläche

Schauen wir uns den Einsatz der Klasse `Graphics` jetzt an einigen Beispielen an. Beginnen wir mit dem Zeichnen eines Rechtecks in ein Formular.

Legen Sie bitte eine neue Windows Forms-Anwendung mit Visual Studio an. Fügen Sie am rechten Rand des Formulars eine Schaltfläche ein, mit der die Anwendung geschlossen werden kann. Dazu benutzen Sie wie gewohnt die Anweisung

```
Close();
```

1. Frei übersetzt bedeutet *Graphics Device Interface Application Programming Interface* so viel wie „Schnittstelle für die Programmierung von Anwendungen für grafische Geräte“.

Das Rechteck lassen wir jetzt direkt beim Zeichnen des Formulars erstellen. Dazu legen wir eine Methode für das Ereignis **Paint**² des Formulars an. Sie finden dieses Ereignis in der Gruppe **Darstellung** im Eigenschaftenfenster.

Die vollständige Methode für das Ereignis finden Sie im folgenden Code. Was dort genau geschieht, erklären wir Ihnen im Anschluss.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    //einen schwarzen Stift erzeugen
    Pen stift = new Pen(Color.Black);
    //ein Rechteck in die Zeichenfläche des Formulars zeichnen
    e.Graphics.DrawRectangle(stift, 1, 1, 100, 100);
}
```

Code 1.1: Das Zeichnen eines Rechtecks

Zunächst einmal erzeugen wir mit der Anweisung

```
Pen stift = new Pen(Color.Black);
```

eine neue Instanz `stift` der Klasse `Pen`. Diese Klasse stellt den Stift für die folgenden Zeichnungen dar. Als Argument übergeben wir dabei die gewünschte Farbe an den Konstruktor. In unserem Beispiel wird also ein Stift in der Farbe Schwarz verwendet.



Die Klasse `Pen` steht für den Zeichenstift. Er bestimmt, wie die Linien der Zeichnungen dargestellt werden. Mehr zum Stift erfahren Sie im Kapitel 3 dieses Studienhefts.

Das eigentliche Zeichnen erfolgt dann mit der Anweisung

```
e.Graphics.DrawRectangle(stift, 1, 1, 100, 100);
```

Der Ausdruck `e.Graphics` bestimmt dabei die Zeichenfläche – also den Bereich, in dem gezeichnet wird. `e` ist vom Typ `PaintEventArgs` und wird beim Aufruf der Methode automatisch übergeben. Da wir die Methode für das Zeichnen des Formulars aufrufen, steht der Ausdruck `e.Graphics` also für die Zeichenfläche des Formulars.

Die Methode `DrawRectangle()`³ zeichnet das Rechteck. Als Argumente werden der Stift, die Koordinaten der linken oberen Ecke, die Breite und die Höhe angegeben. In unserem Beispiel befindet sich die linke obere Ecke also an den Koordinaten 1, 1. Die Breite und die Höhe betragen jeweils 100.

Bei den Koordinaten wird normalerweise erst die Horizontale – die X-Achse – und dann die Vertikale – die Y-Achse – angegeben. Das zweite Argument (die erste 1) von `DrawRectangle()` gibt also den Abstand vom linken Rand an und das dritte (die zweite 1) den Abstand vom oberen Rand.

Die Angaben beziehen sich dabei auf den Rand der Zeichenfläche. Die Koordinate 0, 0 steht normalerweise für den Punkt ganz links oben in der Zeichenfläche.

2. *Paint* bedeutet übersetzt „Zeichne“.

3. *Draw Rectangle* bedeutet übersetzt „Zeichne Rechteck“.

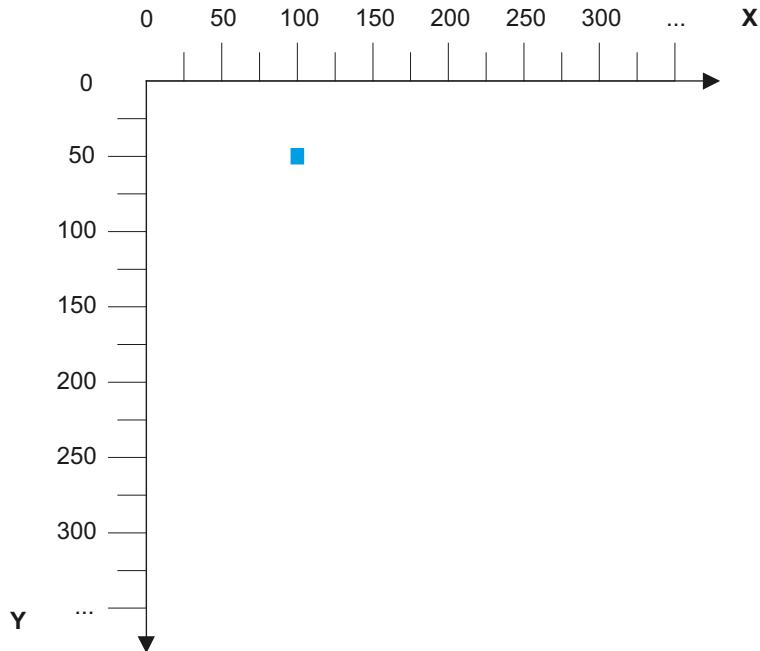


Abb. 1.1: Das Koordinatensystem von Graphics
(der markierte Punkt hat die Koordinaten 100, 50)

Testen Sie das Programm jetzt bitte. Es sollte ein Rechteck wie in der Abb. 1.2 gezeichnet werden.

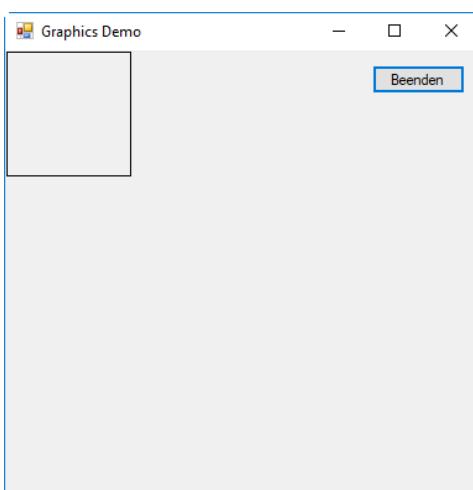


Abb. 1.2: Ein Rechteck im Formular

Bitte beachten Sie:

Es gibt für `Graphics` keinen festen rechten und unteren Rand. Sie können ohne Weiteres über die Zeichenfläche hinaus zeichnen. Probieren Sie das einfach einmal aus. Geben Sie zum Beispiel für die Breite und Höhe des Rechtecks jeweils den Wert 1 000 an. Sie werden sehen, die Figur wird ohne Beschwerde des Compilers gezeichnet. Allerdings sind der rechte und der untere Rand nicht mehr sichtbar.



Den maximalen unteren und rechten noch sichtbaren Rand der Zeichenfläche können Sie entweder über die Eigenschaft `ClientSize` oder über die Eigenschaft `ClientRectangle`⁴ ermitteln.

Die Eigenschaft `ClientSize` liefert Ihnen die Breite und die Höhe des Client-Bereichs als Typ `Size` zurück. Dieser Client-Bereich ist normalerweise auch der Bereich der Zeichenfläche. Auf die Höhe greifen Sie dann über den Ausdruck `ClientSize.Height` zu und auf die Breite über den Ausdruck `ClientSize.Width`.

Die Eigenschaft `ClientRectangle` liefert Ihnen zusätzlich noch die linke obere Ecke über den Typ `Rectangle`. Dieser Typ speichert vier ganze Zahlen, die die Position und Größe eines Rechtecks festlegen.

Die Daten, die `ClientRectangle` liefert, können Sie auch direkt an die Methode `DrawRectangle()` übergeben. Die einzelnen Koordinaten werden dabei automatisch in die richtige Form umgesetzt.

Probieren Sie das jetzt bitte aus. Fügen Sie die Anweisungen aus dem folgenden Code in die Anweisungen aus dem vorigen Code ein. Setzen Sie die Zeilen dabei hinter die Zeile mit dem Erzeugen des Stiftes.

```
//eine Variable vom Typ Rectangle
Rectangle bereich;
//den Client-Bereich beschaffen
bereich = ClientRectangle;
//ein Rechteck in der maximalen Größe zeichnen
e.Graphics.DrawRectangle(stift, bereich);
```

Code 1.2: Ein Rechteck in der Größe des Client-Bereichs

Mit der Anweisung

```
Rectangle bereich;
```

vereinbaren wir eine Variable `bereich` vom Typ `Rectangle`. Dieser Variablen weisen wir anschließend den Wert von `ClientRectangle` zu. Danach wird ein Rechteck mit der Größe des Client-Bereichs gezeichnet.

Hinweis:

Sie können das Rechteck auch direkt mit der Anweisung

```
e.Graphics.DrawRectangle(stift, ClientRectangle);
```

zeichnen lassen. Wir wollen aber gleich noch einige Veränderungen an der Variablen `bereich` vornehmen und gehen daher den kleinen Umweg.

Nun sollten beim Ausführen des Programms zwei Rechtecke in dem Formular erscheinen.

4. Wörtlich übersetzt bedeutet *Client Size* „Größe des Kunden“ und *Client Rectangle* „Rechteck des Kunden“.

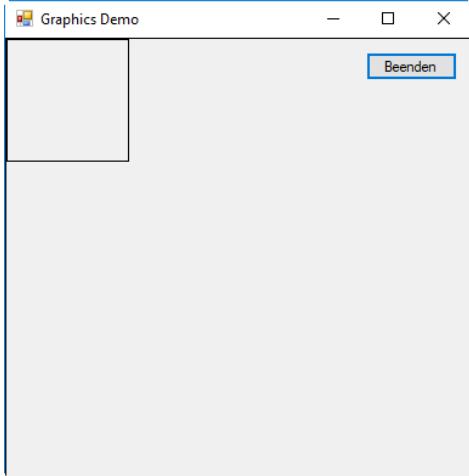


Abb. 1.3: Zwei Rechtecke in dem Formular

Das größere Rechteck ist allerdings nur sehr schwer zu sehen, da es direkt am beziehungsweise auf dem Rand des Formulars gezeichnet wird. Wir verändern daher einmal die Koordinaten der beiden Rechtecke so, dass sie deutlicher zu erkennen sind. Für die Breite und Höhe des ersten Rechtecks ziehen wir 10 Punkte vom Client-Bereich ab und setzen die Startkoordinaten auf 5, 5. Die Startkoordinaten des zweiten Rechtecks setzen wir fest auf 50, 50.

Der geänderte Code sieht dann so aus (zur besseren Übersicht haben wir die geänderten und neuen Anweisungen fett markiert):

```
//einen schwarzen Stift erzeugen
Pen stift = new Pen(Color.Black);
//eine Variable vom Typ Rectangle
Rectangle bereich;
//den Client-Bereich beschaffen
bereich = ClientRectangle;
//und anpassen
bereich.Width = bereich.Width - 10;
bereich.Height = bereich.Height - 10;
bereich.Location = new Point(5, 5);
//ein Rechteck in der maximalen Größe zeichnen
e.Graphics.DrawRectangle(stift, bereich);
//ein Rechteck in die Zeichenfläche des Formulars zeichnen
e.Graphics.DrawRectangle(stift, 50, 50, 100, 100);
```

Code 1.3: Veränderte Koordinaten über die Variable bereich

Die beiden Anweisungen

```
bereich.Width = bereich.Width - 10;
bereich.Height = bereich.Height - 10;
```

verändern die Breite beziehungsweise die Höhe jeweils um den Wert 10. Der Zugriff erfolgt dabei über die Eigenschaften `Width` und `Height`.

Danach setzen wir mit der Anweisung

```
bereich.Location = new Point(5, 5);
```

die obere linke Ecke des Bereichs auf den Wert 5, 5.

Hinweis:

Die Eigenschaft `Location` erwartet ein Wertepaar vom Typ `Point`. Sie können die obere linke Ecke aber auch über die Eigenschaften `Top` und `Left` setzen. `Top` steht dabei für den Abstand vom oberen Rand und `Left` für den Abstand vom linken Rand.

Danach werden dann die beiden Rechtecke gezeichnet. Die Ausgabe sollte nun so aussehen:

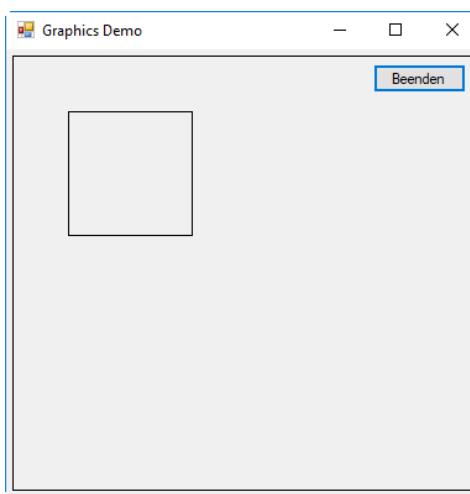


Abb. 1.4: Noch einmal zwei Rechtecke

Wenn Sie jetzt noch die Anweisungen zum Verändern der Werte über die Variable `bereich` sowie zum Zeichen des ersten Rechtecks in eine `for`-Schleife stecken und die zweite Anweisung zum Zeichnen des festen Rechtecks löschen, lässt sich schon ein erster netter grafischer Effekt erzielen.

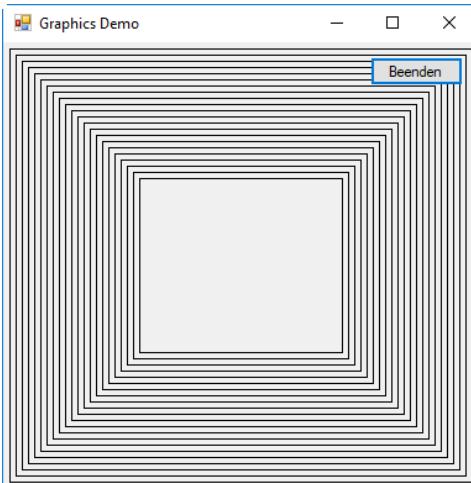
Der geänderte Code könnte so aussehen:

```
...
//den Client-Bereich beschaffen
bereich = ClientRectangle;
//in einer Schleife anpassen
for (int durchlauf = 0; durchlauf < 22; durchlauf++)
{
    //jeweils 10 von der Höhe und Breite abziehen
    bereich.Width = bereich.Width - 10;
    bereich.Height = bereich.Height - 10;
    //den Startpunkt setzen
    //bitte in einer Zeile eingeben
    bereich.Location = new Point(bereich.Location.X +
        5, bereich.Location.Y + 5);
```

```
//ein Rechteck zeichnen
e.Graphics.DrawRectangle(stift, bereich);
}
...
```

Code 1.4: Ein erster grafischer Effekt

Abhängig von Ihrer Bildschirmauflösung und der Größe Ihres Formulars werden immer kleinere Rechtecke in das Formular gezeichnet:

**Abb. 1.5:** Ein erster grafischer Effekt

Wie Sie in der Abb. 1.5 sehen, liegt die Schaltfläche über den Rechtecken. Die Zeichenfläche befindet sich also – wenn Sie so wollen – im Hintergrund des Formulars.

Mit einem kleinen Trick können wir die Zeichnungen auch animieren. Dazu lassen wir jedes gezeichnete Rechteck nach einer kurzen Pause wieder löschen. Auf dem Bildschirm entsteht dann der Eindruck, als würde das Rechteck wandern.

Für das Löschen der Zeichenfläche verwenden wir die Methode `Clear()`⁵ der Klasse `Graphics`. Als Argument erwartet die Methode die Farbe, mit der die Zeichenfläche nach dem Löschen gefüllt werden soll. Wir verwenden in unserem Beispiel einfach die Hintergrundfarbe des Formulars, die wir über die Eigenschaft `BackColor` erhalten.

Damit das Löschen und Neuzeichnen nicht unmittelbar hintereinander erfolgen, legen wir über die Methode `System.Threading.Thread.Sleep()`⁶ eine Pause ein. Als Argument erwartet die Methode die Länge der Pause in Millisekunden.

Die komplette Methode `Form1_Paint()` sieht dann so aus:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    //einen schwarzen Stift erzeugen
    Pen stift = new Pen(Color.Black);
    //eine Variable vom Typ Rectangle
    Rectangle bereich;
```

5. *Clear* bedeutet so viel wie „Lösche“.

6. Wörtlich übersetzt bedeutet *sleep* „Schlaf“.

```
//den Client-Bereich beschaffen
bereich = ClientRectangle;
//in einer Schleife anpassen
for (int durchlauf = 0; durchlauf < 22; durchlauf++)
{
    //jeweils 10 von der Höhe und Breite abziehen
    bereich.Width = bereich.Width - 10;
    bereich.Height = bereich.Height - 10;
    //den Startpunkt setzen
    //bitte in einer Zeile eingeben
    bereich.Location = new Point(bereich.
        Location.X + 5, bereich.Location.Y + 5);
    //ein Rechteck zeichnen
    e.Graphics.DrawRectangle(stift, bereich);
    //einen Moment warten
    System.Threading.Thread.Sleep(100);
    //den Zeichenbereich löschen
    e.Graphics.Clear(BackColor);
}
}
```

Code 1.5: Eine Animation

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich unter dem Namen **Cshp10d_p01**.

Das Programm läuft nicht richtig rund. So erscheint zum Beispiel direkt nach dem Start statt der Schaltfläche ein „Loch“. Das liegt daran, dass die Schaltfläche erst nach dem Zeichnen des Formulars erstellt wird – also erst, nachdem die Animation beendet ist. Außerdem wird die Animation beim Ausführen in der Entwicklungsumgebung nach dem Zeichnen der Schaltfläche noch einmal durchgeführt. Denn hier muss auch das Formular neu gezeichnet werden – und damit werden auch die Anweisungen in der Methode `Form1_Paint()` noch einmal ausgeführt.

Wir werden Ihnen gleich aber noch zeigen, wie Sie dieses Problem lösen können.

Eigentlich unterbricht die Anweisung `System.Threading.Thread.Sleep()` die Ausführung eines Threads⁷ – also eines eigenständigen Teils einer Anwendung, der getrennt von anderen Teilen der Anwendung läuft. Da unsere Anwendung aber nur aus einem einzigen Teil besteht, wird die gesamte Anwendung unterbrochen.

Die Anweisung `System.Threading.Thread.Sleep()` ist eine sehr einfache, aber auch nicht ganz unproblematische Variante, einen Programmablauf zu steuern. So kann es unter Umständen durch die Unterbrechung zu seltsamen Effekten wie Verzögerungen beim Zeichnen kommen.

Programmtechnisch sauberer lassen sich solche Konstruktionen über einen Timer lösen, der Anweisungen zeitgesteuert wiederholt. Sie könnten für den vorigen Code das Verkleinern ja zum Beispiel über einen Timer durchführen lassen und die Verarbeitung nach einer bestimmten Anzahl von Wiederholungen beenden. Dazu müssen Sie aber auch die Zeichenfläche mit anderen Techniken beschaffen. Damit werden uns gleich ebenfalls noch intensiver beschäftigen.

7. *Thread* bedeutet wörtlich übersetzt „Faden“ oder „Gewinde“.

Vorher wollen wir uns noch kurz ansehen, wie Sie auf die Zeichenfläche von einigen anderen Steuerelementen zugreifen.

Speichern Sie bitte die Änderungen am ersten Projekt. Legen Sie dann eine neue Windows Forms-Anwendung an und fügen Sie in diese Anwendung eine Schaltfläche, ein Label und eine PictureBox ein.

Lassen Sie dann im Ereignis **Paint** dieser drei Steuerelemente jeweils die folgenden Anweisungen ausführen:

```
Pen stift = new Pen(Color.Black);  
e.Graphics.DrawRectangle(stift, 1, 1, 10, 10);
```

Code 1.6: Zeichnen in verschiedenen Zeichenflächen

Damit wird in alle drei Steuerelemente ein kleines Rechteck gezeichnet. Das Formular könnte dann zum Beispiel so aussehen:

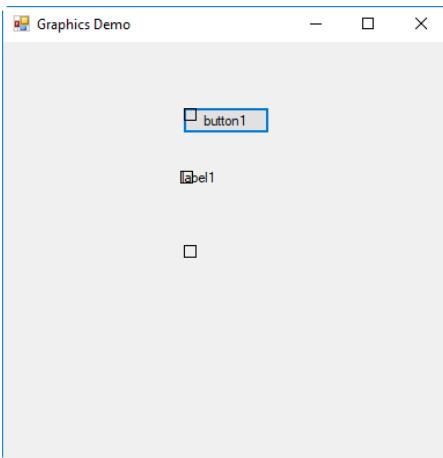


Abb. 1.6: Rechtecke in verschiedenen Zeichenbereichen

Wie Sie in der Abb. 1.6 sehen, beziehen sich die Koordinaten der Rechtecke jeweils auf den Zeichenbereich des angesprochenen Steuerelements. Bitte beachten Sie dabei, dass sich damit auch die Größe des Zeichenbereichs bei jedem Steuerelement ändert. Das können Sie ganz einfach ausprobieren, indem Sie die festen Koordinaten der Rechtecke im vorigen Code jeweils durch die Angabe `ClientRectangle` ersetzen.

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich unter dem Namen **Cshp10d_p02**.

Bisher haben wir auf die Zeichenfläche immer über den Typ `PaintEventArgs` zugegriffen. Das funktioniert auch ohne Probleme – solange der Typ zur Verfügung steht. Aber das ist nicht immer der Fall. Schauen wir uns auch dazu ein Beispiel an.

1.2 Das Beschaffen der Zeichenfläche

Wir wollen das Beispiel aus dem ersten Projekt jetzt einmal so ändern, dass die Animation nicht beim Zeichnen des Formulars ausgeführt wird, sondern über eine zweite Schaltfläche gestartet werden kann. Das hört sich erst einmal nicht weiter schwierig an.

Laden Sie bitte noch einmal das erste Projekt und fügen Sie eine weitere Schaltfläche ein. Legen Sie dann eine Methode für das Anklicken dieser Schaltfläche an und schneiden Sie die Anweisungen aus der Methode `Form1_Paint()` über die Zwischenablage aus. Fügen Sie anschließend die Anweisungen in der Methode für das Anklicken der zweiten Schaltfläche wieder ein.



Bevor Sie weiterlesen:

Überlegen Sie einmal selbst, ob das einfache Umstellen der Anweisungen zum Erfolg führt. Sehen Sie sich dazu den Typ des Parameters `e` der Methode für das Anklicken einer Schaltfläche einmal genau an.

Ganz so einfach funktioniert die Änderung leider nicht. Der Compiler meldet beim Übersetzen mehrere Fehler. Denn an die Methode zum Anklicken einer Schaltfläche wird zwar auch ein Parameter `e` übergeben – er ist allerdings vom Typ `EventArgs`. Und dieser Typ stellt keine Zeichenfläche zur Verfügung.

Wir müssen uns also die Zeichenfläche für das Formular selbst besorgen. Dazu vereinbaren Sie eine Variable vom Typ `Graphics` und beschaffen sich mit der Methode `CreateGraphics()`⁸ eine Referenz auf die gewünschte Zeichenfläche. Diese Referenz speichern Sie in der Variablen vom Typ `Graphics`.

Das könnte in unserem Beispiel so aussehen:

```
//eine Variable vom Typ Graphics
Graphics zeichenflaeche;
//die Variable auf die Zeichenfläche des Formulars setzen
zeichenflaeche = this.CreateGraphics();
```

Code 1.7: Das Beschaffen der Zeichenfläche für das Formular

Hinweis:

Die Angabe `this` vor dem Aufruf der Methode `CreateGraphics()` können Sie auch weglassen. Denn über `this` sprechen Sie ja die aktuelle Instanz des Formulars an – und diese Instanz wird auch ohne `this` verwendet.

Im zweiten Schritt ersetzen Sie dann den Ausdruck `e.Graphics` in den Anweisungen zum Zeichnen der Rechtecke durch den Ausdruck `zeichenflaeche`. Diese Referenz steht jetzt ja für die Zeichenfläche des Formulars.

8. Wörtlich übersetzt bedeutet *Create Graphics* „Erzeuge Grafiken“.

Die vollständige Methode würde so aussehen (neue und geänderte Anweisungen sind wieder fett markiert):

```
private void ButtonStarten_Click(object sender, EventArgs e)
{
    //eine Variable vom Typ Graphics
    Graphics zeichenflaeche;
    //die Variable auf die Zeichenfläche des Formulars setzen
    zeichenflaeche = this.CreateGraphics();
    //einen schwarzen Stift erzeugen
    Pen stift = new Pen(Color.Black);
    //eine Variable vom Typ Rectangle
    Rectangle bereich;
    //den Client-Bereich beschaffen
    bereich = ClientRectangle;
    //in einer Schleife anpassen
    for (int durchlauf = 0; durchlauf < 22; durchlauf++)
    {
        //jeweils 10 von der Höhe und Breite abziehen
        bereich.Width = bereich.Width - 10;
        bereich.Height = bereich.Height - 10;
        //den Startpunkt setzen
        //bitte in einer Zeile eingeben
        bereich.Location = new Point(bereich.Location.X + 5, bereich.Location.Y + 5);
        //ein Rechteck zeichnen
        zeichenflaeche.DrawRectangle(stift, bereich);
        //einen Moment warten
        System.Threading.Thread.Sleep(100);
        //den Zeichenbereich löschen
        zeichenflaeche.Clear(BackColor);
    }
}
```

Code 1.8: Der Start der Animation über eine Schaltfläche

Hinweis:

Der Name der Methode hängt vom Namen des Steuerelements ab. Das vollständige geänderte Projekt finden Sie im heftbezogenen Download-Bereich unter dem Namen **Cshp10d_p03**.

Probieren Sie die Änderungen jetzt einmal aus. Sie werden sehen, die Animation wird nach dem Anklicken der Schaltfläche gestartet – ohne „Löcher“ im Formular.

Mit dem Beschaffen der Zeichenfläche über die Methode `CreateGraphics()` lässt sich jetzt die Verzögerung durch die Methode `Sleep()` auch ohne Probleme durch einen Timer ersetzen. Die Methode `Tick()` für den Timer würde dann so aussehen:

```
private void Timer1_Tick(object sender, EventArgs e)
{
    //der Timer wird alle 100 Millisekunden aufgerufen
    //und zählt über das Feld aufrufe mit, wie oft er
    //bereits ausgeführt wurde
```

```

//die Zeichenfläche, der Pinsel und der
//Zeichenbereich werden einmal im Konstruktor über
//Felder gesetzt
//die Zeichenfläche löschen
//das geschieht jetzt vor dem Zeichnen, damit die
//Grafik einen Moment zu sehen ist
zeichenflaeche.Clear(BackColor);
//wenn aufrufe kleiner ist als 22, wird das
//Rechteck gezeichnet
if (aufrufe < 22)
{
    //jeweils 10 von der Höhe und Breite abziehen
    //bereich ist ein Feld
    bereich.Width = bereich.Width - 10;
    bereich.Height = bereich.Height - 10;
    //den Startpunkt setzen
    //bitte in einer Zeile eingeben
    bereich.Location = new Point(bereich.
        Location.X + 5, bereich.Location.Y + 5);
    //ein Rechteck zeichnen
    zeichenflaeche.DrawRectangle(stift, bereich);
    //aufrufe erhöhen
    aufrufe++;
}
}

```

Code 1.9: Das Steuern der Verzögerung über einen Timer



Damit der vorige Code ohne Fehlermeldungen übersetzt wird, sind noch einige Vereinbarungen erforderlich. Das vollständige Projekt für das Erstellen der Animation mit dem Timer finden Sie im heftbezogenen Download-Bereich im Projekt [Cshp10d_p01_timer](#).

Wichtig ist bei dieser Konstruktion auch, dass Sie die Zeichenfläche nicht direkt nach dem Erstellen der Zeichnung wieder löschen. Denn dann würde das Rechteck ja nur einen ganz kurzen Moment aufblitzen. Wir lassen daher die Zeichenfläche vor dem Zeichnen löschen und nehmen dabei in Kauf, dass diese Anweisung beim ersten Aufruf des Timers eigentlich gar nicht ausgeführt werden müsste. Wenn Sie das stört, können Sie das Löschen auch vom Wert des Feldes `aufrufe` abhängig machen. Es zählt ja mit, wie oft der Timer schon ausgelöst wurde.

Die Referenz für die Zeichenfläche können Sie übrigens auch auf andere Steuerelemente verbiegen. Im folgenden Code wird zum Beispiel nacheinander in drei Steuerelemente gezeichnet. Die Zeichenfläche wird dabei jeweils über den Handle `zeichenflaeche` angesprochen.

```

Graphics zeichenflaeche;
Pen stift = new Pen(Color.Black);
zeichenflaeche = button1.CreateGraphics();
zeichenflaeche.DrawRectangle(stift, 1, 1, 10, 10);
zeichenflaeche = pictureBox1.CreateGraphics();
zeichenflaeche.DrawRectangle(stift, 1, 1, 10, 10);

```

```
zeichenflaeche = label1.CreateGraphics();  
zeichenflaeche.DrawRectangle(stift, 1, 1, 10, 10);
```

Code 1.10: Eine verbogene Referenz für die Zeichenfläche

Probieren Sie diese Technik einmal im zweiten Projekt mit dem Zeichnen in die drei Steuerelemente aus. Löschen Sie die Methoden für die `Paint`-Ereignisse und lassen Sie die Anweisungen aus dem vorigen Code beim Anklicken der Schaltfläche ausführen. Das Ergebnis ist nahezu identisch. Lediglich das Rechteck auf der Schaltfläche verschwindet, wenn Sie den Mauszeiger von der Schaltfläche bewegen. Das liegt daran, dass dann die Schaltfläche neu gezeichnet werden muss. Und dabei wird das Rechteck sozusagen übermalt.

Hinweis:

Instanzen der Klasse `Graphics` und Instanzen anderer Klassen für grafische Operationen – wie zum Beispiel `Pen` – können zu einer Belastung des Systems führen – vor allem, wenn Sie mit vielen Instanzen arbeiten oder immer wieder neue Instanzen erzeugen. Wenn Sie diese Belastung vermeiden wollen, sollten Sie die Instanzen über die Methode `Dispose()` per Hand freigeben, wenn Sie sie nicht mehr brauchen. Achten Sie dabei aber sehr sorgfältig darauf, dass Sie die Instanzen nicht doch noch an einer anderen Stelle des Programms benötigen.

In unseren Beispielen in den Studienheften werden wir auf die manuelle Freigabe aber verzichten, da sich die Belastung nicht weiter auswirkt. Die Freigabe überlassen wir dem Garbage Collector.

So viel an dieser Stelle zur Klasse `Graphics`. Im nächsten Kapitel werden wir uns weitere geometrische Figuren ansehen.

Zusammenfassung

Alle Steuerelemente, in denen ein Programm zeichnen kann, verfügen über eine Zeichenfläche. Diese Zeichenfläche ist ein Objekt der Klasse `Graphics`.

Die Zeichenfläche arbeitet mit einem Koordinatensystem.

Die Zeichenfläche steht nur zur Laufzeit zur Verfügung. Sie kann nicht über das Eigenschaftenfenster verändert werden.

Sie können entweder über einen Parameter vom Typ `PaintEventArgs` oder über eine Variable vom Typ `Graphics` auf die Zeichenfläche zugreifen. Wenn Sie eine Variable verwenden, müssen Sie diese Variable mit der Methode `CreateGraphics()` positionieren.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Wo befinden sich folgende Punkte im Koordinatensystem von `Graphics`?
- a) 0, 0
 - b) 200, 20
 - c) 10, 50

Geben Sie bitte jeweils den Abstand vom linken und vom oberen Rand an.

- a) _____
- b) _____
- c) _____

- 1.2 Mit welchen Eigenschaften können Sie den maximal sichtbaren rechten und unteren Rand einer Zeichenfläche ermitteln?

- 1.3 Wie können Sie die Größe des zu zeichnenden Rechtecks an die Methode `DrawRectangle()` übergeben?

- 1.4 Mit welcher Methode löschen Sie die Zeichenfläche? Was müssen Sie dabei als Argument übergeben?

2 Geometrische Figuren

In diesem Kapitel stellen wir Ihnen weitere geometrische Figuren vor, die Sie über die Klasse `Graphics` zeichnen lassen können.

Hinweis:

Damit Sie die Beispiele in diesem Kapitel direkt nachvollziehen können, legen Sie bitte eine neue Windows Forms-Anwendung an. Bauen Sie in das Formular wieder eine Schaltfläche ein, mit der Sie die verschiedenen Zeichenaktionen starten können, und eine Schaltfläche für das Beenden des Programms. Damit die Beispiele besser aussehen, können Sie den Zeichenbereich des Formulars als Quadrat darstellen lassen. Dazu setzen Sie zunächst die Breite und die Höhe auf identische Werte. Addieren Sie anschließend für die Höhe 28 dazu und für die Breite 6. Diese Werte stehen für die Titelleiste und den Fensterrahmen, die ja nicht mit zum Zeichenbereich gehören.

Die verschiedenen Figuren aus diesem Kapitel finden Sie auch im Projekt `Cshp10d_p04` im heftbezogenen Download-Bereich.

2.1 Linien

Beginnen wir mit dem Zeichnen einer Linie.

Dazu verwenden Sie die Methode `DrawLine()`⁹ von `Graphics`. Als Argumente erwartet die Methode den Stift sowie die Start- und die Endposition. Die Linie endet dabei unmittelbar vor der Position, die Sie angegeben haben.

Im praktischen Einsatz finden Sie die Methode `DrawLine()` im Code 2.1. Er zeichnet ein Gitternetz in das Formular:

```
Graphics zeichenflaeche;
Pen stift;
//die Zeichenfläche beschaffen
zeichenflaeche = CreateGraphics();
//den Stift erzeugen
stift = new Pen(Color.Black);

//für die maximalen Koordinaten
//alternativ wäre auch ClientSize.Width - 1 bzw.
//ClientSize.Height - 1 möglich
int xMax = ClientRectangle.Right - 1;
int yMax = ClientRectangle.Bottom - 1;
//zum Wandern
int xPos = 0, yPos = 0;

//Linien von links nach rechts bis zum unteren Rand des
//Formulars
while (yPos < yMax)
{
```

9. `Draw Line` bedeutet übersetzt „Zeichne Linie“.

```

zeichenflaeche.DrawLine(stift, 0, yPos, xMax, yPos);
yPos = yPos + 10;
}

//Linien von oben nach unten bis zum rechten Rand des
//Formulars
while (xPos < xMax)
{
    zeichenflaeche.DrawLine(stift, xPos, 0, xPos, yMax);
    xPos = xPos + 10;
}

```

Code 2.1: Ein Gitternetz

Zunächst einmal beschaffen wir uns die Zeichenfläche des Formulars und erstellen den Stift. Das übernehmen die beiden Anweisungen

```

zeichenflaeche = CreateGraphics();
stift = new Pen(Color.Black);

```

Danach ermitteln wir mit den Anweisungen

```

int xMax = ClientRectangle.Right - 1;
int yMax = ClientRectangle.Bottom - 1;

```

die maximale Breite und Höhe des Zeichenbereichs. Von der Breite und Höhe ziehen wir dabei 1 ab, um exakt vor dem jeweiligen Rand mit dem Zeichnen aufzuhören.

Die erste Schleife zeichnet dann Linien vom linken zum rechten Rand. Bei jedem Schleifendurchlauf wird dabei der Abstand vom oberen Rand um 10 erhöht. Die Linien bewegen sich also immer weiter nach unten.

Die zweite Schleife zeichnet anschließend mit einer ähnlichen Technik die Linien von oben nach unten. Dabei verändern wir jeweils den Abstand vom linken Rand um 10. Die Linien bewegen sich also immer weiter nach rechts.

Das Ergebnis sollte dann so aussehen:

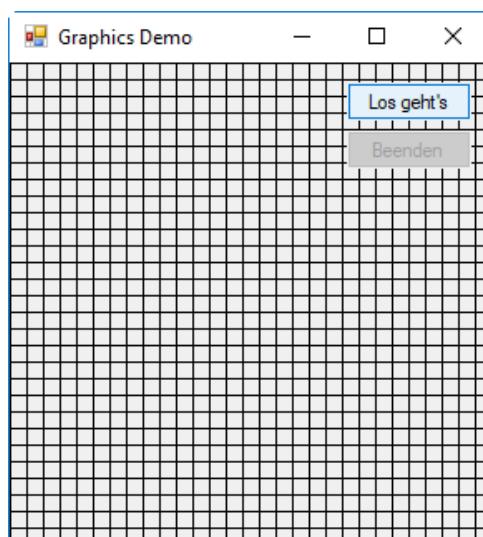


Abb. 2.1: Ein Gitternetz
(die Schaltflächen am Rand stammen aus dem Beispielprojekt **Cshp10d_p04**)

Wenn Sie mehrere Objekte hintereinander zeichnen, überlagern sich diese Objekte. Probieren Sie das einmal aus. Ergänzen Sie die folgenden fett markierten Anweisungen im vorigen Code:

```
...
//den Pinsel erstellen
Brush pinsel = new SolidBrush(Color.Blue);

...
//Linien von oben nach unten bis zum rechten Rand des
//Formulars
while (xPos < xMax)
{
    zeichenflaeche.DrawLine(stift, xPos, 0, xPos, yMax);
    xPos = xPos + 10;
}

//ein gefülltes Rechteck in die Mitte zeichnen
xPos = (ClientSize.Width / 2) - 50;
yPos = (ClientSize.Height / 2) - 50;
zeichenflaeche.FillRectangle(pinsel, xPos, yPos, 100, 100);

//zwei Diagonalen durch das Formular zeichnen
zeichenflaeche.DrawLine(stift, 0, 0, xMax, yMax);
zeichenflaeche.DrawLine(stift, 0, yMax, xMax, 0);
```

Code 2.2: Erweiterung zum vorigen Code

Zunächst einmal erstellen wir mit der Anweisung

```
Brush pinsel = new SolidBrush(Color.Blue);
```

einen Pinsel. Dieser Pinsel übernimmt zum Beispiel die Darstellung von Füllungen. Als Farbe verwenden wir Blau.

Die Klasse `Brush` steht für den Zeichenpinsel. Er bestimmt, wie die Füllungen der Zeichnungen dargestellt werden. Mit dem Pinsel werden wir uns in Kapitel 3 ebenfalls noch ausführlicher beschäftigen.



Die drei Anweisungen

```
xPos = (ClientSize.Width / 2) - 50;
yPos = (ClientSize.Height / 2) - 50;
zeichenflaeche.FillRectangle(pinsel, xPos, yPos, 100, 100);
```

zeichnen ein Quadrat mit der Kantenlänge 100 um den Mittelpunkt des Formulars. Diesen Mittelpunkt berechnen wir, indem wir die Breite und die Höhe des Client-Bereichs des Formulars durch 2 teilen. Danach ziehen wir von den beiden Werten 50 ab und erhalten so die linke obere Ecke des Quadrats.

Die folgenden zwei Anweisungen zeichnen Diagonalen durch das Formular. Die erste Diagonale läuft von links oben nach rechts unten und die zweite von rechts oben nach links unten.

```
zeichenflaeche.DrawLine(stift, 0, 0, xMax, yMax);
zeichenflaeche.DrawLine(stift, 0, yMax, xMax, 0);
```

Das Ergebnis sieht dann so aus:

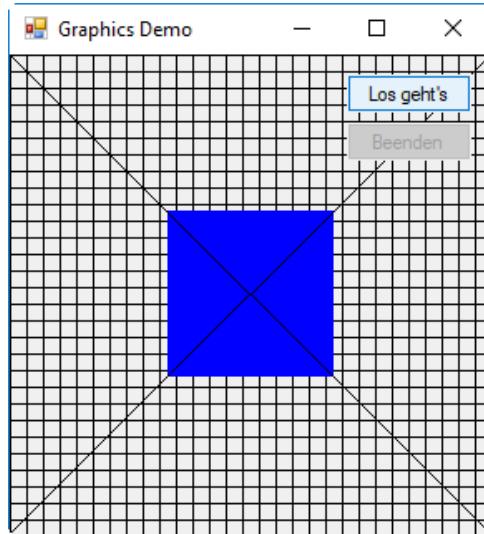


Abb. 2.2: Mehrere sich überlagernde Zeichenobjekte

2.2 Kreise und Ellipsen

Schauen wir uns nun an, wie Sie Kreise und Ellipsen zeichnen. Dazu verwenden Sie die Methode `DrawEllipse()`¹⁰ der Klasse `Graphics`. Die Methode arbeitet allerdings etwas ungewöhnlich: Sie geben nicht den Mittelpunkt und den Radius an, sondern Sie definieren über die Argumente ein Rechteck. In dieses Rechteck werden dann der Kreis beziehungsweise die Ellipse gezeichnet – und zwar genau so, dass die Kreisbahn die inneren Ränder des Rechtecks berührt.

Einige Beispiele:

```
//ein Kreis
zeichenflaeche.DrawEllipse(stift, 10, 10, 100, 100);
//eine Ellipse
zeichenflaeche.DrawEllipse(stift, 150, 150, 50, 100);
//ein Kreis in einem Quadrat
zeichenflaeche.DrawRectangle(stift, 150, 50, 100, 100);
zeichenflaeche.DrawEllipse(stift, 150, 50, 100, 100);
```

Code 2.3: Verschiedene Kreise

Hinweis:

Der Ausdruck `zeichenflaeche` im vorigen Code steht für eine Variable, die eine Referenz auf die Zeichenfläche des Formulars speichert. Wie Sie diese Variable erstellen und die Referenz beschaffen, haben Sie im letzten Kapitel gelernt.

10. *Draw Ellipse* bedeutet übersetzt „Zeichne Ellipse“.

Mit der ersten Anweisung wird ein Quadrat als umgebendes Rechteck definiert. Gezeichnet wird also ein Kreis.

Die zweite Anweisung dagegen erzeugt eine Ellipse, da das umgebende Rechteck nicht quadratisch ist.

Mit den letzten beiden Anweisungen schließlich wird zunächst ein Quadrat gezeichnet und anschließend mit denselben Koordinaten ein Kreis.

Bitte denken Sie daran:

Das dritte und vierte Argument bei einem Rechteck geben nicht direkt die Koordinaten der unteren rechten Ecke an. Sie legen hier die Breite und die Höhe des Rechtecks fest. Damit beschreiben Sie die Koordinaten der unteren rechten Ecke also nur indirekt.



Das Ergebnis dieser Anweisungen auf dem Bildschirm sieht so aus:

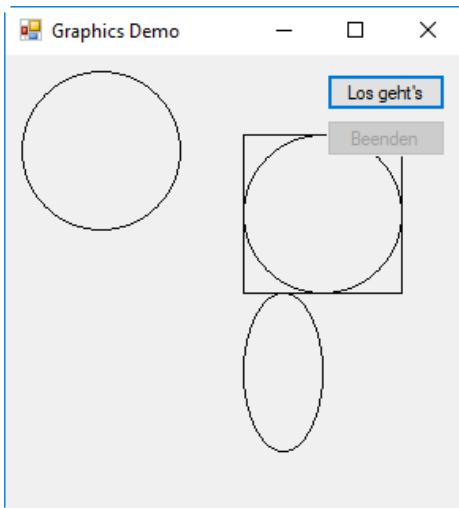


Abb. 2.3: Kreise und Ellipsen

Die erste Anweisung aus dem vorigen Code erzeugt den Kreis links oben, die zweite Anweisung die Ellipse in der Mitte und die beiden letzten Anweisungen schließlich die Figur oberhalb der Ellipse.

Genau wie bei der Methode `DrawRectangle()` können Sie auch an die Methode `DrawEllipse()` ein Argument vom Typ `Rectangle` übergeben. Die folgende Anweisung würde zum Beispiel einen Kreis beziehungsweise eine Ellipse in der maximalen Größe der Zeichenfläche erzeugen:

```
zeichenflaeche.DrawEllipse (stift, ClientRectangle);
```

2.3 Polygone

Eine weitere geometrische Figur ist das Polygon – ein Vieleck. Es handelt sich um geschlossene Figuren mit mehreren Linien. Ein typisches Polygon sieht zum Beispiel so aus:

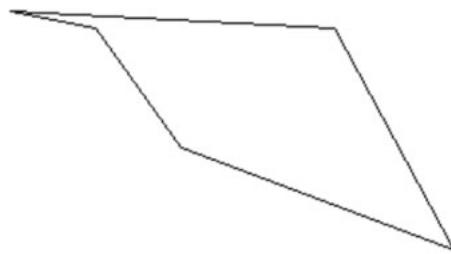


Abb. 2.4: Ein Polygon

Für das Zeichnen eines Polygons verwenden Sie die Methode `DrawPolygon()` der Klasse `Graphics`. Im ersten Schritt müssen Sie dazu zunächst einmal ein Array vom Typ `Point` anlegen. Dieses Array nimmt die Koordinaten der einzelnen Eckpunkte des Polygons auf. Die Eckpunkte werden dabei ebenfalls als Typ `Point` angegeben.

Die Eckpunkte für ein Polygon mit fünf Ecken könnten zum Beispiel so festgelegt werden:

```
Point [] punkte = new Point [5];
punkte[0] = new Point(10,10);
punkte[1] = new Point(60,20);
punkte[2] = new Point(110,90);
punkte[3] = new Point(270,150);
punkte[4] = new Point(200,20);
```



Noch einmal zur Auffrischung, weil es immer wieder für Verwirrung sorgt:

Bei der Vereinbarung für ein Array geben Sie die tatsächliche Anzahl der Elemente an. Das erste Element wird aber durch den Index 0 angesprochen und das letzte Element durch den Index Anzahl –1.

Um das Polygon zu zeichnen, rufen Sie die Methode `DrawPolygon()` auf und übergeben den Stift und das Array mit den Eckpunkten. Für unser Beispiel würde der Aufruf so aussehen:

```
zeichenflaeche.DrawPolygon(stift, punkte);
```

Die Methode zeichnet dann Linien zwischen den Eckpunkten. Vom letzten Punkt wird automatisch eine Linie zum ersten Punkt gezogen. Damit entsteht eine geschlossene Form.

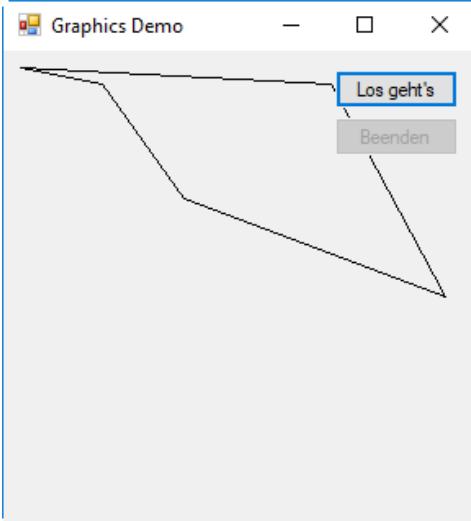


Abb. 2.5: Das Polygon im Formular

Wenn die Form nicht geschlossen werden soll, verwenden Sie die Methode `DrawLines()` der Klasse `Graphics`. Sie arbeitet fast genauso wie die Methode `DrawPolygon()`, zeichnet aber nur eine Linie über mehrere Punkte.

Bitte beachten Sie:

Die Methode `DrawLine()` zeichnet **eine** Linie, die Methode `DrawLines()` **dagegen mehrere**. Den Unterschied können Sie sich ganz einfach an dem s merken. *Lines* ist die Mehrzahl und *Line* die Einzahl.



Mit dem Array `punkte` von oben könnte der Aufruf von `DrawLines()` so erfolgen:

```
zeichnenflaeche.DrawLines(stift, punkte);
```

Das Ergebnis im Formular sieht dann so aus:

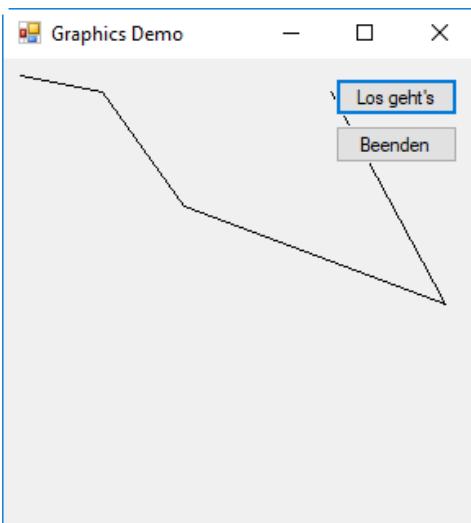


Abb. 2.6: Eine Linie über mehrere Punkte

2.4 Weitere Figuren

Abschließend stellen wir Ihnen noch kurz einige weitere Methoden für das Zeichnen von geometrischen Figuren vor.

Tab. 2.1: Weitere Methoden für geometrische Figuren

Methode	Wirkung
DrawArc()	Es wird ein Kreisbogen gezeichnet.
DrawClosedCurve()	Es wird eine geschlossene Kurve gezeichnet.
DrawCurve()	Es wird eine Kurve gezeichnet.
DrawPie()	Es wird eine Art „Tortenstück“ gezeichnet.
FillEllipse()	Es werden ein gefüllter Kreis beziehungsweise eine gefüllte Ellipse gezeichnet. Die Füllung wird über den Pinsel bestimmt.
FillPie()	Es wird ein gefülltes „Tortenstück“ gezeichnet. Die Füllung wird über den Pinsel bestimmt.
FillPolygon()	Es wird ein gefülltes Polygon gezeichnet. Die Füllung wird über den Pinsel bestimmt.

Details zum Einsatz dieser und noch weiterer Methoden zum Zeichnen finden Sie in der Hilfe. Suchen Sie dazu in der Hilfe nach dem Eintrag **Graphics-Klasse**. Klicken Sie dann auf den Eintrag **Methoden** oben im Fenster.

Methoden

- AddMetafileComment (Byte[])

Fügt der aktuellen [Metafile](#) einen Kommentar hinzu.
- BeginContainer()

Speichert einen Grafikcontainer mit dem aktuellen Zustand dieses [Graphics](#) und öffnet und verwendet einen neuen Grafikcontainer.
- BeginContainer (Rectangle, Rectangle, GraphicsUnit)

Speichert einen Grafikcontainer mit dem aktuellen Zustand dieses [Graphics](#) und öffnet und verwendet einen neuen Grafikcontainer mit der angegebenen Skalierungstransformation.
- BeginContainer (RectangleF, RectangleF, GraphicsUnit)

Speichert einen Grafikcontainer mit dem aktuellen Zustand dieses [Graphics](#) und öffnet und verwendet einen neuen Grafikcontainer mit der angegebenen Skalierungstransformation.
- Clear(Color)

Löscht die gesamte Zeichenoberfläche und füllt sie mit der angegebenen Hintergrundfarbe aus.
- CopyFromScreen(Int32, Int32, Int32, Int32, Size)

Führt entsprechend einem Rechteck aus Pixeln einen Bitblocktransfer der Farbdaten vom Bildschirm auf die Zeichenoberfläche des [Graphics](#) aus.
- CopyFromScreen(Int32, Int32, Int32, Int32, Size, CopyPixelOperation)

Führt entsprechend einem Rechteck aus Pixeln einen Bitblocktransfer der Farbdaten vom Bildschirm auf die Zeichenoberfläche des [Graphics](#) aus.
- CopyFromScreen(Point, Point, Size)

Führt entsprechend einem Rechteck aus Pixeln einen Bitblocktransfer der Farbdaten vom Bildschirm auf die Zeichenoberfläche des [Graphics](#) aus.

Abb. 2.7: Die Hilfe zur Klasse `Graphics`

So viel zu den verschiedenen grafischen Formen. Im nächsten Kapitel werden wir uns um die Gestaltung kümmern – wir bringen etwas Farbe ins Spiel.

Zusammenfassung

Mit der Methode `DrawLine()` zeichnen Sie eine Linie.

Neben Rechtecken und Linien können Sie mit den Methoden der Klasse `Graphics` auch Ellipsen, Polygone und weitere geometrische Figuren zeichnen.

Aufgaben zur Selbstüberprüfung

- 2.1 Sie wollen eine Linie von der Position 10, 10 zur Position 100, 100 zeichnen.
Mit welcher Anweisung lassen Sie die Linie erstellen?

- 2.2 Woher erhält die Methode `DrawPolygon()` die Koordinaten für die Eckpunkte?

- 2.3 Was unterscheidet die Methoden `DrawPolygon()` und `DrawLines()`?

- 2.4 Beschreiben Sie bitte, wie die Methode `DrawEllipse()` einen Kreis zeichnet.
Was wird über die fünf Argumente von `DrawEllipse()` dargestellt?

3 Gestaltung

In diesem Kapitel stellen wir Ihnen verschiedene Möglichkeiten zur Gestaltung von Zeichnungen vor. Sie erfahren unter anderem, wie Sie die Stift- und die Füllfarbe setzen und wie Sie die Linienbreite und das Füllmuster ändern.

Hinweis:

Legen Sie auch für dieses Kapitel bitte wieder eine neue Windows Forms-Anwendung an, damit Sie die Beispiele sofort nachvollziehen können.

Die verschiedenen Techniken aus diesem Kapitel finden Sie auch im Projekt Cshp10d_p05 im heftbezogenen Download-Bereich.

3.1 Der Stift und der Pinsel

Die Gestaltung der Zeichnungen erfolgt über die Klassen `Pen` und `Brush`. Wie Sie ja bereits wissen, ist die Klasse `Pen` – übersetzt „Stift“ – für die Linien zuständig und die Klasse `Brush` – übersetzt „Pinsel“ – für die Füllung.

Schauen wir uns zuerst einige wichtige Eigenschaften der Klasse `Pen` an.

Über die Eigenschaft `Color` setzen Sie zum Beispiel die Farbe, in der gezeichnet wird. Die Farbe muss dabei vom Typ `Color` sein.

Tipp:

Eine Liste der verschiedenen Farben finden Sie in der Hilfe unter dem Eintrag **Color-Struktur**.

Ein einfaches Beispiel:

```
stift.Color = Color.Red;
zeichenflaeche.DrawRectangle(stift, 10, 10, 100, 100);
```

Mit der ersten Anweisung wird die Farbe für einen Stift `stift` auf Rot gesetzt. Danach wird ein Quadrat gezeichnet. Der Rand dieses Quadrats ist jetzt nicht mehr schwarz, sondern rot.

Zur Erinnerung:

Vor den Namen einer Farbe müssen Sie die Angabe `Color.` setzen. Andernfalls erhalten Sie eine Meldung, dass der Bezeichner nicht bekannt ist.



Über die Eigenschaft `DashStyle`¹¹ setzen Sie den Linienstil für den Stift. Einige mögliche Werte für `DashStyle` finden Sie in der Tab. 3.1:

Tab. 3.1: Die Eigenschaft `DashStyle` der Klasse `Pen`¹²

Wert	Darstellung
<code>System.Drawing.Drawing2D.DashStyle.Dash</code>	eine gestrichelte Linie
<code>System.Drawing.Drawing2D.DashStyle.DashDot</code>	eine Linie, in der sich Striche und Punkte abwechseln
<code>System.Drawing.Drawing2D.DashStyle.DashDotDot</code>	eine Linie, in der sich Striche und zwei Punkte abwechseln
<code>System.Drawing.Drawing2D.DashStyle.Dot</code>	eine Linie aus Punkten
<code>System.Drawing.Drawing2D.DashStyle.Solid</code>	eine durchgezogene Linie

Über die Eigenschaften `StartCap` und `EndCap`¹³ der Klasse `Pen` setzen Sie den Anfang und das Ende einer Linie. Sie können hier zum Beispiel über den Wert `System.Drawing.Drawing2D.LineCap.DiamondAnchor` ein rautenförmiges Ende darstellen lassen. Einige weitere mögliche Werte für die Eigenschaften `StartCap` und `EndCap` finden Sie in der Tab. 3.2:

Tab. 3.2: Die Eigenschaften `StartCap` und `EndCap` für die Klasse `Pen`

Wert	Darstellung
<code>System.Drawing.Drawing2D.LineCap.ArrowAnchor</code>	eine Pfeilspitze, die dicker dargestellt wird
<code>System.Drawing.Drawing2D.LineCap.Flat</code>	ein abgeflachtes Ende
<code>System.Drawing.Drawing2D.LineCap.Round</code>	ein rundes Ende
<code>System.Drawing.Drawing2D.LineCap.RoundAnchor</code>	ein rundes Ende, das dicker dargestellt wird
<code>System.Drawing.Drawing2D.LineCap.Square</code>	ein quadratisches Ende
<code>System.Drawing.Drawing2D.LineCap.SquareAnchor</code>	ein quadratisches Ende, das dicker dargestellt wird
<code>System.Drawing.Drawing2D.LineCap.Triangle</code>	ein dreieckiges Ende

Mit der Eigenschaft `width` der Klasse `Pen` schließlich setzen Sie die Breite der Linie.

Kommen wir nun zur Klasse `Brush`. Hier gibt es verschiedene weitere abgeleitete Klassen für unterschiedliche Einsatzgebiete.

11. *Dash Style* bedeutet wörtlich übersetzt „Stil des Gedankenstrichs“.

12. *Dot* bedeutet übersetzt „Punkt“ und *solid* „geschlossen, fest“.

13. *Cap* bedeutet übersetzt „Kappe“.

Die Klasse `SolidBrush` legt zum Beispiel einen Pinsel für einfache Füllfarben fest. Welche Farbe für die Füllung verwendet wird, bestimmt die Eigenschaft `Color`.

Über die Klasse `System.Drawing.Drawing2D.HatchBrush`¹⁴ können Sie auch einen Pinsel mit Füllmustern verwenden. Dazu übergeben Sie an den Konstruktor der Klasse das gewünschte Muster sowie die Vordergrundfarbe und die Hintergrundfarbe. Einige mögliche Werte für die Muster finden Sie in der Tab. 3.3:

Tab. 3.3: Einige Muster der Klasse `System.Drawing.Drawing2D.HatchBrush`

Wert	Darstellung
<code>System.Drawing.Drawing2D.HatchStyle.BackwardDiagonal</code>	ein Muster aus diagonalen Linien von rechts oben nach links unten
<code>System.Drawing.Drawing2D.HatchStyle.Cross</code>	ein Muster aus horizontalen und vertikalen Linien, die sich kreuzen
<code>System.Drawing.Drawing2D.HatchStyle.DottedGrid</code>	ein Gittermuster aus gepunkteten Linien
<code>System.Drawing.Drawing2D.HatchStyle.ForwardDiagonal</code>	ein Muster aus diagonalen Linien von links oben nach rechts unten
<code>System.Drawing.Drawing2D.HatchStyle.Sphere</code>	ein Muster aus aneinander gereihten Kreisen
<code>System.Drawing.Drawing2D.HatchStyle.Vertical</code>	ein Muster aus vertikalen Linien
<code>System.Drawing.Drawing2D.HatchStyle.Wave</code>	ein wellenförmiges Muster
<code>System.Drawing.Drawing2D.HatchStyle.ZigZag</code>	ein Zickzackmuster

Bitte beachten Sie:

Sie müssen die Eigenschaften für einen Pinsel mit Mustern über den Konstruktor setzen. Eine nachträgliche Änderung ist nicht möglich.



Eine Demonstration der verschiedenen Eigenschaften für `Pen` und `Brush` finden Sie in den folgenden Codes. Der erste Code zeichnet erst einige Linien in das Formular und dann einige Quadrate. Der zweite Code setzt unterschiedliche Linienenden für den Stift.

```
//die Zeichenfläche löschen
zeichenflaeche.Clear(BackColor);
//die Farbe des Stiftes ändern
stift.Color = Color.Red;
//ein Rechteck zeichnen
zeichenflaeche.DrawRectangle(stift, 10, 10, 100, 100);
```

14. *Hatch* bedeutet so viel wie „Schraffiere“.

```
//jetzt wird der Stift grün
stift.Color = Color.Green;
//der Liniestil gestrichelt
//bitte in einer Zeile eingeben
stift.DashStyle = System.Drawing.Drawing2D.
DashStyle.DashDotDot;
//und der Stift wird dicker
stift.Width = 3;
//eine Linie zeichnen
zeichenflaeche.DrawLine(stift, 0, 120, 100, 120);
//und die Einstellungen noch einmal ändern
stift.Color = Color.Blue;
stift.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;
zeichenflaeche.DrawLine(stift, 0, 140, 100, 140);
//und noch einmal
stift.DashStyle = System.Drawing.Drawing2D.DashStyle.Solid;
stift.Width = 10;
zeichenflaeche.DrawLine(stift, 0, 160, 100, 160);
//nun auch den Pinsel ändern
pinsel.Color = Color.Red;
zeichenflaeche.FillRectangle(pinsel, 150, 10, 90, 90);
//einen Pinsel für Muster erzeugen
//bitte in einer Zeile eingeben
System.Drawing.Drawing2D.HatchBrush musterPinsel = new
System.Drawing.Drawing2D.HatchBrush(System.Drawing.
Drawing2D.HatchStyle.Cross, Color.Blue, Color.Green);
zeichenflaeche.FillRectangle(musterPinsel, 150, 110, 90, 90);
```

Code 3.1: Linien und Füllmuster

Hinweis:

Die Vereinbarungen für die Zeichenfläche, den Stift und den Pinsel sowie das Beschaffen der Zeichenfläche haben wir im Code aus Platzgründen weggelassen. Sie finden diese Vereinbarungen im Projekt **Cshp10d_p05** in der Klasse `Form1`. Das Beschaffen der Zeichenfläche erfolgt im Projekt in der Methode `Form1_Load()`.

Das Ergebnis sollte ungefähr so aussehen:

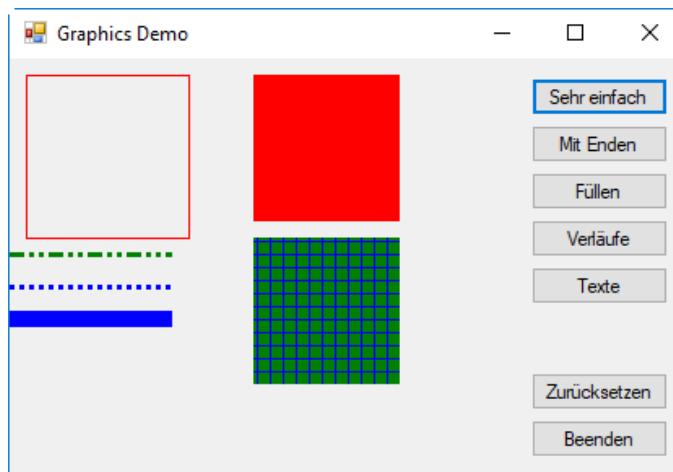
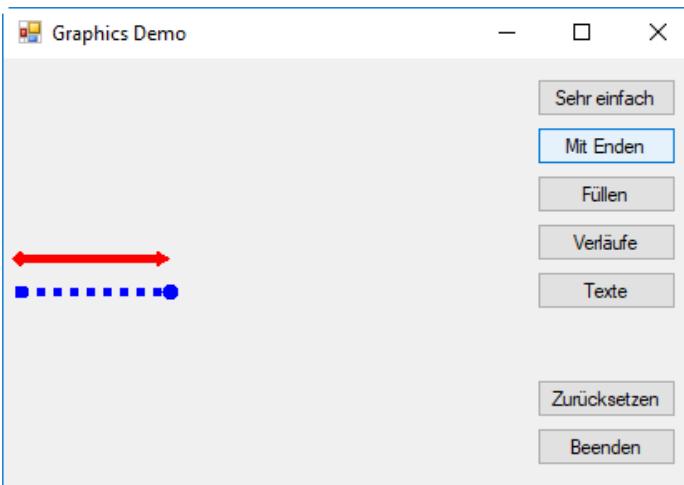


Abb. 3.1: Unterschiedliche Linien und Füllmuster
(die Schaltflächen am Rand stammen aus dem Beispielprojekt **Cshp10d_05**)

```
//die Zeichenfläche löschen  
zeichenflaeche.Clear(BackColor);  
//die Farbe des Stiftes ändern  
stift.Color = Color.Red;  
//die Linienenden setzen  
//bitte jeweils in einer Zeile eingeben  
stift.StartCap = System.Drawing.Drawing2D.  
LineCap.DiamondAnchor;  
stift.EndCap = System.Drawing.Drawing2D.  
LineCap.ArrowAnchor;  
//und der Stift wird dicker  
stift.Width = 5;  
//eine Linie zeichnen  
zeichenflaeche.DrawLine(stift, 10, 120, 100, 120);  
//und die Einstellungen noch einmal ändern  
stift.Color = Color.Blue;  
stift.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;  
//bitte in einer Zeile eingeben  
stift.StartCap = System.Drawing.Drawing2D.  
LineCap.SquareAnchor;  
stift.EndCap = System.Drawing.Drawing2D.LineCap.RoundAnchor;  
zeichenflaeche.DrawLine(stift, 10, 140, 100, 140);
```

Code 3.2: Unterschiedliche Linienenden

Die Linien aus dem vorigen Code würden ungefähr so aussehen:

**Abb. 3.2:** Unterschiedliche Linienenden

Wenn Ihnen die Standardfüllmuster für einen Pinsel nicht gefallen, können Sie auch eigene Muster verwenden. Dazu erstellen Sie zum Beispiel eine Bitmap-Grafik im BMP-Format und erzeugen dann mit dieser Grafik eine neue Instanz der Klasse `TextureBrush`¹⁵.

15. *Texture* bedeutet übersetzt „Textur“ oder „Gewebe“.

Der Code 3.3 definiert zum Beispiel einen kleinen „Smiley“ als Füllmuster und zeichnet dann ein Rechteck.

```
zeichnenflaeche.Clear(BackColor);
try
{
    //eine Referenz auf eine Bitmap, die Bitmap wird aus
    //einer Datei geladen
    //der Pfad muss ggf. angepasst werden!!
    //bitte in einer Zeile eingeben
    Bitmap bild = (Bitmap)(Image.FromFile
        ("c:\\beispiele\\smiley.bmp"));
    //einen neuen Pinsel mit einer Grafik erzeugen
    TextureBrush grafikPinsel = new TextureBrush(bild);
    //die Darstellung soll gekachelt erfolgen
    //bitte jeweils in einer Zeile eingeben
    grafikPinsel.WrapMode = System.Drawing.Drawing2D.
    WrapMode.Tile;
    zeichenflaeche.FillRectangle(grafikPinsel, 10, 10,
        240, 240);
}
catch(System.IO.FileNotFoundException)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Die Datei smiley.bmp ist nicht
        vorhanden.", "Fehler beim Laden");
}
```

Code 3.3: Eine Grafik als Füllmuster

Im `try`-Block erstellen wir eine Instanz der Klasse `Bitmap` und weisen dieser Instanz eine Referenz auf ein `Image`-Objekt zu, das wir über die Methode `Image.FromFile()` aus der Datei `smiley.bmp` laden. Da die Methode `Image.FromFile()` eine Rückgabe vom Typ `Image` liefert, wandeln wir das Ergebnis über ein Casting in den Typ `Bitmap` um.



Zur Auffrischung:

Bei Pfadnamen müssen Sie das Trennzeichen \ doppelt angeben. Alternativ können Sie auch das Zeichen @ vor die Zeichenkette mit dem Pfad stellen.

Anschließend erzeugen wir dann einen Pinsel vom Typ `TextureBrush` und weisen diesem Pinsel die `Bitmap` zu. Danach setzen wir die Darstellung für den Pinsel auf `gekachelt` und lassen ein Rechteck mit dem Füllmuster zeichnen.

Im `catch`-Block überprüfen wir lediglich, ob die Datei geladen werden konnte. Wenn das nicht der Fall war, geben wir eine Meldung aus.

Hinweis:

Die Datei mit dem Smiley finden Sie im heftbezogenen Download-Bereich bei den Beispielen. Bitte kopieren Sie die Datei vor dem Test des vorigen Codes in den Ordner `c:\beispiele` oder passen Sie den Pfad im Code entsprechend an.

3.2 Verläufe

Neben einfachen Füllungen oder Füllungen mit Mustern können Sie auch Füllungen mit Verläufen benutzen. Dazu verwenden Sie als Pinsel eine Instanz der Klasse `System.Drawing.Drawing2D.LinearGradientBrush`¹⁶. An den Konstruktor übergeben Sie dabei den Start- und Endpunkt für den Verlauf als Typ `Point` sowie die Start- und Endfarbe.

Die folgende Anweisung erzeugt zum Beispiel einen Pinsel `verlaufPinsel` mit einem Farbverlauf von Rot nach Blau. Der Verlauf beginnt dabei an der Koordinate 10, 10 und endet an der Koordinate 300, 300.

```
System.Drawing.Drawing2D.LinearGradientBrush  
verlaufPinsel = new System.Drawing.Drawing2D.  
LinearGradientBrush(new Point(10,10), new  
Point(300,300), Color.Red, Color.Blue);
```

Je weiter unten rechts sich ein Punkt in diesem Bereich befindet, desto stärker ist der blaue Anteil – oder andersherum ausgedrückt: Je weiter oben links sich der Punkt befindet, desto stärker ist der rote Anteil.

Im praktischen Einsatz finden Sie einen Pinsel für einen Verlauf im Code 3.4. Er zeichnet einige Figuren mit unterschiedlichen Verläufen.

```
zeichenflaeche.Clear(BackColor);  
//ein linearer Farbverlauf  
//einen neuen Pinsel erzeugen mit den Punkten und den Farben  
//bitte in einer Zeile eingeben  
System.Drawing.Drawing2D.LinearGradientBrush  
verlaufPinsel = new System.Drawing.Drawing2D.  
LinearGradientBrush(new Point(10,10), new  
Point(300,300),Color.Red, Color.Blue);  
//und nun einige Sachen zeichnen  
zeichenflaeche.FillRectangle(verlaufPinsel, 10, 10, 50, 50);  
zeichenflaeche.FillEllipse(verlaufPinsel, 10, 60, 50, 50);  
zeichenflaeche.FillRectangle(verlaufPinsel, 70, 10, 200, 200);  
//den Pinsel für den Stift auf den Verlaufspinsel setzen  
stift.Brush = verlaufPinsel;  
//und nun eine dicke Linie mit dem Verlauf zeichnen  
stift.Width = 10;  
zeichenflaeche.DrawLine(stift, 10, 250, 270, 250);
```

Code 3.4: Einige Verläufe

16. *Gradient* bedeutet übersetzt so viel wie „Steigung, Gefälle“.

Das Ergebnis auf dem Bildschirm sieht so aus:

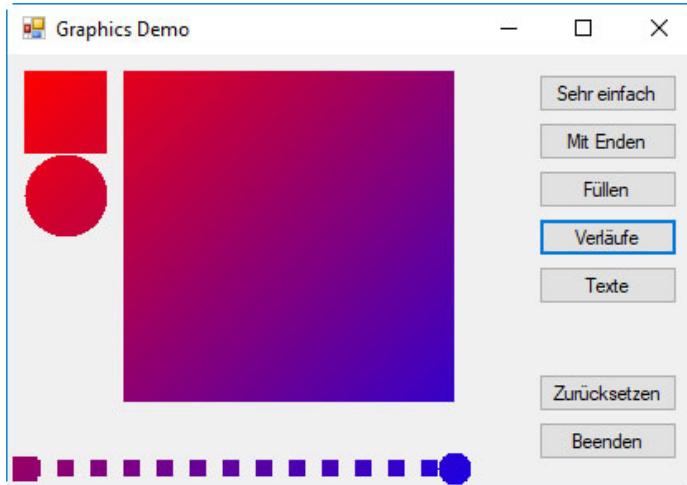


Abb. 3.3: Figuren mit Verläufen

Die Linie ganz unten wird durch die Anweisungen

```
stift.Brush = verlaufPinsel;
stift.Width = 10;
zeichnflaeche.DrawLine(stift, 10, 250, 270, 250);
```

am Ende des Codes erzeugt. Hier wird mit der Zeile

```
stift.Brush = verlaufPinsel;
```

die Eigenschaft `Brush` unseres Stiftes `stift` auf den Pinsel mit dem Verlauf gesetzt. Die Anweisung

```
zeichnflaeche.DrawLine(stift, 10, 250, 270, 250);
```

benutzt dann nicht mehr einen einfarbigen Pinsel für das Zeichnen der Linie, sondern den Pinsel mit dem Verlauf.

3.3 Texte in Grafiken

Schauen wir uns nun an, wie Sie Texte in einer Zeichenfläche ausgeben lassen.

Dazu verwenden Sie die Methode `DrawString()`¹⁷ der Klasse `Graphics`. Die Methode erwartet den auszugebenden Text, die Schriftart, den Pinsel und die Koordinaten für die Ausgabe als Argumente. Die Schriftart geben Sie dabei als Typ `Font` an. Der Pinsel bestimmt die Farbe der Schrift.

Die beiden folgenden Anweisungen erzeugen zum Beispiel zuerst eine Instanz der Klasse `Font` mit der Schrift Arial und der Schriftgröße 12. Danach wird dann mit der Methode `DrawString()` ein Text an der Position 20, 20 auf der Zeichenfläche ausgegeben.

```
//die Schriftart setzen
Font schrift = new Font("Arial",12);
//einfache Ausgabe
```

17. *Draw String* bedeutet frei übersetzt so viel wie „Zeichne Zeichenkette“.

```
//bitte in einer Zeile eingeben  
zeichenflaeche.DrawString("Ich stehe auf der  
Zeichenfläche", schrift, pinsel, new Point(20,20));
```

Code 3.5: Eine Textausgabe auf der Zeichenfläche

Wenn Sie als letztes Argument für die Methode `DrawString()` einen Bereich vom Typ `Rectangle` angeben, wird der Text nur in diesem Bereich dargestellt. Dabei erfolgt ein automatischer Umbruch. Die folgenden Anweisungen vereinbaren zunächst ein Rechteck und geben dann einen Text in diesem Rechteck aus.

```
Rectangle bereich;  
zeichenflaeche.ClearBackColor();  
//die Schriftart setzen  
Font schrift = new Font("Arial",12);  
//Ausgabe in einem Rechteck  
bereich = ClientRectangle;  
bereich.Width = 100;  
bereich.Height = 200;  
bereich.Location = new Point(100, 100);  
//bitte in einer Zeile eingeben  
zeichenflaeche.DrawString("Ich stehe in einem Rechteck",  
schrift, pinsel, bereich);
```

Code 3.6: Textausgabe in einem Rechteck

Abschließend noch ein wichtiger Hinweis:

Die Eigenschaften für den Stift und die Pinsel werden nach dem Zeichnen nicht wieder auf die Standardwerte zurückgesetzt. Achten Sie deshalb sorgfältig darauf, dass Sie die Eigenschaften vor dem Zeichnen korrekt setzen. Das ist vor allem dann wichtig, wenn Sie mehrere Objekte nacheinander zeichnen.

Auch beim Aufruf der Methode `Clear()` werden die Eigenschaften nicht wieder auf die Standardwerte zurückgesetzt. Es wird lediglich die Zeichenfläche gelöscht. Sie können aber die Standardwerte vor dem ersten Zeichnen speichern und später wieder abrufen. Dazu können Sie zum Beispiel über die Methoden `Clone()` der Klassen `Pen` und `Brush` Kopien erstellen und die Werte dieser Kopien dann bei Bedarf wieder zuweisen.

Das Erstellen der Kopien könnte zum Beispiel so erfolgen:

```
//Stift und Pinsel erzeugen  
stift = new Pen(Color.Black);  
pinsel = new SolidBrush(Color.Blue);  
//Kopien erstellen  
stiftKopie = (Pen)(stift.Clone());  
pinselKopie = (SolidBrush)(pinsel.Clone());
```

Code 3.7: Kopien von Stift und Pinsel erzeugen**Hinweis:**

Die Methode `Clone()` liefert den sehr allgemeinen Typ `object` zurück. Diese Rückgabe müssen Sie durch ein Casting in den passenden Typ umwandeln.

Um die ursprünglichen Einstellungen für den Stift oder Pinsel wiederherzustellen, weisen Sie `stift` beziehungsweise `pinsel` die Werte von `stiftKopie` beziehungsweise `pinselKopie` zu. Dabei gibt es keine Besonderheiten.

So viel zu den Gestaltungsmöglichkeiten. Im nächsten Kapitel werden wir eine kleine Anwendung programmieren, in der Sie mit Ihrem neuen Wissen spielen können.

Zusammenfassung

Über die Klassen `Pen` und `Brush` können Sie unter anderem Farben und Füllmuster setzen.

Neben einem Standardpinsel für einfache Füllungen gibt es auch Pinsel für Füllmuster und Verläufe.

Mit der Methode `DrawString()` geben Sie Texte direkt in einer Zeichenfläche aus.

Aufgaben zur Selbstüberprüfung

- 3.1 Welche Klasse steuert die Gestaltung von Linien? Welche Klassen steuern die Gestaltung der Füllungen?

- 3.2 Sie wollen eine gestrichelte Linie mit der Breite 2 zeichnen. Wie lauten die Anweisungen, um diese Eigenschaften zu setzen?

- 3.3 Sie wollen eine Grafik als Füllmuster verwenden. Welche Schritte sind dazu erforderlich?

- 3.4 Wie können Sie sehr schnell Kopien eines Stifts und eines Pinsels erstellen?

- 3.5 Wodurch wird die Farbe von Texten bestimmt, die direkt auf der Zeichenfläche ausgegeben werden?

4 Eine kleine Spielerei

In diesem Kapitel werden wir ein Programm erstellen, in dem der Anwender über verschiedene Bedienelemente Einstellungen für unterschiedliche grafische Figuren vornehmen kann.

Wir werden diese Anwendung wie gewohnt Schritt für Schritt ausbauen. Die endgültige Version soll ungefähr so aussehen:

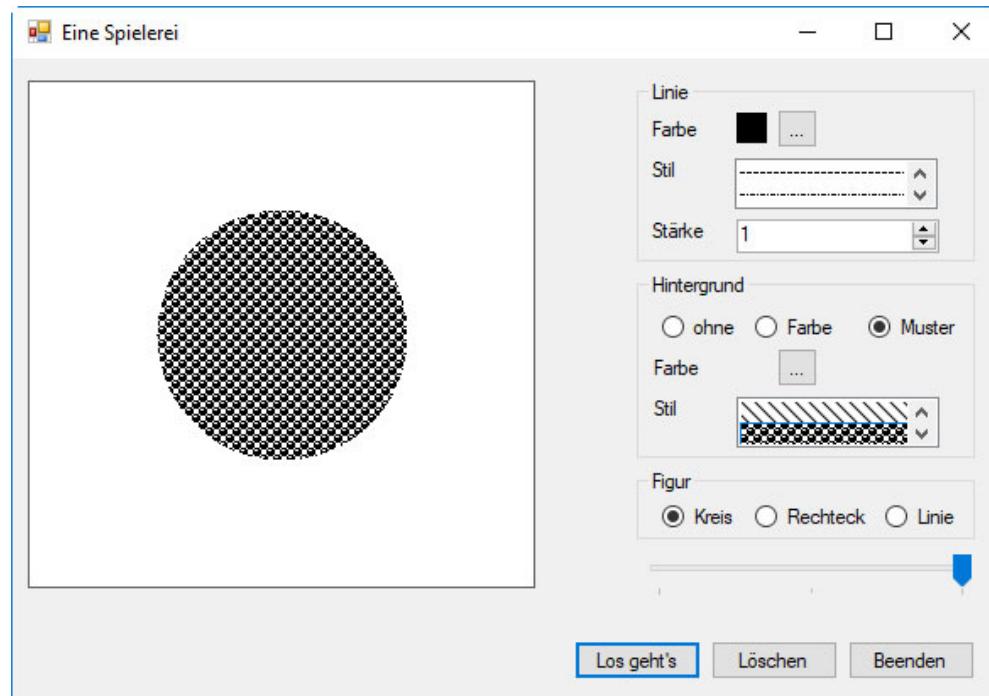


Abb. 4.1: Eine Spielerei

Über die Felder im Bereich **Linie** soll der Anwender die Farbe, den Stil und die Breite der Linie einstellen können. Die entsprechenden Einstellungen für den Hintergrund erfolgen im Bereich **Hintergrund**.

Im unteren Bereich des Formulars können die Form und auch die Größe der Figur ausgewählt werden.

Bevor wir uns an die Umsetzung machen, zunächst noch ein paar Vorüberlegungen.

4.1 Einige Vorüberlegungen

Die Anzeige der Figuren nehmen wir in einem eigenen Bereich des Formulars vor. Dazu verwenden wir ein Steuerelement **Panel**. Damit das Ganze etwas netter aussieht, stellen wir das Panel als Quadrat dar und versehen es mit einem weißen Hintergrund. Auf diesem weißen Quadrat sollen dann die Figuren erscheinen.

Für die Einstellungen der Figuren benötigen wir zunächst einmal folgende Steuerelemente:

- zwei Kombinationsfelder für die Farben,
- zwei Listenfelder für den Stil,
- ein Drehfeld für die Breite des Stiftes,
- Optionsfelder für die Auswahl der Füllung,
- Optionsfelder für die Figur und
- einen Schieberegler für die Größe.

Die Steuerelemente für die Linie und den Hintergrund setzen wir in einzelne Gruppen, um sie optisch voneinander zu trennen.

Außerdem setzen wir noch drei Schaltflächen ein. Mit der ersten Schaltfläche starten wir das Zeichnen der Figur, die zweite soll die Zeichnung löschen und die dritte schließlich das Programm beenden.

4.2 Das Formular

Erstellen wir jetzt das Formular für das Programm. Legen Sie bitte eine neue leere Windows Forms-Anwendung an.

Setzen Sie die Breite des Formulars dann auf 600 und die Höhe auf 420. Als Titel können Sie zum Beispiel **Eine Spielerei** benutzen.

Fügen Sie anschließend das Panel für die Anzeige der Figuren oben links in der Ecke ein. Sie finden das entsprechende Steuerelement in der Gruppe **Container** der Toolbox. Setzen Sie die Breite und Höhe des Panels bitte jeweils auf 300 Punkte. Wählen Sie anschließend über die Eigenschaft **BackColor** die Farbe Weiß aus. Dazu müssen Sie unter Umständen erst bei der Auswahl der Farben in das Register **Web** oder **Benutzerdefiniert** wechseln. Wenn Sie wollen, können Sie außerdem über die Eigenschaft **BorderStyle** noch einen Rahmen um das Panel zeichnen lassen.

Fügen Sie dann rechts oben im Formular untereinander zwei GroupBoxen ein. Sie finden dieses Steuerelement ebenfalls in der Gruppe **Container** der Toolbox. Setzen Sie die Eigenschaft **Text** der ersten Gruppe auf **Linie** und die Eigenschaft **Text** der zweiten Gruppe auf **Hintergrund**.

Zur Auffrischung:

Wenn Sie ein Steuerelement mehrfach einfügen wollen, halten Sie beim Ablegen des Steuerelements die Taste **Strg** gedrückt. Bei jedem Mausklick in das Formular wird dann ein weiteres Steuerelement vom ausgewählten Typ eingefügt. Um die Funktion wieder abzuschalten, drücken Sie die Taste **Esc**.



Setzen Sie in die Gruppe für die Linien ein Kombinationsfeld, ein Listenfeld und ein Drehfeld. Das Drehfeld finden Sie in der Gruppe **Allgemeine Steuerelemente** der Toolbox unter dem Eintrag **NumericUpDown**¹⁸.

Verkleinern Sie im nächsten Schritt das Listenfeld so, dass nur zwei Zeilen angezeigt werden. Setzen Sie anschließend vor das Kombinationsfeld ein Label mit dem Text **Farbe**, vor das Listenfeld ein Label mit dem Text **Stil** und vor das Drehfeld ein Label mit dem Text **Stärke**.

Fügen Sie dann in die Gruppe für den Hintergrund oben drei Optionsfelder ein. Das erste Optionsfeld bekommt den Text **ohne**, das zweite den Text **Farbe** und das dritte den Text **Muster**.

Setzen Sie außerdem in die Gruppe für den Hintergrund ein Kombinationsfeld für die Farbe und ein zweizeiliges Listenfeld für das Muster. Ergänzen Sie auch für diese Steuerelemente beschreibende Texte über entsprechende Labels.

Hinweis:

Wir werden die Farbauswahl zunächst einmal über die Kombinationsfelder vornehmen. Die Auswahl über die Schaltfläche und die Anzeige setzen wir später um.

Fügen Sie anschließend unterhalb der beiden Gruppen eine weitere GroupBox mit drei Optionsfeldern ein. Setzen Sie den Text für die GroupBox auf **Figur** und die Texte für die Optionsfelder auf **Kreis**, **Rechteck** und **Linie**.

Unterhalb der neuen Gruppe fügen Sie dann bitte noch den Schieberegler ein. Sie finden ihn in der Gruppe **Alle Windows Forms** der Toolbox unter dem Eintrag **TrackBar**¹⁹.

Setzen Sie dann noch die drei Schaltflächen in das Formular und versehen Sie sie mit entsprechenden Beschriftungen.

18. Wörtlich übersetzt bedeutet **NumericUpDown** „Numerisches Auf und Ab“.

19. Wörtlich übersetzt bedeutet **TrackBar** „Spurbalken“.

Im letzten Schritt ordnen Sie die ganzen Steuerelemente noch passend unter- und neben- einander an. Das Formular könnte nun ungefähr so aussehen:

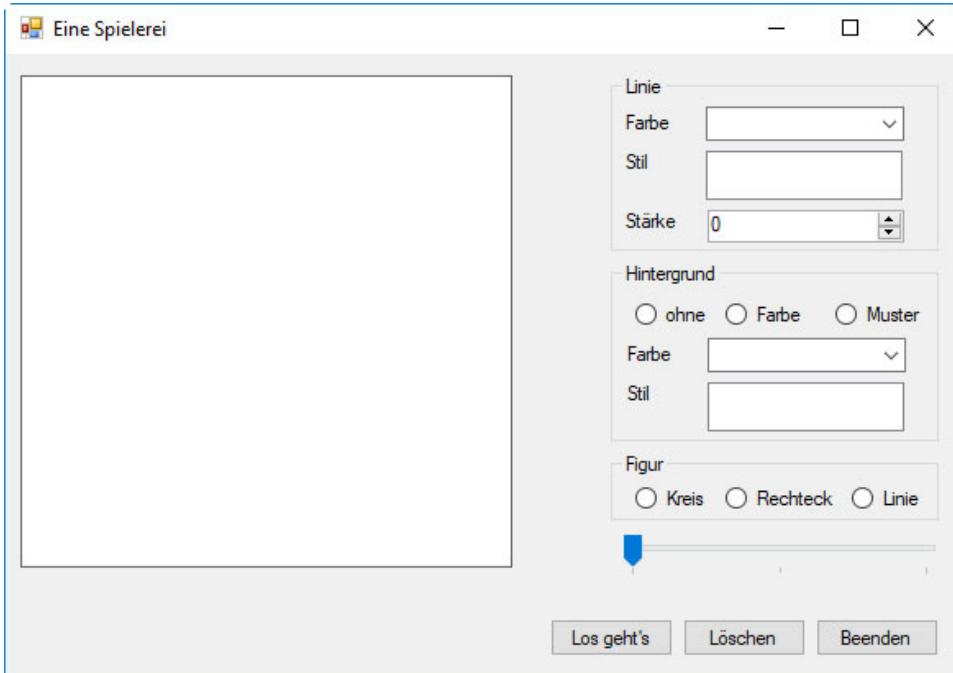


Abb. 4.2: Das Formular mit den Steuerelementen

Nach der Gestaltung können wir uns jetzt um die Eigenschaften für die Steuerelemente kümmern.

Vergeben Sie bitte im ersten Schritt zumindest für die Steuerelemente, die mehrfach vorkommen, eindeutige Namen. Wir verwenden in unserem Projekt die folgenden Bezeichner:

Tab. 4.1: Die Bezeichner der Steuerelemente im Projekt **Spielerei**

Steuerelement	(Name)
Kombinationsfeld im Bereich Linie	comboBoxLinieFarbe
Listenfeld im Bereich Linie	listBoxLinieStil
Drehfeld im Bereich Linie	numericUpDownLinieStaerke
Optionsfeld ohne im Bereich Hintergrund	radioButtonHintergrundOhne
Optionsfeld Farbe im Bereich Hintergrund	radioButtonHintergrundFarbe
Optionsfeld Muster im Bereich Hintergrund	radioButtonHintergrundMuster
Kombinationsfeld im Bereich Hintergrund	comboBoxHintergrundFarbe
Listenfeld im Bereich Hintergrund	listBoxHintergrundMuster
Optionsfeld Kreis im Bereich Figur	radioButtonKreis
Optionsfeld Rechteck im Bereich Figur	radioButtonRechteck

Steuerelement	(Name)
Optionsfeld Linie im Bereich Figur	radioButtonLinie
Schaltfläche Los geht's	buttonStart
Schaltfläche Löschen	buttonLoeschen
Schaltfläche Beenden	buttonBeenden

Legen Sie dann für die beiden Kombinationsfelder für die Farben über die Eigenschaft **Items** einige Einträge an – zum Beispiel Schwarz, Rot, Blau und Grün.

Hinweis:

Wir werden in einem weiteren Schritt die Farbauswahl noch etwas ausführlicher gestalten. Jetzt belassen wir es erst einmal bei einigen wenigen fest vorgegebenen Farben. Auch um die Auswahl der Linienstile und der Muster kümmern wir uns später. Die Listenfelder bleiben daher zunächst einmal komplett leer.

Sorgen Sie anschließend dafür, dass über das Drehfeld für die Liniенstärke Werte zwischen 1 und 20 ausgewählt werden können. Dazu setzen Sie die Eigenschaft **Minimum** auf 1 und die Eigenschaft **Maximum** auf 20. Sie finden diese beiden Eigenschaften in der Gruppe **Daten** des Eigenschaftenfensters.

Hinweis:

Durch das Verändern der Eigenschaft **Minimum** wird auch die Eigenschaft **Value** automatisch auf 1 gesetzt. Sie steht für den aktuellen Wert des Drehfelds.

Setzen Sie dann die Eigenschaft **Checked** für das Optionsfeld **ohne** im Bereich **Hintergrund** und für das Optionsfeld **Kreis** im Bereich **Figur** jeweils auf **True**. Dadurch werden die beiden Optionen direkt nach dem Start der Anwendung markiert.

Passen Sie nun noch den Schieberegler so an, dass drei unterschiedliche Werte eingestellt werden können. Dazu setzen Sie die Eigenschaft **Minimum** auf 1 und die Eigenschaft **Maximum** auf 3. Bei einem Schieberegler finden Sie diese beiden Eigenschaften in der Gruppe **Verhalten** des Eigenschaftenfensters.

Damit der Regler zunächst in der Mitte steht, ändern Sie außerdem den Wert der Eigenschaft **Value** auf 2. Diese Eigenschaft steht für den aktuellen Wert des Schiebereglers.

Nach diesen Änderungen sollte das Formular nun ungefähr so aussehen:

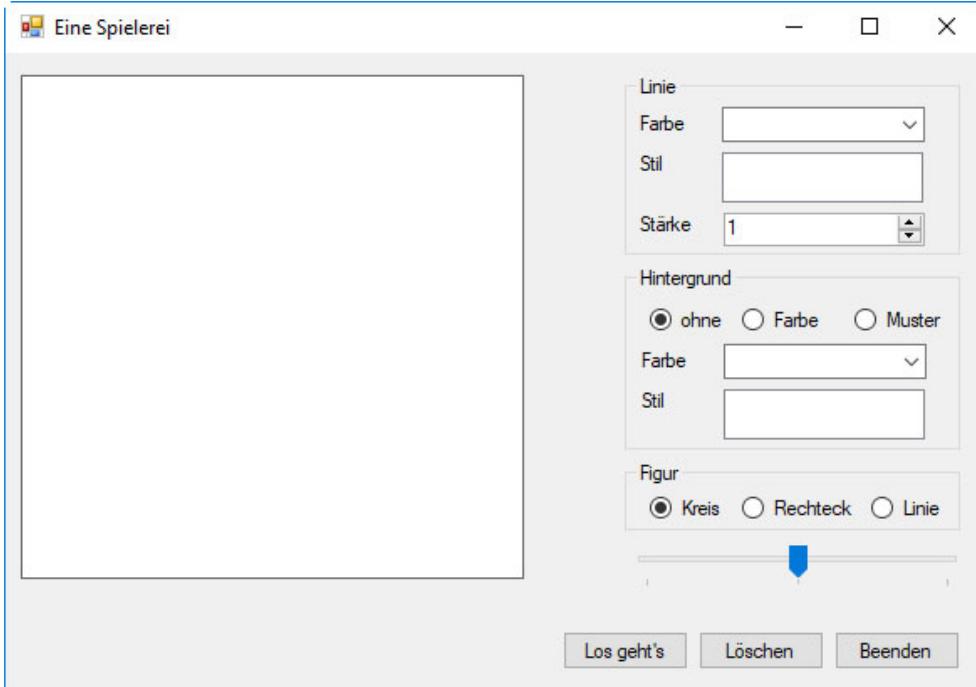


Abb. 4.3: Das Formular nach dem Setzen der Eigenschaften

4.3 Das Zeichnen der Figuren

Nun können wir uns um das Zeichnen der Figuren kümmern.

Im ersten Schritt sorgen wir erst einmal dafür, dass in den beiden Kombinationsfeldern jeweils der erste Eintrag markiert ist. Dazu setzen wir im Ereignis **Load** des Formulars die Eigenschaft `SelectedIndex` der beiden Kombinationsfelder auf den Wert 0.

Außerdem beschaffen wir uns in dem Ereignis die Zeichenfläche für das Panel und setzen eine Referenz auf diese Zeichenfläche.

Die vollständige Methode für das Ereignis **Load** sieht so aus:

```
private void Form1_Load(object sender, EventArgs e)
{
    //den ersten Eintrag in den Kombinationsfeldern
    //auswählen
    comboBoxHintergrundFarbe.SelectedIndex = 0;
    comboBoxLinieFarbe.SelectedIndex = 0;
    //eine Referenz auf die Zeichenfläche des Panels
    //beschaffen
    //zeichenflaeche ist als Feld der Klasse Form1
    //vereinbart
    zeichenflaeche = panel1.CreateGraphics();
}
```

Code 4.1: Die Methode `Form1_Load()`



Bitte beachten Sie:

Da wir in unterschiedlichen Methoden auf die Zeichenfläche zugreifen werden, haben wir `zeichenflaeche` der Einfachheit halber als Feld der Klasse `Form1` vereinbart. Als Typ müssen Sie dabei `Graphics` verwenden.

Sie können die Zeichenfläche aber auch in den folgenden Methoden jeweils neu setzen. Dann können Sie auch mit einer lokalen Variablen arbeiten.

Beim Anklicken der Schaltfläche **Löschen** löschen wir die Zeichenfläche über die Methode `Clear()`. Denken Sie dabei bitte daran, dass Sie als Hintergrundfarbe die Hintergrundfarbe des Panels verwenden müssen – und nicht die Hintergrundfarbe des Formulars. Die entsprechende Anweisung muss also so aussehen:

```
zeichenflaeche.Clear(panel1.BackColor);
```

Beim Anklicken der Schaltfläche **Beenden** schließen wir die gesamte Anwendung über die Methode `Close()`.

Spannend wird es beim Anklicken der Schaltfläche **Los geht's**. Hier übernehmen wir die verschiedenen Einstellungen aus den Feldern und lassen dann – abhängig vom ausgewählten Eintrag in der Gruppe für die Figur – entweder einen Kreis, ein Rechteck oder eine Linie in dem Panel links oben im Formular erstellen. Je nach ausgewählter Größe verschieben wir dabei die Startpunkte nach rechts und unten beziehungsweise nach links und oben.



Bitte beachten Sie:

Um die Einstellungen für den Linienstil des Stifts und für die Füllmuster kümmern wir uns hier noch nicht. Die entsprechenden Auswahlen über die Listenfelder programmieren wir im letzten Schritt.

Die Anweisungen für das Ereignis **Click** der Schaltfläche **Los geht's** finden Sie im Code 4.2. Erklärungen erhalten Sie wie gewohnt im Anschluss.

```
//eine lokale Variable für die Größe
int groesse = 0;
//einen schwarzen Stift erzeugen
Pen stift = new Pen(Color.Black);
//einen schwarzen Pinsel erzeugen
SolidBrush pinsel = new SolidBrush(Color.Black);
//je nach Wert in den Kombinationsfeldern die Farben
//setzen
switch (comboBoxLinieFarbe.SelectedIndex)
{
    case 0:
        stift.Color = Color.Black;
        break;
    case 1:
        stift.Color = Color.Red;
        break;
```

```
        case 2:
            stift.Color = Color.Blue;
            break;
        case 3:
            stift.Color = Color.Green;
            break;
    }
    switch (comboBoxHintergrundFarbe.SelectedIndex)
    {
        case 0:
            pinsel.Color = Color.Black;
            break;
        case 1:
            pinsel.Color = Color.Red;
            break;
        case 2:
            pinsel.Color = Color.Blue;
            break;
        case 3:
            pinsel.Color = Color.Green;
            break;
    }
    //die Dicke des Stiftes setzen
    //bitte in einer Zeile eingeben
    stift.Width = Convert.ToInt32(numericUpDownLinieStaerke.
Value);
    //die Größe der Figur ermitteln
    switch (trackBar1.Value)
    {
        case 1:
            groesse = 125;
            break;
        case 2:
            groesse = 100;
            break;
        case 3:
            groesse = 75;
            break;
    }
    //Figur ermitteln
    //beim Kreis und beim Rechteck auch die Füllung
    //überprüfen
    if (radioButtonKreis.Checked == true)
    {
        if (radioButtonHintergrundOhne.Checked == true)
            //bitte in einer Zeile eingeben
            zeichenflaeche.DrawEllipse(stift,
            panel1.ClientRectangle.Left + groesse,
            panel1.ClientRectangle.Top + groesse,
            panel1.ClientRectangle.Width - (groesse*2),
            panel1.ClientRectangle.Height - (groesse * 2));
        if (radioButtonHintergrundFarbe.Checked == true)
            //bitte in einer Zeile eingeben
            zeichenflaeche.FillEllipse(pinsel,
            panel1.ClientRectangle.Left + groesse,
            panel1.ClientRectangle.Top + groesse,
```

```

        panel1.ClientRectangle.Width - (groesse*2),
        panel1.ClientRectangle.Height - (groesse * 2));
    }
    if (radioButtonRechteck.Checked == true)
    {
        if (radioButtonHintergrundOhne.Checked == true)
            //bitte in einer Zeile eingeben
            zeichenflaeche.DrawRectangle(stift,
                panel1.ClientRectangle.Left + groesse,
                panel1.ClientRectangle.Top + groesse,
                panel1.ClientRectangle.Width - (groesse*2),
                panel1.ClientRectangle.Height - (groesse * 2));
        if (radioButtonHintergrundFarbe.Checked == true)
            //bitte in einer Zeile eingeben
            zeichenflaeche.FillRectangle(pinsel,
                panel1.ClientRectangle.Left + groesse,
                panel1.ClientRectangle.Top + groesse,
                panel1.ClientRectangle.Width - (groesse*2),
                panel1.ClientRectangle.Height - (groesse * 2));
    }
    if (radioButtonLinie.Checked == true)
        //bitte in einer Zeile eingeben
        zeichenflaeche.DrawLine(stift,
            panel1.ClientRectangle.Left + groesse,
            panel1.ClientRectangle.Height / 2,
            panel1.ClientRectangle.Width - groesse,
            panel1.ClientRectangle.Height / 2);
}

```

Code 4.2: Die Anweisungen zum Zeichnen

Mit der ersten Anweisung vereinbaren wir eine Variable `groesse`, in der wir später den Abstand der Figuren von den Rändern ablegen.

Danach erzeugen wir mit den Anweisungen

```

Pen stift = new Pen(Color.Black);
SolidBrush pinsel = new SolidBrush(Color.Black);

```

einen Stift und einen Pinsel für eine einfache Füllung. In beiden Fällen setzen wir die Farbe zunächst auf Schwarz.

Anschließend ermitteln wir in den `switch`-Konstruktionen, welche Farbe aktuell in den Kombinationsfeldern ausgewählt ist, und setzen die Eigenschaft `Color` für den Stift beziehungsweise den Pinsel auf den entsprechenden Wert.

Hinweis:

Wenn Sie andere Farben in Ihren Kombinationsfeldern benutzen, müssen Sie diese Anweisungen entsprechend anpassen.

Danach setzen wir die Breite des Stifts auf den aktuellen Wert des Drehfelds. Diesen Wert erhalten wir über die Eigenschaft `value`. Denken Sie bitte daran, dass Sie den Wert der Eigenschaft erst noch konvertieren müssen.

Anschließend legen wir in einer `switch`-Konstruktion den Wert für die Variable `groesse` fest. Ausgewertet wird dabei der aktuelle Wert des Schiebereglers, den wir über die Eigenschaft `value` erhalten.

Je kleiner der eingestellte Wert ist, desto größer wird der Abstand der Figuren vom Rand. Und je größer der Abstand vom Rand, desto kleiner wird die Figur selbst.



In den `if`-Anweisungen erfolgt dann das eigentliche Zeichnen der Figuren. Dazu werten wir die Optionsfelder für die Figur und den Hintergrund aus und lassen anschließend die entsprechenden Objekte zeichnen. Die Koordinaten des Kreises und des Rechtecks werden dabei aus den Eigenschaften `ClientRectangle.Left` und `ClientRectangle.Top` des Panels berechnet. Die Breite und Höhe ermitteln wir aus den Eigenschaften `ClientRectangle.Width` beziehungsweise `ClientRectangle.Height` des Panels. Um den Abstand zum linken und rechten Rand zu berücksichtigen, müssen wir von diesen Werten den **doppelten** Wert von `groesse` abziehen.

Hinweis:

Denken Sie bitte daran, dass Sie bei den allermeisten grafischen Objekten nicht direkt die rechte untere Koordinate angeben, sondern die Breite und die Höhe. Deshalb müssen wir auch den doppelten Wert von `groesse` abziehen. Wenn Sie nur den einfachen Wert abziehen, wird das Objekt zu nah an den rechten Rand gezeichnet.

Bei der Linie zeichnen wir in der Mitte des Panels vom linken Rand bis zum rechten Rand. Die Mitte des Panels berechnen wir dabei, indem wir die Höhe durch 2 teilen.

Übernehmen Sie jetzt die Anweisungen aus dem Code. Speichern Sie dann die Änderungen und lassen Sie das Programm ausführen. Alle Einstellungen bis auf den Linien- und Füllstil sollten funktionieren.

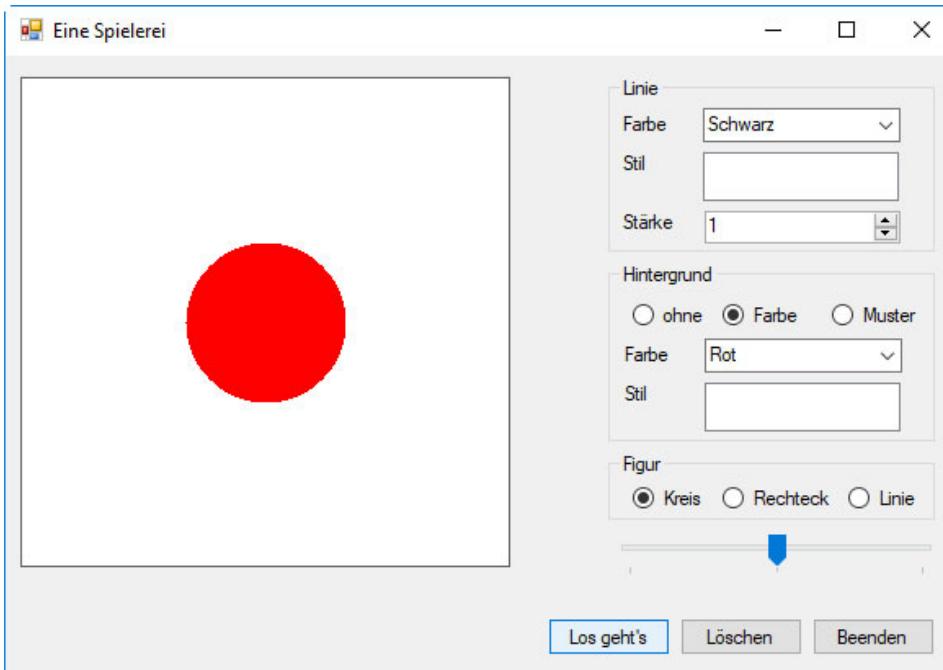


Abb. 4.4: Das Programm in Aktion

4.4 Die Farbauswahl

Das Zeichnen funktioniert jetzt zwar schon ganz gut, die Auswahl der Farben ist aber alles andere als elegant. Sie könnten zwar die Listen in den Kombinationsfeldern beliebig erweitern, allerdings wird dann auch die Überprüfung und Zuordnung der Farben im Quelltext immer aufwendiger und auch anfälliger.

Sehr viel komfortabler wäre es, den Anwender die Farbe über einen Dialog auswählen zu lassen und dann auch noch eine kurze Vorschau im Formular anzuzeigen. Das lässt sich über das Steuerelement **ColorDialog** und ein weiteres Panel bewerkstelligen.

Schauen wir uns das Vorgehen an der Farbeinstellung für die Linie an. Löschen Sie im ersten Schritt bitte das Kombinationsfeld `comboBoxLinienFarbe` aus dem Formular. Fügen Sie stattdessen rechts neben dem Label ein kleines Panel und eine kleine Schaltfläche mit drei Punkten ein. Als Namen für das Panel verwenden wir `panelLinieFarbeVorschau` und als Namen für die Schaltfläche `buttonLinieFarbe`.

Da wir den Stift im Programm beim Erzeugen direkt auf die Farbe Schwarz setzen, können Sie den Hintergrund dieses Panels ebenfalls auf Schwarz setzen.

Nehmen Sie dann ein Steuerelement **ColorDialog** in das Formular auf. Sie finden es in der Gruppe **Dialogfelder** der Toolbox.

Beim Anklicken der Schaltfläche für die Farbauswahl lassen Sie dann den Dialog zur Farbauswahl modal anzeigen. Da der Dialog nicht – wie zum Beispiel der Standarddialog **Öffnen** – über ein Ereignis **FileOK** verfügt, müssen Sie selbst überprüfen, ob der Dialog über die **OK**-Schaltfläche verlassen wurde. Dazu vergleichen Sie den Rückgabewert der Methode `ShowDialog()` mit dem Wert `DialogResult.OK`.

Wenn der Anwender im Dialog eine Farbe ausgewählt hat, setzen wir zunächst einmal die Hintergrundfarbe für das kleine Panel zur Vorschau neu. Damit wir auch beim Zeichnen der Figuren auf die Farbe zugreifen können, vereinbaren wir zusätzlich ein Feld `linienfarbe` vom Typ `Color` in der Klasse `Form1`. Diesem Feld weisen wir dann die ausgewählte Farbe zu.

Die vollständige Methode für die Schaltfläche zur Farbauswahl der Linie sieht dann so aus:

```
private void ButtonLinienFarbe_Click(object sender,
EventArgs e)
{
    //den Dialog zur Farbauswahl anzeigen
    if (colorDialog1.ShowDialog() == DialogResult.OK)
    {
        //die Hintergrundfarbe für das Panel auf die
        //ausgewählte Farbe setzen
        panelLinienFarbeVorschau.BackColor = colorDialog1.Color;
        //und die Linienfarbe
        //linienfarbe ist ein Feld der Klasse Form1
        linienfarbe = colorDialog1.Color;
    }
}
```

Code 4.3: Die Methode zur Farbauswahl für die Linie

Hinweis:

Wie die Methode genau heißt, hängt vom Namen der Schaltfläche ab. Auch der Name des Panels kann abweichen.

Im nächsten Schritt müssen wir nun das Setzen der Farbe umbauen. Löschen Sie dazu bitte zunächst die beiden Anweisungen für die Auswahl des ersten Eintrags in den Kombinationsfeldern aus der Methode `Form1_Load()`. Ergänzen Sie dann in dieser Methode eine Anweisung, die das Feld `linienFarbe` auf Schwarz setzt. Damit ist sichergestellt, dass der Stift keine undefinierte Farbe hat.

In der Methode `ButtonStart_Click()` ändern Sie die Anweisung zum Erzeugen des Stifts so, dass nicht mehr fest die Farbe Schwarz benutzt wird, sondern der Wert aus dem Feld `linienFarbe`. Anschließend löschen Sie die gesamte `switch`-Konstruktion zum Auswerten des ersten Kombinationsfelds.

Führen Sie diese Änderungen jetzt bitte durch. Denken Sie dabei auch daran, das Feld `linienFarbe` in der Klasse `Form1` zu vereinbaren. Testen Sie dann die neue Farbauswahl.

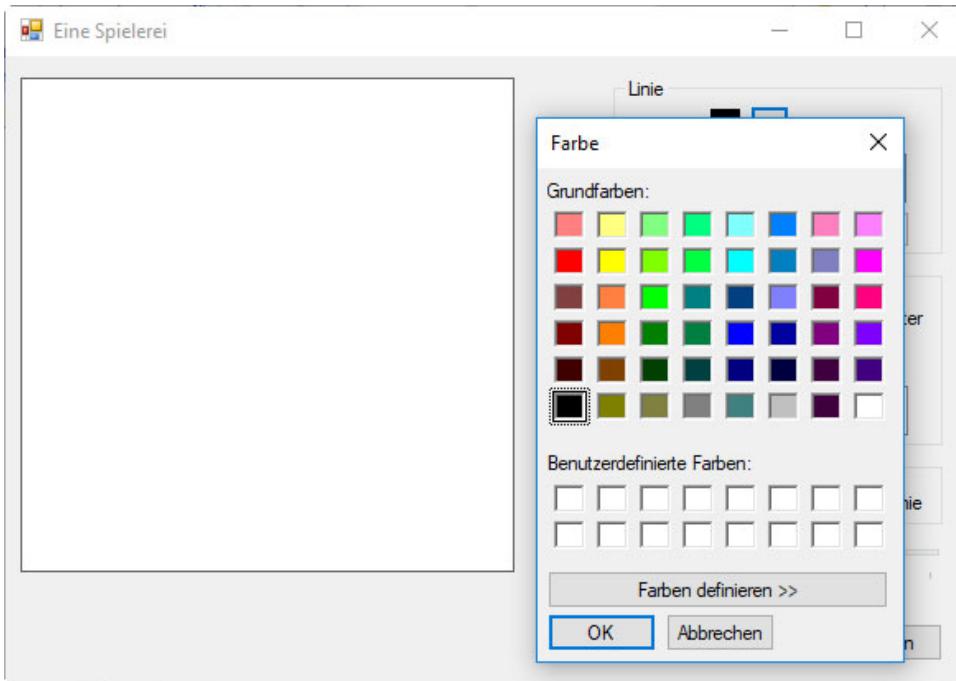


Abb. 4.5: Der Dialog zur Farbauswahl

Die entsprechenden Änderungen für die Farbauswahl des Hintergrunds erfolgen mit einer sehr ähnlichen Technik.

Im ersten Schritt vereinbaren Sie ein neues Feld `hintergrundfarbe` vom Typ `Color` in der Klasse `Form1`. Dieses Feld setzen Sie im Ereignis `Load` für das Formular ebenfalls auf Schwarz.

Danach ersetzen Sie das Kombinationsfeld für die Farbauswahl durch ein Panel und eine Schaltfläche. Beim Anklicken der Schaltfläche lassen Sie dann wieder den Dialog zur Farbauswahl modal anzeigen, ändern die Hintergrundfarbe des Panels für die Vorschau und setzen das Feld `hintergrundfarbe` auf die ausgewählte Farbe.

Die entsprechende Methode sieht so aus:

```
private void ButtonHintergrundFarbe_Click(object sender,
EventArgs e)
{
    //den Dialog zur Farbauswahl anzeigen
    if (colorDialog1.ShowDialog() == DialogResult.OK)
    {
        //die Hintergrundfarbe für das Panel auf die
        //ausgewählte Farbe setzen
        //bitte in einer Zeile eingeben
        panelHintergrundFarbeVorschau.BackColor =
            colorDialog1.Color;
        //und die eigentliche Hintergrundfarbe
        //hintergrundfarbe ist ein Feld der Klasse Form1
        hintergrundfarbe = colorDialog1.Color;
        //die Auswahl Farbe aktivieren
        radioButtonHintergrundFarbe.Checked = true;
    }
}
```

Code 4.4: Die Methode zur Farbauswahl für den Hintergrund

Hinweis:

Auch in diesem Code hängen der Name der Methode und des Panels von Ihren Einstellungen ab.

Wie Sie selbst sehen, bestehen keine großen Unterschiede zur Farbauswahl für die Linie. Wir ändern lediglich die Vorschau in dem zweiten Panel und setzen am Ende das Feld `hintergrundfarbe`. Außerdem aktivieren wir zusätzlich noch das Optionsfeld **Farbe** im Bereich **Hintergrund**. Damit ist sichergestellt, dass die neue Farbe auch tatsächlich beim nächsten Zeichnen benutzt wird.

In der Methode `ButtonStart_Click()` ersetzen Sie dann die Farbangabe `Color.Black` beim Erzeugen des Pinsels durch das Feld `hintergrundfarbe`. Außerdem löschen Sie die gesamte `switch`-Konstruktion mit der Auswertung des Kombinationsfelds `comboBoxHintergrundFarbe`.

Die ersten Zeilen der Methode sollten nun so aussehen:

```
//eine lokale Variable für die Größe
int groesse = 0;
//einen Stift in der aktuellen Farbe erzeugen
Pen stift = new Pen(linienfarbe);
//einen Pinsel in der aktuellen Farbe erzeugen
SolidBrush pinsel = new SolidBrush(hintergrundfarbe);
...
```

Code 4.5: Die geänderten Anweisungen in der Methode `ButtonStart_Click()`

Übernehmen Sie auch diese Änderungen und testen Sie das Programm noch einmal. Nun sollte sich auch die Hintergrundfarbe über den Dialog festlegen lassen.

Im letzten Schritt programmieren wir jetzt die Listen für den Linienstil und das Hintergrundmuster.

4.5 Die Listen für den Linienstil und das Hintergrundmuster

Dazu zunächst noch einmal zur Auffrischung einige Hinweise, wie ein Listenfeld überhaupt aufgebaut ist.

Die einzelnen Einträge in der Liste werden über die Eigenschaft `Items` abgebildet. Mit der Methode `Add()` können Sie selbst zur Laufzeit neue Einträge am Ende der Liste anfügen. Der Text des Eintrags wird dabei als Typ `string` übergeben.

Die Anweisung

```
listBox1.Items.Add("Ich bin neu");
```

würde zum Beispiel am Ende der Liste von `listBox1` einen neuen Eintrag mit dem Text „Ich bin neu“ anfügen.

Den aktuell ausgewählten Eintrag in einem Listenfeld erhalten Sie über die Eigenschaft `SelectedIndex`. Der Wert 0 steht dabei für das erste Element, der Wert 1 für das zweite und so weiter. Wenn `SelectedIndex` den Wert -1 liefert, ist kein Element ausgewählt worden.

Über `SelectedIndex` können Sie auch selbst einen Eintrag in einem Listenfeld auswählen. Dazu weisen Sie der Eigenschaft einfach einen gültigen Wert zu.

Eine recht einfache Möglichkeit, die Listenfelder für die Stile zu erstellen, wäre es nun, die Werte wie ursprünglich bei den Farben fest einzutragen, dann in einer `if`-Konstruktion auszuwerten und anschließend die jeweiligen Eigenschaften zu setzen. Das ist aber nicht nur sehr mühselig, sondern auch alles andere als elegant.

Wir gehen daher einen anderen Weg und machen uns dabei zunutze, dass sowohl die verschiedenen Linienstile als auch die verschiedenen Hintergrundmuster als Aufzählungstyp vorliegen. Wir können sie also recht einfach in einer Schleife verarbeiten.

Schauen wir uns das jetzt am Beispiel des Linienstils an. Im ersten Schritt vereinbaren wir in der Klasse `Form1` ein Array vom Typ `System.Drawing`.

`Drawing2D.DashStyle`. In diesem Array führen wir dann verschiedene Linienstile auf. Das könnte so aussehen:

```
System.Drawing.Drawing2D.DashStyle[] linienstil =  
{  
    System.Drawing.Drawing2D.DashStyle.Dash,  
    System.Drawing.Drawing2D.DashStyle.DashDot,  
    System.Drawing.Drawing2D.DashStyle.DashDotDot,  
    System.Drawing.Drawing2D.DashStyle.Dot,  
    System.Drawing.Drawing2D.DashStyle.Solid  
};
```

Hinweis:

Die möglichen Werte für `System.Drawing.Drawing2D.DashStyle` können Sie ganz einfach aus dem Hilfedokument für die `DashStyle`-Enumeration übernehmen. Im Bereich **Felder** finden Sie neben den Werten auch eine kurze Beschreibung.

Custom	5	Gibt eine benutzerdefinierte Strichart an.
Dash	1	Gibt eine Linie an, die aus Strichen besteht.
DashDot	3	Gibt eine Linie an, die aus einer sich wiederholenden Strich-Punkt-Folge besteht.
DashDotDot	4	Gibt eine Linie an, die aus einer sich wiederholenden Strich-Punkt-Punkt-Folge besteht.
Dot	2	Gibt eine Linie an, die aus Punkten besteht.
Solid	0	Gibt eine durchgehende Linie an.

Beispiele

Im folgenden Codebeispiel wird veranschaulicht, wie einen Stift erstellt, und legen Sie dessen `DashStyle` Eigenschaft mit dem `DashStyle` Enumeration.

In diesem Beispiel wird mit Windows Forms verwendet werden soll. Erstellen Sie ein Formular, enthält eine `Button` mit dem Namen `Button3`. Fügen Sie den Code in das Formular, und ordnen die `Button3_Click` -Methode mit der Schaltfläche `Click` Ereignis.

Abb. 4.6: Das Hilfedokument zur `DashStyle`-Enumeration

Nachdem wir jetzt den Inhalt der Liste festgelegt haben, müssen wir noch dafür sorgen, dass die Liste auch im Listenfeld erscheint. Dazu können Sie zum Beispiel im Ereignis **Load** des Formulars eine Schleife erstellen, die alle Werte aus der Liste über die Methode `Add()` in das Listenfeld einfügt. Die entsprechenden Anweisungen könnten so aussehen:

```
//bitte in einer Zeile eingeben
foreach (System.Drawing.Drawing2D.DashStyle element in
linienstil)
    listBoxLinieStil.Items.Add(element);
```

Code 4.6: Das Füllen des Listenfeldes für die Linienstile

In der `foreach`-Schleife werden alle Elemente aus dem Array `linienstil` nacheinander in das Listenfeld eingetragen.

Bei der Zuweisung der Stifteigenschaften können Sie nun auch den Linienstil berücksichtigen. Dazu ergänzen Sie im Ereignis **Click** für die Schaltfläche **Los geht's** die folgenden Anweisungen:

```
if (listBoxLinieStil.SelectedIndex >= 0)
    //bitte in einer Zeile eingeben
    stift.DashStyle =
        linienstil[listBoxLinieStil.SelectedIndex] ;
```

Code 4.7: Das Setzen des Linienstils

Zunächst überprüfen wir über `SelectedIndex`, ob überhaupt ein Wert ausgewählt wurde. Wenn das der Fall ist, wird der entsprechende Wert aus dem Array `linienstil` der Eigenschaft `DashStyle` des Stifts zugewiesen. Als Index benutzen wir dabei den Wert von `SelectedIndex` aus dem Listenfeld.

Hinweis:

Wo Sie die beiden Anweisungen im Ereignis **Click** einfügen, ist relativ beliebig. Sie müssen lediglich vor den Anweisungen zum Zeichnen stehen.

Speichern Sie jetzt die Änderungen und testen Sie dann die Einstellungen für den Linienstil. Sie werden sehen, es funktioniert – allerdings nicht besonders elegant. Denn in der Liste stehen nur die englischen Namen der Stile.

Sehr viel schöner wäre es, dem Anwender die Linien in der Liste gleich in einer Vorschau zu präsentieren. Auch das geht – und zwar, indem Sie das Ereignis **DrawItem**²⁰ für das Listenfeld programmieren. Dieses Ereignis tritt jedes Mal ein, wenn ein Element in der Liste gezeichnet werden muss. Wir können hier also Linien in den verschiedenen Stilen direkt auf die Zeichenfläche jedes einzelnen Elements in der Liste „malen“.

Damit das funktioniert, müssen Sie im ersten Schritt die Eigenschaft **DrawMode** des Listenfelds in `OwnerDrawFixed`²¹ ändern. Sie finden diese Eigenschaft in der Gruppe **Verhalten** des Eigenschaftenfensters.

Das Ereignis **DrawItem** für ein Listenfeld tritt nur dann ein, wenn die Eigenschaft **DrawMode** nicht auf `Normal` steht.



Im nächsten Schritt ändern Sie die Anweisung

```
listBoxLinieStil.Items.Add(element) ;
```

im Ereignis **Load** für das Formular bitte in

```
listBoxLinieStil.Items.Add("") ;
```

Damit erzeugen wir leere Einträge in der Liste.

20. **DrawItem** bedeutet so viel wie „Zeichne Element“.

21. `OwnerDrawFixed` steht für „Selbst gezeichnet in fester Größe“.

Danach können wir auch das „Malen“ der Linien in die Liste beim Ereignis **DrawItem** programmieren. Die vollständige Methode für das Ereignis sieht so aus:

```
private void ListBoxLinieStil_DrawItem(object sender,
DrawItemEventArgs e)
{
    //eine lokale Variable für die Berechnung der Mitte
    int y;
    //ein neuer lokaler Stift
    Pen boxStift = new Pen(Color.Black);
    //die Mitte berechnen
    y = (e.Bounds.Top + e.Bounds.Bottom) / 2;
    //den Hintergrund zeichnen
    e.DrawBackground();
    //und die Linie
    boxStift.DashStyle = linienstil[e.Index];
    //bitte in einer Zeile eingeben
    e.Graphics.DrawLine(boxStift, e.Bounds.Left+1, y,
    e.Bounds.Right-1, y);
}
```

Code 4.8: Das Zeichnen in die Liste für die Linienstile

Zunächst einmal vereinbaren wir eine lokale Variable `y`, um später etwas leichter die Mitte für das Zeichnen der Linie ansprechen zu können.

Danach erzeugen wir einen neuen lokalen Stift, den wir für das Zeichnen der Linien benutzen.

Mit der Anweisung

```
y = (e.Bounds.Top + e.Bounds.Bottom) / 2;
```

ermitteln wir dann die Mitte für die Zeichnung – also die y-Koordinate der Linie. Dazu addieren wir die obere und die untere Grenze des einzelnen Listeneintrags und dividieren das Ergebnis durch 2. Die Grenzen erhalten wir dabei über den Ausdruck `e.Bounds`²². `e` wird dabei als Parameter vom Typ `DrawItemEventArgs` an die Methode übergeben.

Danach lassen wir über die Anweisung

```
e.DrawBackground();
```

den Hintergrund für jeden Eintrag in der Liste zeichnen. Normalerweise wird dieser Hintergrund automatisch gezeichnet. Aber da wir das Ereignis **DrawItem** selbst programmieren, müssen wir auch den Hintergrund selbst zeichnen lassen.

Mit den folgenden Anweisungen wird dann die Linie in das Listenelement gezeichnet. Der Stil wird dabei über den Ausdruck `linienstil[e.Index]` festgelegt. `e.Index` steht dabei für das aktuelle Element, das gerade in der Liste gezeichnet wird. Da das Ereignis **DrawItem** für jeden einzelnen Listeneintrag eintritt, erhalten wir als Endergebnis ein Listenfeld mit den gezeichneten Linien in den verschiedenen Stilen.

22. *Bounds* bedeutet übersetzt „Grenzen“.

Bitte beachten Sie:

Das Ereignis **DrawItem** tritt nur dann ein, wenn die Eigenschaft **DrawMode** eines Listenfelds nicht auf dem Standardeintrag **Normal** steht. Falls bei Ihnen also beim Test eine leere Liste angezeigt wird, überprüfen Sie noch einmal, ob Sie die Eigenschaft **DrawMode** des Listenfelds korrekt auf **OwnerDrawFixed** gesetzt haben.



Das Listenfeld mit den Hintergrundmustern können Sie jetzt mit ähnlichen Techniken erstellen. Zuerst vereinbaren Sie als neues Feld für die Klasse ein Array vom Typ `System.Drawing.Drawing2D.HatchStyle` und legen in diesem Array einige Hintergrundmuster ab.

```
System.Drawing.Drawing2D.HatchStyle[] fuellstil =
{
    System.Drawing.Drawing2D.HatchStyle.BackwardDiagonal,
    System.Drawing.Drawing2D.HatchStyle.Cross,
    System.Drawing.Drawing2D.HatchStyle.DottedGrid,
    System.Drawing.Drawing2D.HatchStyle.ForwardDiagonal,
    System.Drawing.Drawing2D.HatchStyle.Sphere,
    System.Drawing.Drawing2D.HatchStyle.Vertical,
    System.Drawing.Drawing2D.HatchStyle.Wave,
    System.Drawing.Drawing2D.HatchStyle.ZigZag
};
```

Code 4.9: Das Array für die Hintergrundmuster

Danach erzeugen Sie im Ereignis **Load** des Formulars eine leere Liste mit der nötigen Anzahl Einträge.

```
//bitte in einer Zeile eingeben
foreach (System.Drawing.Drawing2D.HatchStyle element in
fuellstil)
    listBoxHintergrundMuster.Items.Add ("");
```

Code 4.10: Das Erzeugen der Liste für die Hintergrundmuster

Anschließend setzen Sie die Eigenschaft **DrawMode** für das Listenfeld auf **OwnerDrawFixed** und lassen dann im Ereignis **DrawItem** Rechtecke mit den verschiedenen Mustern zeichnen. Die entsprechende Methode sieht so aus:

```
private void ListBoxHintergrundMuster_DrawItem(object
sender, DrawItemEventArgs e)
{
    //ein neuer lokaler Pinsel für das Muster
    //bitte in einer Zeile eingeben
    System.Drawing.Drawing2D.HatchBrush boxPinsel = new
    System.Drawing.Drawing2D.HatchBrush(fuellstil
    [e.Index], Color.Black, Color.White);
    //den Hintergrund zeichnen
    e.DrawBackground();
    //und das Rechteck
```

```
//bitte in einer Zeile eingeben
e.Graphics.FillRectangle(boxPinsel,
e.Bounds.Left+1, e.Bounds.Top+1, e.Bounds.Width-1,
e.Bounds.Height-1);
}
```

Code 4.11: Das Zeichnen in die Liste für die Hintergrundmuster

Die Zuweisung des ausgewählten Füllmusters erfolgt dann wieder im Ereignis **Click** für die Schaltfläche **Los geht's**. Dazu überprüfen Sie beim Kreis und beim Rechteck, ob das Optionsfeld **Muster** markiert ist und gleichzeitig ein Eintrag im Listenfeld für das Muster ausgewählt wurde. Wenn das der Fall ist, erzeugen Sie einen neuen Pinsel für das Muster und zeichnen eine gefüllte Figur. Für den Kreis könnten die entsprechenden Anweisungen so aussehen (sie sind im folgenden Code fett markiert):

```
...
if (radioButtonKreis.Checked == true)
{
    if (radioButtonHintergrundOhne.Checked == true)
        zeichenflaeche.DrawEllipse(stift,
        panel1.ClientRectangle.Left + groesse,
        panel1.ClientRectangle.Top + groesse,
        panel1.ClientRectangle.Width - (groesse*2),
        panel1.ClientRectangle.Height - (groesse *
        2));
    if (radioButtonHintergrundFarbe.Checked == true)
        zeichenflaeche.FillEllipse(pinsel,
        panel1.ClientRectangle.Left + groesse,
        panel1.ClientRectangle.Top + groesse,
        panel1.ClientRectangle.Width - (groesse*2),
        panel1.ClientRectangle.Height - (groesse *
        2));
    //soll mit Muster gezeichnet werden und ist ein
    //Muster ausgewählt?
    //bitte in einer Zeile eingeben
    if (radioButtonHintergrundMuster.Checked == true &&
    listBoxHintergrundMuster.SelectedIndex >= 0)
    {
        //einen neuen Pinsel für das Muster erzeugen
        //die Vordergrundfarbe kommt vom Stift, der
        //Hintergrund ist immer weiß
        //bitte jeweils in einer Zeile eingeben
        System.Drawing.Drawing2D.HatchBrush
        musterPinsel = new System.Drawing.Drawing2D.
        HatchBrush(fuellstil[listBoxHintergrundMuster.
        SelectedIndex], stift.Color, Color.White);
        zeichenflaeche.FillEllipse(musterPinsel,
        panel1.ClientRectangle.Left + groesse,
        panel1.ClientRectangle.Top + groesse,
        panel1.ClientRectangle.Width - (groesse * 2),
        panel1.ClientRectangle.Height - (groesse * 2));
    }
}
...
```

Code 4.12: Das Zeichnen eines Kreises mit Muster

Hinweis:

Wenn ein Anwender das Optionsfeld **Muster** markiert, aber keinen Eintrag in der Liste auswählt, geschieht nichts weiter. Sie können hier ja – wenn Sie wollen – noch eine Meldung einbauen, dass auch ein Muster ausgewählt werden muss. Wir haben darauf verzichtet, damit der Code nicht zu unübersichtlich wird.

Damit ist unsere Spielerei fertiggestellt. Eine Erweiterung wartet noch als Einsendeaufgabe auf Sie.

Abschließend noch ein Hinweis:

Wenn Sie das Fenster der Spielerei minimieren und danach wiederherstellen, sind die Zeichnungen verschwunden. Ähnliches geschieht unter Umständen auch, wenn ein anderes Fenster über dem Fenster der Spielerei liegt. Auch hier gehen möglicherweise die Teile der Zeichnung verloren, die von dem anderen Fenster verdeckt werden.

Das liegt daran, dass das Formular beziehungsweise Teile des Formulars neu gezeichnet werden, sobald es wiederhergestellt wird oder wenn ein anderes Fenster, das über dem Formular lag, geschlossen wird.

Da unsere Zeichenfläche aber nur beim Anklicken einer Schaltfläche mit den Figuren gefüllt wird, geht der Inhalt beim Neuzeichnen in Teilen oder sogar komplett verloren.

Zeichnungen, die ständig auf dem Formular zu sehen sein sollen, müssen Sie daher im Ereignis **Paint** des Formulars erstellen lassen. Dieses Ereignis tritt immer dann ein, wenn das Formular oder Teile des Formulars neu gezeichnet werden. Die Anweisungen in dem Ereignis werden allerdings noch vor dem Zeichnen der Elemente ausgeführt, die fest im Formular hinterlegt sind. Bei unserer Spielerei würde also zum Beispiel das Panel, in dem wir unsere Objekte erstellt haben, erst nach den Anweisungen im Ereignis **Paint** erstellt. Damit würde es über Zeichnungen liegen, die im Ereignis **Paint** an derselben Stelle im Formular gezeichnet werden.

Bei Zeichnungen, die nur nach dem Anklicken eines Steuerelements erstellt werden sollen, gibt es mehrere Möglichkeiten, den ursprünglichen Zustand beim Neuzeichnen wiederherzustellen:

- Sie lagern die Zeichenfunktionen in das Ereignis **Paint** des Formulars aus und lassen dann beim Anklicken des Steuerelements das Formular neu zeichnen. Dazu rufen Sie die Methode `Invalidate()` auf.
- Sie rufen im Ereignis **Paint** die Methode auf, die auch beim Anklicken des Steuerelements ausgeführt wird – in unserer Spielerei zum Beispiel die Methode `ButtonStart_Click()`.
- Sie erstellen eine eigene Methode für die Zeichnungen. Diese Methode rufen Sie dann sowohl beim Anklicken des Steuerelements als auch im Ereignis **Paint** auf.

Falls das Auslagern in das Ereignis **Paint** nicht möglich ist – zum Beispiel, weil Sie mehrere sehr unterschiedliche Aktionen ausführen lassen wollen – können Sie auch versuchen, den Zustand auf der Zeichenfläche zu sichern – zum Beispiel in verschiedenen Feldern. In diesen Feldern legen Sie bei jeder Änderung die aktuellen Daten der Zeichnung ab und lassen dann im Ereignis **Paint** die Zeichnung mit den Daten aus den Feldern wiederherstellen.

Auch wenn sich so das Verschwinden der Zeichnung vermeiden lässt, haben alle Techniken auch einige Nachteile:

- Das Ereignis **Paint** tritt auch unmittelbar nach dem Start des Programms ein. Denn auch hier muss das Formular ja neu gezeichnet werden.
- Das Ereignis **Paint** tritt häufig auch dann ein, wenn Sie das Formular über den Rahmen in der Größe verändern oder wenn andere Fenster über dem Formular liegen. Bei diesen Aktionen geht aber der Inhalt der Zeichenfläche normalerweise nicht verloren, da nur die Teile neu gezeichnet werden, die verdeckt waren. Aufwendige Aktionen im Ereignis **Paint** können dann dazu führen, dass das Fenster beim Verkleinern oder Vergrößern über den Bildschirm springt oder sogar so langsam reagiert, dass der Anwender meint, das Programm sei abgestürzt.
- Das Ereignis **Paint** tritt ebenfalls ein, wenn Sie zum Beispiel den Mauszeiger auf eine Schaltfläche stellen. Denn hier wird ja in der Standardeinstellung der Hintergrund der Schaltfläche zur Hervorhebung in einer anderen Farbe gezeichnet. Und das löst das Ereignis **Paint** aus.

Der Aufwand für das Wiederherstellen der Zeichnungen ist also recht hoch. Ob sich dieser Aufwand für Ihre Anwendung lohnt und welche Technik die beste ist, müssen Sie im Einzelfall selbst entscheiden.

Zusammenfassung

Über das Steuerelement **TrackBar** können Sie einen Schieberegler einfügen. Sie finden das Steuerelement in der Gruppe **Alle Windows Forms** der Toolbox.

Über das Steuerelement **ColorDialog** fügen Sie einen Dialog zur Farbauswahl ein. Sie finden das Steuerelement in der Gruppe **Dialogfelder** der Toolbox.

Falls die Eigenschaft **DrawMode** für ein Listenfeld nicht auf `Normal` gesetzt ist, tritt das Ereignis **DrawItem** ein, wenn ein Element in der Liste gezeichnet werden muss.

Wenn das Formular oder Teile des Formulars neu gezeichnet werden müssen, tritt das Ereignis **Paint** ein. Über die Methode `Invalidate()` können Sie das Neuzeichnen auch selbst durchführen lassen.

Aufgaben zur Selbstüberprüfung

- 4.1 Über welche Eigenschaften setzen Sie den kleinsten und den größten Wert, der über ein Drehfeld ausgewählt werden kann?

- 4.2 Über welche Eigenschaft erhalten Sie den aktuellen Wert eines Schiebereglers?

- 4.3 Sie wollen den Anwender über einen Dialog eine Farbe auswählen lassen. Ein entsprechendes Steuerelement mit dem Namen `colorDialog2` haben Sie bereits eingefügt. Mit welcher Anweisung lassen Sie den Dialog anzeigen? Wie können Sie dabei gleichzeitig ermitteln, ob der Anwender den Dialog über die Schaltfläche **OK** verlassen hat?

- 4.4 Sie lassen die Elemente in einem Listenfeld über das Ereignis **DrawItem** selbst zeichnen. Wie ermitteln Sie das Element der Liste, das gerade gezeichnet wird?

Schlussbetrachtung

Herzlichen Glückwunsch!

Sie haben erfolgreich die ersten Schritte bei der Grafikprogrammierung gemeistert. Sie wissen jetzt, was es mit der Klasse `Graphics` auf sich hat, wie verschiedene Figuren gezeichnet werden und wie Sie die Eigenschaften dieser Figuren über den Stift und die Pinsel verändern können.

Außerdem haben Sie gelernt, wie Sie dem Anwender verschiedene Farben über einen Dialog anbieten und wie Sie Zeichnungen direkt in einem Listenfeld durchführen.

Die kleine grafische Spielerei, die Sie am Ende dieses Studienhefts erstellt haben, können Sie noch ein wenig verfeinern. So gibt es ja zum Beispiel für Linien keine Füllmuster. Sie könnten also die Felder in der Gruppe **Hintergrund** deaktivieren, wenn der Anwender eine Linie zeichnen lassen möchte. Denken Sie aber daran, die Felder auch wieder zu aktivieren, wenn der Anwender andere Einträge auswählt.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 a) Der Punkt 0,0 liegt ganz oben links in der Ecke. Der Abstand zum linken und oberen Rand ist 0.
 b) Der Punkt 200,20 liegt 200 Punkte vom linken Rand und 20 Punkte vom oberen Rand entfernt.
 c) Der Punkt 10,50 liegt 10 Punkte vom linken Rand und 50 Punkte vom oberen Rand entfernt.
- 1.2 Den rechten Rand ermitteln Sie mit der Eigenschaft `ClientSize.Width`. Den unteren Rand ermitteln Sie mit der Eigenschaft `ClientSize.Height`.

Alternativ können Sie auch die Eigenschaft `ClientRectangle` verwenden.

Für die richtige Lösung reicht es aus, wenn Sie eine der beiden Möglichkeiten angegeben haben.

- 1.3 Sie übergeben vier einzelne Argumente vom Typ `int`. Alternativ können Sie die Daten auch gemeinsam über den Typ `Rectangle` übergeben.
- 1.4 Die Methode zum Löschen der Zeichenfläche heißt `Clear()`. Als Argument übergeben Sie die Farbe, in der der Zeichenbereich nach dem Löschen neu gezeichnet werden soll.

Kapitel 2

- 2.1 Die Anweisung lautet:

```
zeichenflaeche.DrawLine(stift, 10, 10, 100, 100);
```

`zeichenflaeche` steht dabei für die Zeichenfläche und `stift` für den Stift.

- 2.2 Die Eckpunkte werden in einem Array vom Typ `Point` festgelegt.
- 2.3 Bei der Methode `DrawPolygon()` wird der letzte Punkt automatisch mit dem ersten verbunden, bei der Methode `DrawLines()` nicht.
- 2.4 Der Kreis wird durch ein umgebendes Rechteck beschrieben. Die Argumente geben den Stift, die Koordinaten für die linke obere Ecke sowie die Breite und die Höhe an.

Kapitel 3

- 3.1 Die Gestaltung der Linien wird von der Klasse `Pen` gesteuert. Die Gestaltung der Füllungen erfolgt über die Klasse `Brush` beziehungsweise entsprechende abgeleitete Klassen.

- 3.2 Die Anweisungen lauten:

```
stift.DashStyle = System.Drawing.Drawing2D.  
DashStyle.Dash;  
stift.Width = 2;
```

Der Bezeichner für den Stift ist dabei beliebig.

- 3.3 Im ersten Schritt vereinbaren Sie eine Variable vom Typ `Bitmap`. Über diese Variable laden Sie mit der Methode `Image.FromFile()` die gewünschte Datei. Dabei müssen Sie die Rückgabe der Methode in den Typ `Bitmap` umwandeln. Danach erzeugen Sie eine neue Instanz der Klasse `TextureBrush` und weisen diesem Pinsel das Bild zu. Im letzten Schritt lassen Sie dann das gefüllte Objekt mit dem Pinsel der Klasse `TextureBrush` zeichnen.

- 3.4 Kopien eines Pinsels und eines Stifts erzeugen Sie mit der Methode `Clone()`.

- 3.5 Die Farbe wird durch den Pinsel bestimmt.

Kapitel 4

- 4.1 Der kleinste Wert für ein Drehfeld wird über die Eigenschaft **Minimum** gesetzt. Der größte Wert wird über die Eigenschaft **Maximum** gesetzt.

- 4.2 Den aktuellen Wert eines Schiebereglers erhalten Sie über die Eigenschaft **Value**.

- 4.3 Die Anweisung könnte so aussehen:

```
if (colorDialog2.ShowDialog() == DialogResult.OK)
```

- 4.4 Das Element, das gerade gezeichnet wird, erhalten Sie über den Ausdruck `e.Index`. `e` wird automatisch an die Methode übergeben.

B. Glossar

Aufzählungstyp	Über Aufzählungstypen können Sie die möglichen Werte eines Typs auf eine bestimmte Auswahl begrenzen – zum Beispiel auf Wochentage. Ein Aufzählungstyp ist ein benutzerdefinierter Datentyp.
Casting	Siehe Typecasting
Client-Bereich	Der Client-Bereich ist der innere Bereich eines Steuerlements ohne Rahmen oder Ähnliches. Bei einem Formular stellt der Client-Bereich zum Beispiel das Innere des Formulars dar.
Garbage Collection	Die Garbage Collection gibt Speicher frei und reorganisiert den Speicher. Sie wird automatisch ausgeführt.
Garbage Collector	Der Garbage Collector ist der Prozess, der die Garbage Collection ausführt.
GDI+	GDI+ ist eine geräteunabhängige Programmierschnittstelle für die Ausgabe von Informationen auf grafikfähigen Geräten.
GDI+ API	Siehe GDI+
Graphics Device Interface+ Application Programming Interface	Siehe GDI+
Pixel	Ein Pixel ist ein Bildpunkt.
Polygon	Ein Polygon ist ein Vieleck – eine geschlossene geometrische Figur, die aus mehreren Linien besteht.
Polylinie	Eine Polylinie ist eine Linie, die über mehrere Punkte verläuft.
Referenz	Eine Referenz ist ein Verweis auf ein Objekt.
Timer	Ein <i>Timer</i> löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis aus, in dem Sie Anweisungen verarbeiten lassen können. Ein <i>Timer</i> ermöglicht damit die zeitgesteuerte Verarbeitung von Anweisungen.
Typecasting	<i>Typecasting</i> bezeichnet das Umwandeln eines Wertes in einen anderen Datentyp.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in Visual C# 2019. Ideal für Programmieranfänger geeignet.* 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Das Koordinatensystem von Graphics	5
Abb. 1.2	Ein Rechteck im Formular	5
Abb. 1.3	Zwei Rechtecke in dem Formular	7
Abb. 1.4	Noch einmal zwei Rechtecke	8
Abb. 1.5	Ein erster grafischer Effekt	9
Abb. 1.6	Rechtecke in verschiedenen Zeichenbereichen	11
Abb. 2.1	Ein Gitternetz	18
Abb. 2.2	Mehrere sich überlagernde Zeichenobjekte	20
Abb. 2.3	Kreise und Ellipsen	21
Abb. 2.4	Ein Polygon	22
Abb. 2.5	Das Polygon im Formular	23
Abb. 2.6	Eine Linie über mehrere Punkte	23
Abb. 2.7	Die Hilfe zur Klasse Graphics	25
Abb. 3.1	Unterschiedliche Linien und Füllmuster	30
Abb. 3.2	Unterschiedliche Linienenden	31
Abb. 3.3	Figuren mit Verläufen	34
Abb. 4.1	Eine Spielerei	38
Abb. 4.2	Das Formular mit den Steuerelementen	41
Abb. 4.3	Das Formular nach dem Setzen der Eigenschaften	43
Abb. 4.4	Das Programm in Aktion	47
Abb. 4.5	Der Dialog zur Farbauswahl	49
Abb. 4.6	Das Hilfedokument zur DashStyle-Enumeration	52

E. Tabellenverzeichnis

Tab. 2.1	Weitere Methoden für geometrische Figuren	24
Tab. 3.1	Die Eigenschaft DashStyle der Klasse Pen	28
Tab. 3.2	Die Eigenschaften StartCap und EndCap für die Klasse Pen	28
Tab. 3.3	Einige Muster der Klasse System.Drawing.Drawing2D.HatchBrush	29
Tab. 4.1	Die Bezeichner der Steuerelemente im Projekt Spielerei	41

F. Codeverzeichnis

Code 1.1	Das Zeichnen eines Rechtecks	4
Code 1.2	Ein Rechteck in der Größe des Client-Bereichs	6
Code 1.3	Veränderte Koordinaten über die Variable bereich	7
Code 1.4	Ein erster grafischer Effekt	9
Code 1.5	Eine Animation	10
Code 1.6	Zeichnen in verschiedenen Zeichenflächen	11
Code 1.7	Das Beschaffen der Zeichenfläche für das Formular	12
Code 1.8	Der Start der Animation über eine Schaltfläche	13
Code 1.9	Das Steuern der Verzögerung über einen Timer	14
Code 1.10	Eine verbogene Referenz für die Zeichenfläche	15
Code 2.1	Ein Gitternetz	18
Code 2.2	Erweiterung zum vorigen Code	19
Code 2.3	Verschiedene Kreise	20
Code 3.1	Linien und Füllmuster	30
Code 3.2	Unterschiedliche Linienenden	31
Code 3.3	Eine Grafik als Füllmuster	32
Code 3.4	Einige Verläufe	33
Code 3.5	Eine Textausgabe auf der Zeichenfläche	35
Code 3.6	Textausgabe in einem Rechteck	35
Code 3.7	Kopien von Stift und Pinsel erzeugen	35
Code 4.1	Die Methode Form1_Load()	43
Code 4.2	Die Anweisungen zum Zeichnen	46
Code 4.3	Die Methode zur Farbauswahl für die Linie	48
Code 4.4	Die Methode zur Farbauswahl für den Hintergrund	50
Code 4.5	Die geänderten Anweisungen in der Methode ButtonStart_Click()	50
Code 4.6	Das Füllen des Listenfeldes für die Linienstile	52
Code 4.7	Das Setzen des Linienstils	53
Code 4.8	Das Zeichnen in die Liste für die Linienstile	54
Code 4.9	Das Array für die Hintergrundmuster	55
Code 4.10	Das Erzeugen der Liste für die Hintergrundmuster	55
Code 4.11	Das Zeichnen in die Liste für die Hintergrundmuster	56
Code 4.12	Das Zeichnen eines Kreises mit Muster	56

G. Sachwortverzeichnis

C	V
Client-Bereich	Verlauf
	33
E	Z
Ellipse	Zeichenfläche
	Beschaffen der
Ereignis	12
DrawItem	Zugriff auf die
Paint	3
G	
GDI+	
	3
Grafik	
Text in	34
K	
Klasse	
Brush	28
Graphics	3
Pen	27
SolidBrush	29
Koordinatensystem	5
Kopie	
von Stift und Pinsel erzeugen	35
Kreis	20
gefüllter	24
Kreisbogen	24
Kurve	24
geschlossene	24
L	
Linie	17
Linienstil	28
M	
Methode	12
P	
Polygon	22
gefülltes	24
S	
Steuerelement	
ColorDialog	48

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP10D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

1. Sie wollen einen Kreis mit dem Durchmesser 100 Punkte zeichnen. Der Mittelpunkt des Kreises soll an der Position 150, 150 liegen. Wie lautet die entsprechende Anweisung, um den Kreis zu erstellen?

10 Pkt.

2. Erweitern Sie die grafische Spielerei um Animationen für die Elemente ohne Füllungen. Die Kreise beziehungsweise Rechtecke sollen dabei von innen nach außen schrittweise vergrößert werden. Bei den Linien soll die Animation von der Mitte sowohl nach unten als auch nach oben laufen.

Bei jedem Schritt soll das alte Element gelöscht und ein neues gezeichnet werden.

Wenn der äußere Rand erreicht ist, soll sich die Animation wieder nach innen bewegen, bis ungefähr die ursprüngliche Größe beziehungsweise die ursprüngliche Position erreicht ist.

Die Anzahl der Wiederholungen für die Animation und die Geschwindigkeit der Animation soll der Anwender über geeignete Steuerelemente selbst festlegen können.

Wie Sie die Lösung umsetzen, ist Ihnen weitgehend freigestellt. Dokumentieren Sie Ihren Ansatz aber ausführlich.

Schicken Sie für die Lösung bitte das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

90 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Pong – ein Computerspiel

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0819N01

CSHP11D

Objektorientierte Software-Entwicklung mit C#

Pong – ein Computerspiel

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Pong – ein Computerspiel

Inhaltsverzeichnis

Einleitung	1
1 Spielregeln und Vorüberlegungen	3
2 Spielfeld erstellen	5
2.1 Menüleiste erstellen	5
2.2 Ball und Schläger einfügen	10
2.3 Das Zeichnen des Spielfelds	14
2.4 Das Bewegen des Schlägers	15
2.5 Das Bewegen des Balls	17
2.6 Das Zurückschlagen	18
Zusammenfassung	22
3 Starten und Anhalten	23
3.1 Vorüberlegungen	23
3.2 Die Vorbereitungen	24
3.3 Unterbrechen und Fortsetzen	27
3.4 Neu starten	29
3.5 Das Spielende	31
Zusammenfassung	32
4 Die Bestenliste	34
4.1 Vorüberlegungen	34
4.2 Die Klasse Score	35
4.3 Die Ausgabe der Punkte	36
4.4 Die Verwaltung der Liste	38
4.5 Die Anzeige der Liste	46
Zusammenfassung	49
5 Die Spieleinstellungen	51
5.1 Der Schwierigkeitsgrad	51
5.2 Die Spielfeldgröße	54
Zusammenfassung	57
Schlussbetrachtung	59

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	60
B.	Glossar	62
C.	Literaturverzeichnis	64
D.	Abbildungsverzeichnis	65
E.	Tabellenverzeichnis	66
F.	Codeverzeichnis	67
G.	Sachwortverzeichnis	69
H.	Einsendeaufgabe	71

Einleitung

In diesem Studienheft werden Sie eine Art „Bildschirmtennis“ für einen Spieler programmieren. Das Spiel lehnt sich an eines der ersten Computerspiele überhaupt an: das Spiel „Pong“.

Die Grundidee des Spiels ist es, einen Ball, der sich über den Bildschirm bewegt und an den Rändern abprallt, mit einem beweglichen Schläger zu treffen und so wieder zurückzuschlagen.

Wir werden dieses Spiel wie gewohnt Schritt für Schritt umsetzen und immer weiter ausbauen.

Im Einzelnen lernen Sie in diesem Studienheft,

- wie Sie Menüleisten erstellen,
- wie Sie den Schläger in einem Mausereignis bewegen,
- wie Sie den Ball über einen Timer bewegen,
- wie Sie zufällige Zahlen erzeugen,
- wie Sie auf Kollisionen zwischen dem Ball und dem Schläger reagieren,
- wie Sie verschiedene Klassen zusammenarbeiten lassen,
- wie Sie das Spiel unterbrechen, fortsetzen und neu starten,
- wie Sie die Spielzeit auf dem Bildschirm anzeigen,
- wie Sie eine Bestenliste für das Spiel programmieren,
- wie Sie Daten sortieren lassen,
- wie Sie unterschiedliche Schwierigkeitsgrade im Spiel programmieren und
- wie Sie die Spielfeldgröße durch den Anwender einstellen lassen.

Bevor wir mit der Programmierung beginnen, legen wir zunächst einmal die Spielregeln fest und machen uns einige Gedanken, wie sich das Spiel umsetzen lässt.

Christoph Siebeck

1 Spielregeln und Vorüberlegungen

In diesem Kapitel stellen wir Ihnen die Spielregeln vor und machen uns einige Gedanken zur praktischen Umsetzung.

Beginnen wir mit den Spielregeln.

- Das Spielfeld besteht aus Balken oben, hinten und unten sowie einem Schläger am linken Rand und einem Ball. Der Ball bewegt sich über das Spielfeld und ändert seine Richtung, wenn er auf einen der Balken am Rand trifft.
- Der Spieler muss verhindern, dass der Ball am linken Rand aus dem Spielfeld verschwindet. Dazu kann er ihn mit dem Schläger zurückspielen. Der Schläger kann dabei nur nach oben und unten, aber nicht nach vorne oder hinten bewegt werden.
- Für jedes erfolgreiche Zurückschlagen erhält der Spieler in der Standardeinstellung einen Punkt.
- Ein Spiel soll zwei Minuten dauern.
- Der Spieler kann zwischen verschiedenen Schwierigkeitsgraden wählen, die die Geschwindigkeit und den Bewegungswinkel des Balls sowie die Größe des Schlägers verändern. Je höher der Schwierigkeitsgrad ist, desto mehr Punkte gibt es für jedes Zurückschlagen.
- Die erzielten Punkte werden in einer Bestenliste abgelegt. Diese Bestenliste steht dabei zunächst nur so lange zur Verfügung, wie die Anwendung läuft.

Hinweis:

Um das Speichern der Bestenliste auf einem Datenträger werden wir uns später kümmern.

So viel zu den Spielregeln. Bevor wir jetzt mit der Programmierung beginnen, wie gewohnt noch einige Vorüberlegungen:

Wir werden für das Spiel zwei Klassen anlegen, die das Formular und die Logik des Spiels sowie die Verwaltung der Bestenliste abbilden. Auf diese Weise können wir das Spiel Schritt für Schritt erweitern und die Bestenliste später auch in anderen Projekten wieder verwenden.

Das Spielfeld soll zunächst eine feste Größe von 640×480 Punkten haben. Später werden wir noch eine Funktion einbauen, in der der Spieler auch andere Größen auswählen kann.

Die Bewegungen des Schlägers steuern wir über die Maus. Bei gedrückter linker Maustaste folgt der Schläger dem Mauszeiger nach oben oder nach unten. Bewegungen nach vorne oder hinten sind ja nicht möglich.

Für den Schläger und den Ball benutzen wir zwei Panels. Dann können wir nämlich sowohl den Ball als auch den Schläger ohne Probleme animieren, da die Steuerelemente ja über dem Hintergrund des Formulars liegen.

Die Funktionen zur Steuerung des Spiels und zum Setzen der Einstellungen rufen wir über eine Menüleiste auf.

Das fertige Spiel sollte ungefähr so aussehen:

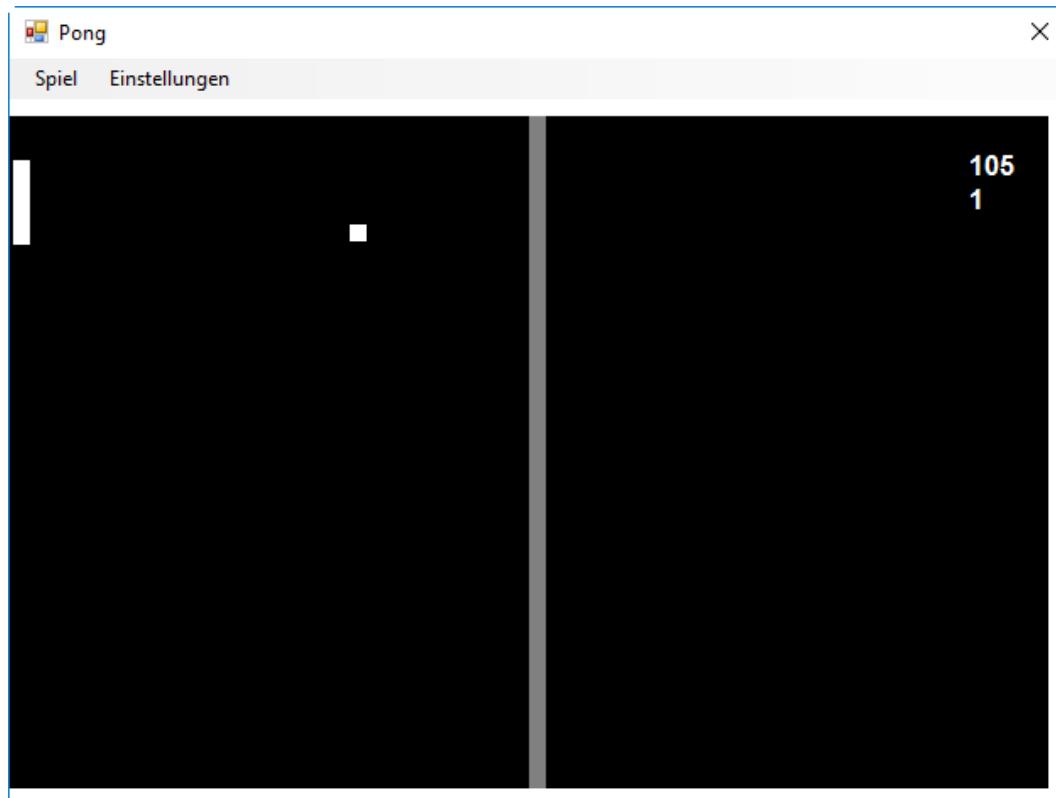


Abb. 1.1: Das Pong-Spiel

Beginnen wir jetzt mit dem Erstellen des Spielfelds.

2 Spielfeld erstellen

In diesem Kapitel erstellen wir das Spielfeld.

Legen Sie eine neue leere Windows Forms-Anwendung für unser Spiel an. Damit keine Änderungen der Formulargröße über den Rand oder die Symbole in der Titelleiste möglich sind, ändern Sie die Eigenschaft **FormBorderStyle** in `FixedSingle` und setzen Sie die Eigenschaften **MaximizeBox** sowie **MinimizeBox** jeweils auf `False`. Setzen Sie dann die Eigenschaften **Size/Width** und **Size/Height** auf `640` beziehungsweise auf `480`. Ändern Sie außerdem noch die Eigenschaft **Text** für das Formular – zum Beispiel in `Pong`.

Tipp:

Über die Eigenschaft **StartPosition** in der Gruppe **Layout** können Sie das Formular auch automatisch positionieren lassen. Der Wert `CenterScreen` zentriert das Formular zum Beispiel auf dem Desktop.

2.1 Menüleiste erstellen

Fügen wir jetzt eine Menüleiste ein. Das erfolgt über das Steuerelement **MenuStrip**¹. Sie finden es in der Gruppe **Menüs & Symbolleisten** der Toolbox.

Nach dem Einfügen erscheint eine Menüleiste oben im Formular. Außerdem wird das Steuerelement unterhalb des Formulars im Bereich für die nicht visuellen Elemente angezeigt.

1. *Strip* bedeutet übersetzt so viel wie „Streifen“ oder „Leiste“.

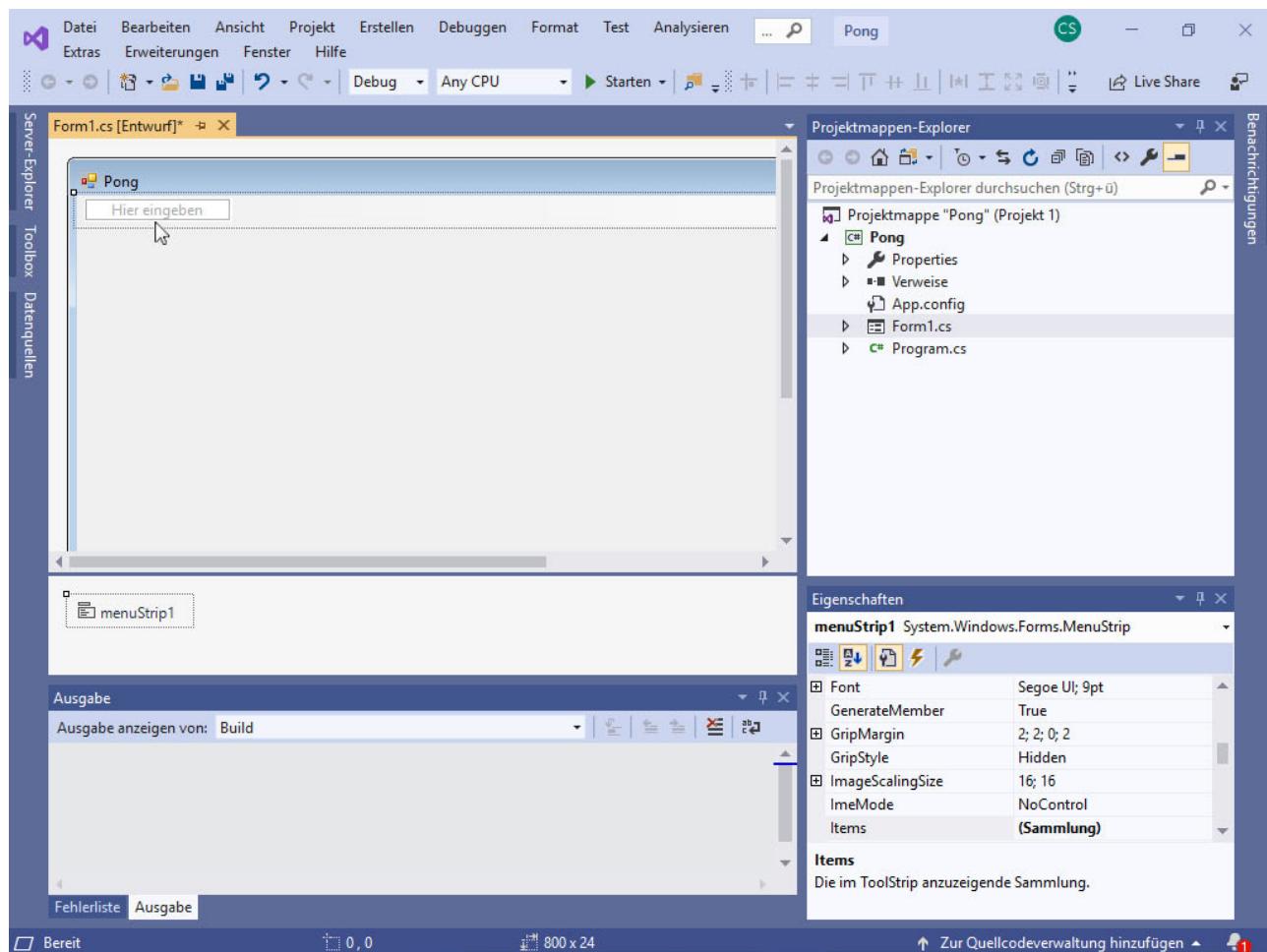


Abb. 2.1: Die Menüleiste im Formular
(oben links in der Abbildung am Mauszeiger)

Das Erstellen und Bearbeiten der eigentlichen Menüs erfolgt am bequemsten über die Eigenschaft **Items** in der Gruppe **Daten** des **MenuStrips**. Stellen Sie dazu die Einfügemarke in das Feld hinter der Eigenschaft und klicken Sie dann mit der Maus auf das Symbol rechts im Feld.

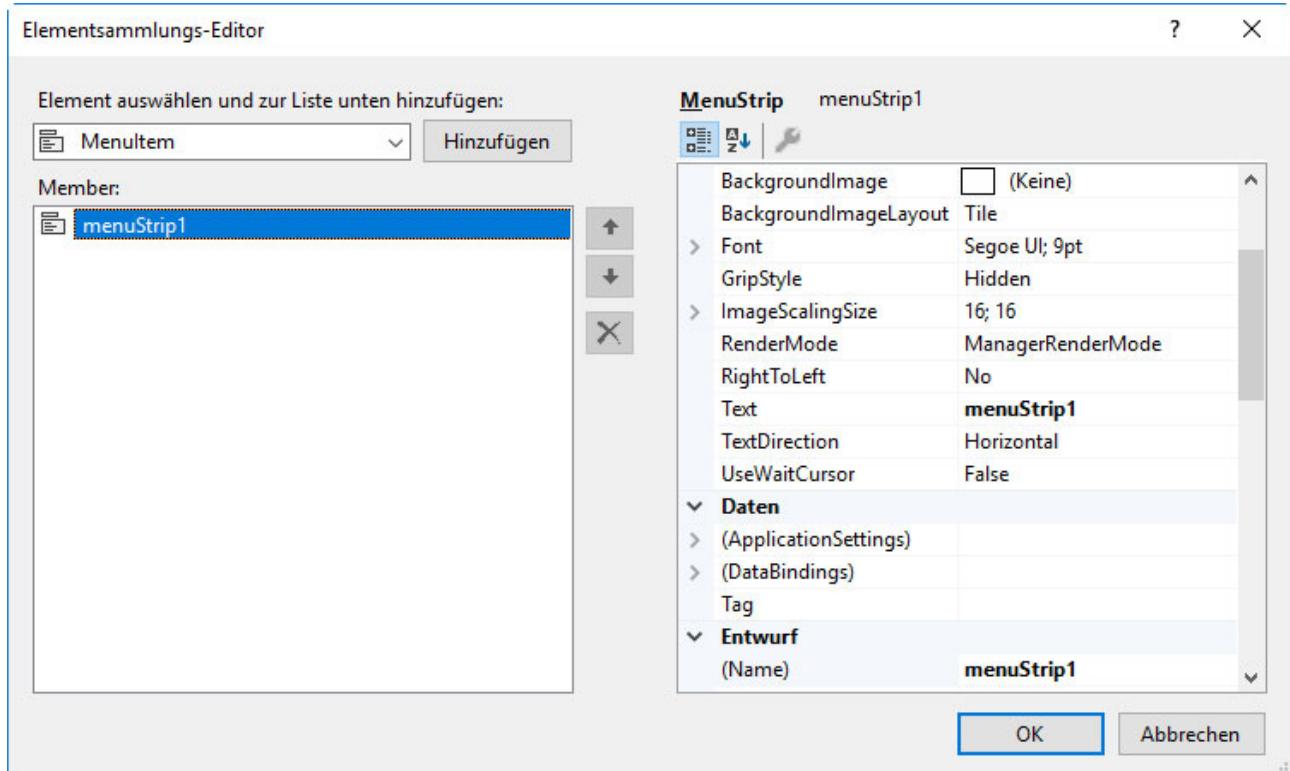


Abb. 2.2: Der Elementsammlungs-Editor

Im **Elementsammlungs-Editor** können Sie dann über die Schaltfläche **Hinzufügen** oben im Fenster neue Einträge hinzufügen. Welches Element dabei eingefügt wird, legen Sie über das Kombinationsfeld links neben der Schaltfläche fest. Neben einem **MenuItem** – einem normalen Menüeintrag – können Sie hier auch ein Kombinationsfeld oder ein Eingabefeld auswählen.

Zu welchem Element das neue Element hinzugefügt wird, erkennen Sie an dem markierten Eintrag im Listenfeld **Member**. Da die Menüleiste in unserem Beispiel noch ganz leer ist, wird hier lediglich die Menüleiste an sich angezeigt – nämlich das Steuerelement `menuStrip1`.

Fügen Sie jetzt bitte einen neuen Menüeintrag ein. Klicken Sie dazu auf die Schaltfläche **Hinzufügen**.

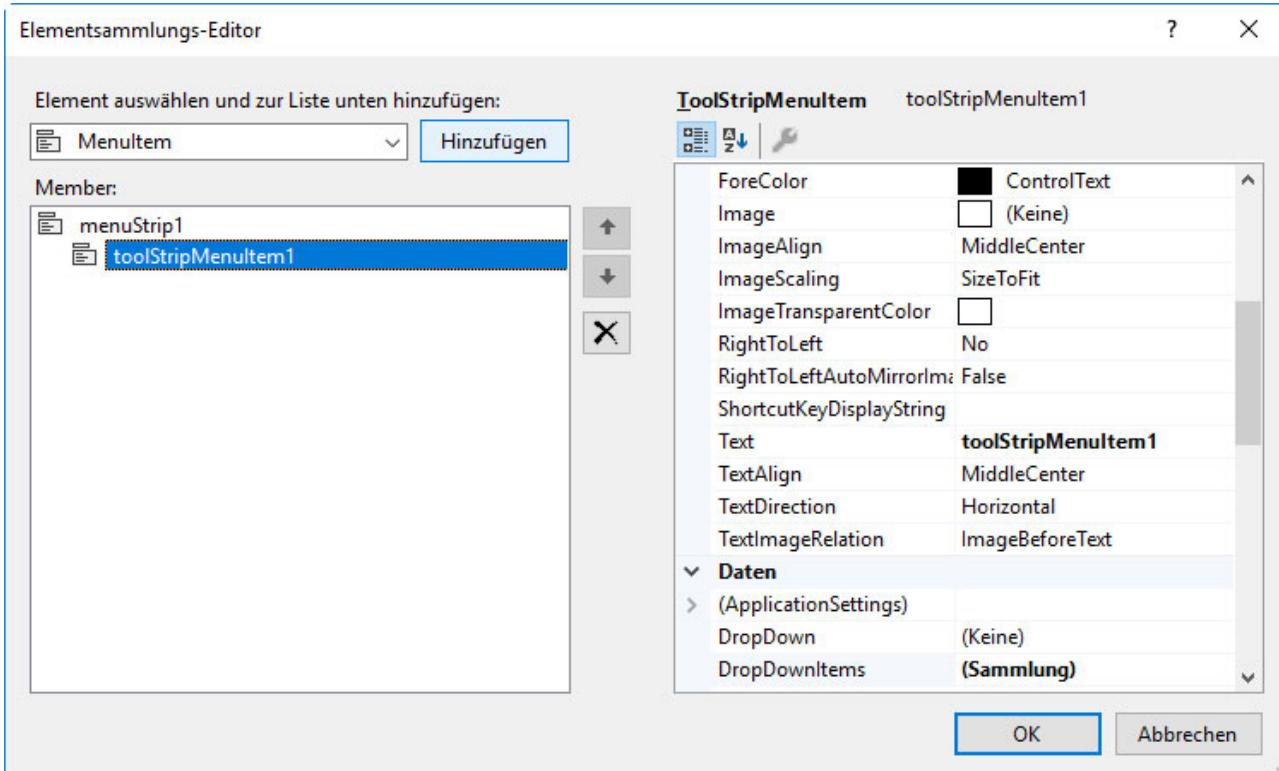


Abb. 2.3: Der neu eingefügte Menüeintrag

Über die Eigenschaften im rechten Bereich des Fensters können Sie jetzt den Namen und auch noch weitere Einstellungen für den neuen Eintrag festlegen. Setzen Sie im ersten Schritt bitte die Eigenschaft **Text** auf **&Spiel**. Das Zeichen & markiert dabei den Buchstaben, über den das Menü später auch mit der Tastatur geöffnet werden kann.

Hinweis:

Um ein Menü über die Tastatur zu öffnen, müssen Sie immer auch die Taste **Alt** drücken. In unserem Beispiel ist die Tastenkombination zum Öffnen des Menüs **Spiel** also **Alt + S**. Die Taste **S** alleine öffnet das Menü nur dann, wenn ein Eintrag in der Menüleiste markiert ist und sämtliche Menüs geschlossen sind.

Ändern Sie anschließend die Eigenschaft **(Name)** des Eintrags so, dass Sie den Eintrag eindeutig zuordnen können – zum Beispiel in `spielToolStripMenuItem`.

Welche Einträge in einem Menü enthalten sind, legen Sie über die Eigenschaft **DropDownItems**² des Menüs fest. Dabei können Sie ebenfalls wieder den Elementsammlungs-Editor benutzen.

Überprüfen Sie noch einmal, ob der Eintrag `spielToolStripMenuItem` beziehungsweise der Eintrag `toolStripMenuItem1` im Feld **Member** markiert ist. Suchen Sie dann im rechten Bereich des Fensters nach der Eigenschaft **DropDownItems** und klicken Sie auf das Symbol ..., das rechts im Feld erscheint, wenn Sie die Einfügemarke in das Feld stellen.

2. *Drop down* bedeutet übersetzt so viel wie „herunterfallen“.

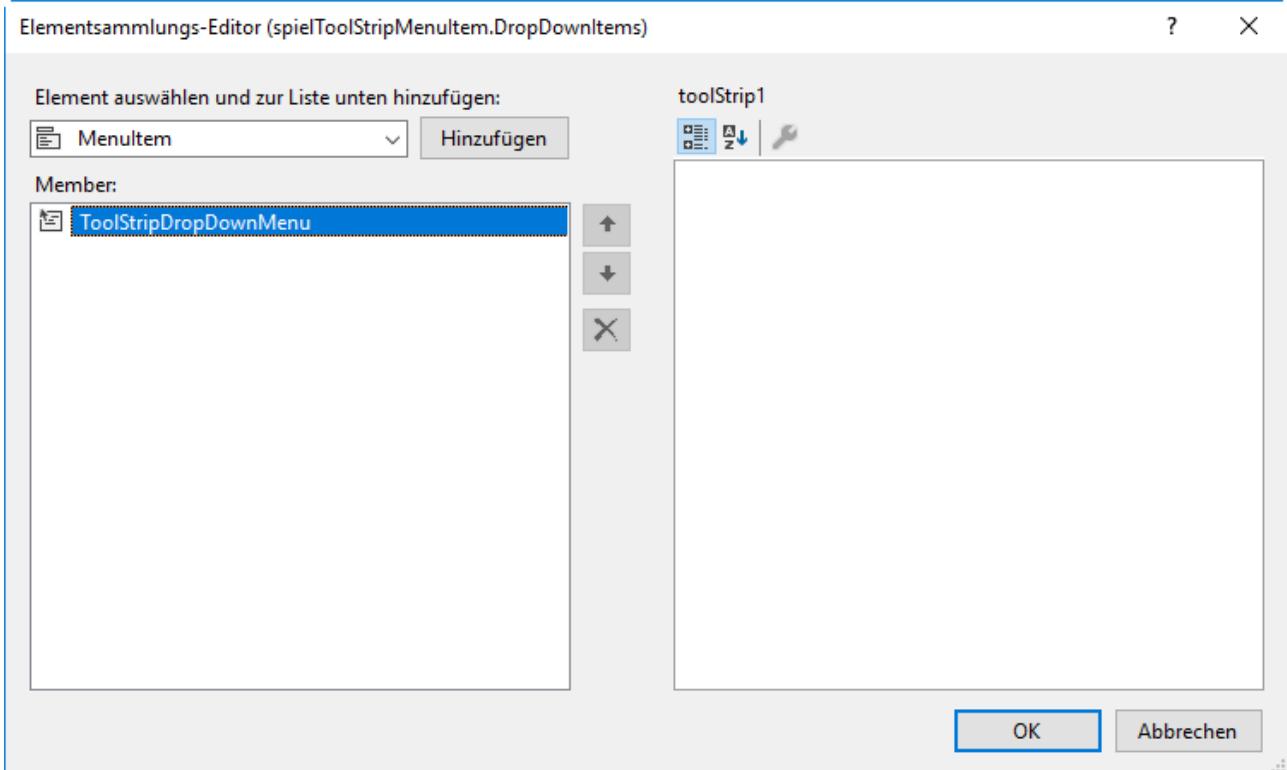


Abb. 2.4: Der Elementsammlungs-Editor für die Einträge im Menü **Spiel**

Jetzt erscheint ein zweites Fenster mit dem Elementsammlungs-Editor. In diesem Fenster legen Sie nun die eigentlichen Einträge für das Menü an. Das erfolgt im Prinzip genauso wie das Anlegen des eigentlichen Menüs.

Legen Sie einen Eintrag **Beenden** für das Menü **Spiel** an. Klicken Sie auf die Schaltfläche **Hinzufügen**. Setzen Sie anschließend die Eigenschaft **Text** des neuen Eintrags auf **&Beenden** und die Eigenschaft **(Name)** auf **beendenToolStripMenuItem**. Schließen Sie dann die beiden Fenster für den Elementsammlungs-Editor jeweils über die Schaltfläche **OK**.

In der Menüleiste im Formular sollte nun der Eintrag **Spiel** erscheinen. Der Buchstabe **S** für die Auswahl des Menüs über die Tastatur wird dabei unterstrichen dargestellt. Durch einen Mausklick auf den Menüeintrag im Designer können Sie jetzt auch das Menü aufklappen lassen.

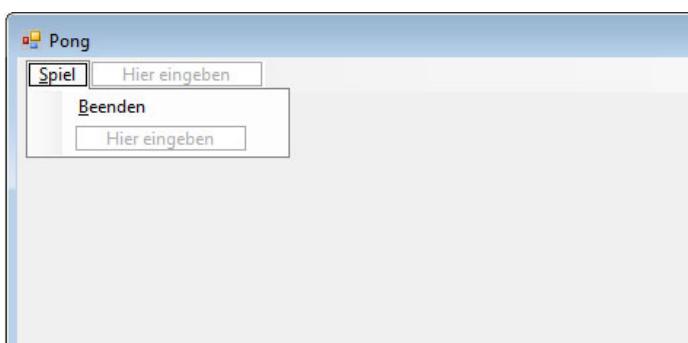


Abb. 2.5: Das geöffnete Menü **Spiel** im Designer

Nun müssen wir noch dafür sorgen, dass beim Anklicken des Eintrags **Beenden** im Menü **Spiel** die Anwendung geschlossen wird. Das erfolgt wie gewohnt über das Ereignis **Click** des Steuerelements.

Markieren Sie bitte den Menüeintrag **Beenden**. Erstellen Sie dann über das Eigenschaftenfenster eine Methode für das Ereignis **Click** und schließen Sie in dieser Methode die Anwendung über die Anweisung `Close()`.

Speichern Sie anschließend die Änderungen und testen Sie das gerade erstellte Menü. Probieren Sie dabei auch die Auswahl über die Tastatur aus.

Wenn Ihnen das Anlegen oder Bearbeiten der Menüs mit dem Elementsammlungs-Editor zu umständlich ist, können Sie die Einträge auch direkt im Formular erstellen beziehungsweise bearbeiten. Zum Erstellen zeigen Sie zunächst mit der Maus auf das Feld **Hier eingeben** in der Menüleiste beziehungsweise in einem Menü. Klicken Sie anschließend auf das Symbol hinten in dem Feld und wählen Sie über die Liste die gewünschte Art des Eintrags aus. Danach legen Sie wie gewohnt die Eigenschaften für den neuen Eintrag über das Eigenschaftenfenster fest.

Bitte beachten Sie aber, dass sich die Menüs beim direkten Bearbeiten im Formular zum Teil etwas ungewöhnlich verhalten. So wird zum Beispiel für einen neuen Eintrag in einem Menü immer auch ein leeres Untermenü erzeugt und ein neuer leerer Eintrag in das Untermenü gestellt. Diese leeren Elemente verschwinden aber, sobald Sie das Bearbeiten abschließen.

Tipp:

Das Löschen eines Menüeintrags über die Taste funktioniert nicht immer. Sie können alternativ aber auch die Funktion **Löschen** im Kontextmenü des Eintrags benutzen.

Sie können in ein Menü auch vorgefertigte Standardeinträge einfügen. Klicken Sie dazu mit der rechten Maustaste auf die Menüleiste und wählen Sie dann im Kontextmenü die Funktion **Standardelemente einfügen**. Anschließend können Sie die Standardeinträge an Ihre eigenen Anforderungen anpassen. Die Aktionen beim Anklicken müssen Sie allerdings in jedem Fall selbst erstellen. Hier nimmt Ihnen Visual Studio keine Arbeit ab.

Hinweis:

Die weiteren Menüeinträge werden wir später ergänzen und auch mit Leben füllen.

2.2 Ball und Schläger einfügen

Setzen Sie nun ein Panel in das Formular. Nennen Sie es **spielfeld** und ändern Sie die Eigenschaft **Dock** auf **Fill**. Fügen Sie dann in dieses Panel zwei weitere Panels ein. Legen Sie dazu die weiteren Panels einfach in dem bereits vorhandenen Panel ab. Nennen Sie eins der neuen Panels **schlaeger** und das andere **ball**. Die Größe spielt bei diesen beiden Panels zunächst einmal keine Rolle. Wir werden die entsprechenden Einstellungen gleich zur Laufzeit des Programms vornehmen.

Bitte beachten Sie:

Die beiden Panels für den Schläger und den Ball müssen sich im Panel für das Spielfeld befinden. Andernfalls funktioniert gleich die Steuerung nicht richtig.

Wechseln Sie anschließend in den Quelltext des Formulars und legen Sie für die Klasse `Form1` die Felder aus dem folgenden Code an. Eine kurze Beschreibung der Felder finden Sie dabei jeweils in den Kommentaren:

```
//die Felder
//eine Struktur für die Richtung des Balls
struct Spielball
{
    //wenn die Richtung true ist, geht es nach oben
    //bzw. nach rechts,
    //sonst nach unten bzw. nach links
    public bool richtungX;
    public bool richtungY;
    //für die Veränderung des Bewegungswinkels
    public int winkel;
}

//für die Zeichenfläche
Graphics zeichenflaeche;
//für das Spielfeld
Rectangle spielfeldGroesse;
int spielfeldMaxX, spielfeldMaxY, spielfeldMinX,
spielfeldMinY;
int spielfeldLinienbreite;
//für den Schläger
int schlaegerGroesse;
//für den Ball
Spielball ballPosition;
//zum Zeichnen
SolidBrush pinsel;
```

Code 2.1: Die Felder für das Spiel

Zusätzlich benötigen wir auch noch einige Methoden. Beginnen wir mit dem Konstruktor. Hier setzen wir einige Standardwerte für das Spielfeld, den Schläger und den Ball. Außerdem erzeugen wir zunächst einmal einen schwarzen Pinsel und beschaffen uns die Zeichenfläche des Panels für das Spielfeld. Der vollständige Quelltext für den Konstruktor sieht so aus:

```
public Form1()
{
    InitializeComponent();
    //die Breite der Linien
    spielfeldLinienbreite = 10;
    //die Größe des Schlägers
    schlaegerGroesse = 50;
    //erst einmal geht der Ball nach rechts und oben mit
    //dem Winkel 0
    ballPosition.richtungX = true;
```

```

ballPosition.richtungY = true;
ballPosition.winkel = 0;
//den Pinsel erzeugen
pinsel = new SolidBrush(Color.Black);
//die Zeichenfläche beschaffen
zeichenflaeche = spielfeld.CreateGraphics();
//das Spielfeld bekommt einen schwarzen Hintergrund
spielfeld.BackColor = Color.Black;
}

```

Code 2.2: Der Konstruktor

Die erste Position des Schlägers und des Balls lassen wir später in einer Methode `NeuerBall()` berechnen. Dort werden der Schläger und der Ball dann auch zum ersten Mal gezeichnet.

In einer Methode `SetzeSpielfeld()` weisen wir den Feldern `spielfeldGroesse`, `spielfeldMaxX`, `spielfeldMinX`, `spielfeldMaxY` und `spielfeldMinY` die Grenzen des Spielfelds zu. Das Feld `spielfeldGroesse` enthält dabei das vollständige Spielfeld, bei den anderen vier Feldern berücksichtigen wir die Breite der Linien, die den Rahmen für das Spielfeld bilden. Diese vier Felder liefern also die „tatsächlichen“ Dimensionen des Spielfelds. Grundlage der Berechnung ist die Größe des Client-Bereichs des Panels `spielfeld`. Diesen Wert erhalten wir über den Ausdruck `spielfeld.ClientRectangle`.

Die vollständige Methode `SetzeSpielfeld()` sieht so aus:

```

void SetzeSpielfeld()
{
    spielfeldGroesse = spielfeld.ClientRectangle;
    //die minimalen und die maximalen Ränder festlegen
    //dabei werden die Linien berücksichtigt
    //bitte in einer Zeile eingeben
    spielfeldMaxX = spielfeldGroesse.Right -
        spielfeldLinienbreite;

    //den linken Rand verschieben wir ein Pixel nach rechts
    //bitte jeweils in einer Zeile eingeben
    spielfeldMinX = spielfeldGroesse.Left +
        spielfeldLinienbreite + 1;
    spielfeldMaxY = spielfeldGroesse.Bottom -
        spielfeldLinienbreite;
    spielfeldMinY = spielfeldGroesse.Top +
        spielfeldLinienbreite;
}

```

Code 2.3: Die Methode SetzeSpielfeld()

Jetzt kommt das Zeichnen des Spielfelds an die Reihe. Hier werden die drei Balken als Rechtecke an die Ränder gezeichnet. Die Koordinaten der Rechtecke berechnen wir dabei über die vier Felder, die die tatsächliche Dimension des Spielfelds enthalten, und die Breite der Linie. Damit können wir die Balken unabhängig von der Größe des Spielfelds immer exakt an den jeweiligen Rand zeichnen.

Damit das Spielfeld nicht ganz so trist aussieht, zeichnen wir zusätzlich noch eine graue Linie von oben nach unten in die Mitte.

Die komplette Methode `ZeichneSpielfeld()` finden Sie im Code 2.4:

```
void ZeichneSpielfeld()
{
    //die weißen Begrenzungen
    pinsel.Color = Color.White;
    //ein Rechteck oben
    //bitte jeweils in einer Zeile eingeben
    zeichenflaeche.FillRectangle(pinsel, 0, 0,
        spielfeldMaxX, spielfeldLinienbreite);
    //ein Rechteck rechts
    zeichenflaeche.FillRectangle(pinsel, spielfeldMaxX,
        0, spielfeldLinienbreite, spielfeldMaxY +
        spielfeldLinienbreite);
    //und noch eins unten
    zeichenflaeche.FillRectangle(pinsel, 0,
        spielfeldMaxY, spielfeldMaxX,
        spielfeldLinienbreite);
    //damit es nicht langweilig wird, noch eine graue
    //Linie in die Mitte
    pinsel.Color = Color.Gray;
    //bitte in einer Zeile eingeben
    zeichenflaeche.FillRectangle(pinsel, spielfeldMaxX
        / 2, spielfeldMinY, spielfeldLinienbreite,
        spielfeldMaxY - spielfeldLinienbreite);
}
```

Code 2.4: Die Methode `ZeichneSpielfeld()`

So viel zum Spielfeld.

Jetzt erstellen wir noch einige Methoden für den Ball und den Schläger. Die Methoden `ZeichneSchlaeger()` und `ZeichneBall()` sollen dabei die Position der beiden Panels auf Koordinaten setzen, die wir als Parameter übergeben. Beim Schläger verändern wir lediglich die Y-Position, da er sich nur nach oben beziehungsweise unten bewegen kann. Beim Zeichnen des Balls dagegen setzen wir sowohl eine neue X- als auch eine neue Y-Position.

Mit einer Methode `NeuerBall()` lassen wir dann den Ball und den Schläger das erste Mal zeichnen. Dabei setzen wir die Größe, die Startpositionen und die Farbe für die Panels.

Ausprogrammiert finden Sie die drei Methoden im Code 2.5:

```
//setzt die Position des Balls
void ZeichneBall(Point position)
{
    ball.Location = position;
}

//setzt die Y-Position des Schlägers
void ZeichneSchlaeger(int y)
```

```

    {
        schlaeger.Top = y;
    }

    //setzt die Einstellungen für einen neuen Ball und einen
    //neuen Schläger
void NeuerBall()
{
    //die Größe des Balls setzen
    ball.Width = 10;
    ball.Height = 10;
    //die Größe des Schlägers setzen
    schlaeger.Width = spielfeldLinienbreite;
    schlaeger.Height = schlaegerGroesse;
    //beide Panels werden weiß
    ball.BackColor = Color.White;
    schlaeger.BackColor = Color.White;
    //den Schläger positionieren
    //links an den Rand
    schlaeger.Left = 2;
    //ungefähr in die Mitte
    //bitte in einer Zeile eingeben
    ZeichneSchlaeger((spielfeldMaxY / 2) -
    (schlaegerGroesse /2));
    //der Ball kommt vor den Schläger ungefähr in die Mitte
    ZeichneBall(new Point(spielfeldMinX, spielfeldMaxY / 2));
}

```

Code 2.5: Die Methoden zum „Zeichnen“ von Ball und Schläger

2.3 Das Zeichnen des Spielfelds

Im Konstruktor setzen wir die Grenzen des Spielfelds und lassen einen neuen Ball sowie einen neuen Schläger zeichnen. Dazu rufen Sie am Ende des Konstruktors die beiden Methoden `SetzeSpielfeld()` und `NeuerBall()` auf.

Das Zeichnen des eigentlichen Spielfelds soll im Ereignis **Paint** für das Panel `spielfeld` erfolgen. Erstellen Sie die entsprechende Methode und rufen Sie dort unsere eigene Methode `ZeichneSpielfeld()` auf.

Nach diesen ganzen Vorbereitungen wird es jetzt Zeit für einen ersten Test. Speichern Sie alle Änderungen und lassen Sie das Programm ausführen. Es sollte ein Spielfeld mit einem Schläger und einem Ball erscheinen.

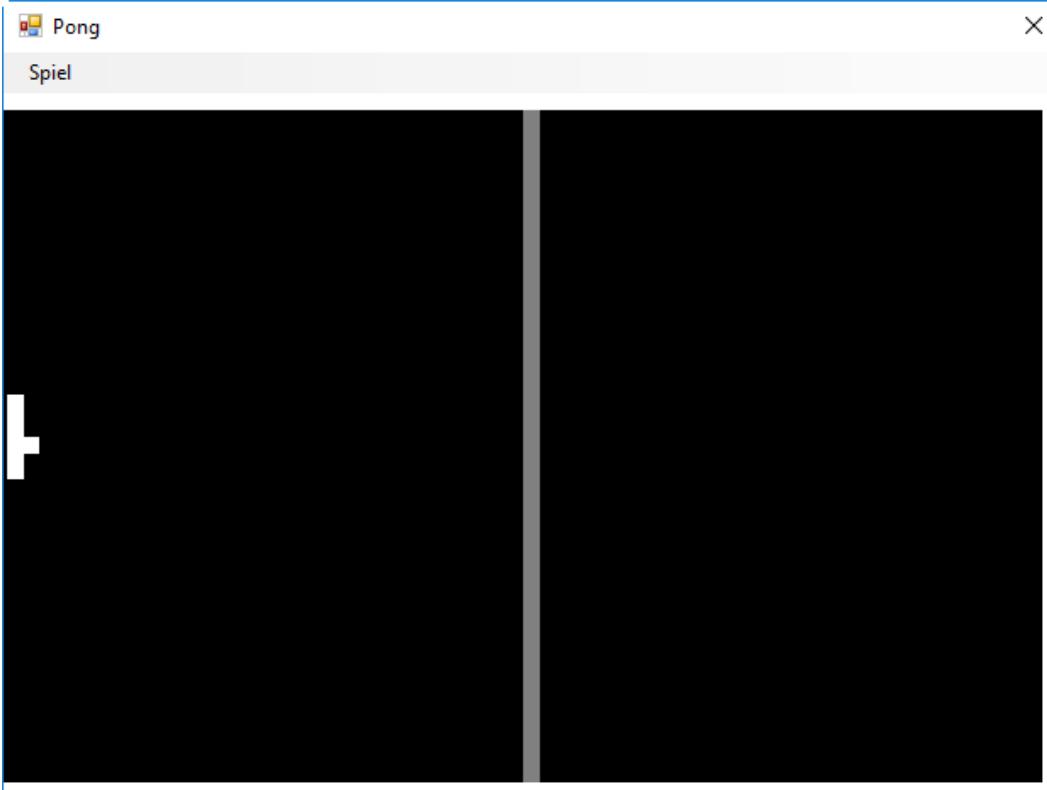


Abb. 2.6: Das Spielfeld

Im nächsten Schritt werden wir uns nun darum kümmern, dass wir den Schläger mit der Maus bewegen können.

2.4 Das Bewegen des Schlägers

Das ist nicht weiter schwierig. Wir überprüfen im Ereignis **MouseMove** des Panels **schlaeger**, ob die linke Maustaste gedrückt ist, und setzen dann den Schläger über unsere Methode **ZeichneSchlaeger()** auf die neue Position. Die dazu erforderlichen Daten liefert uns der Parameter **e** vom Typ **MouseEventArgs**. Der Ausdruck **e.Y** liefert uns die Position der Maus – allerdings relativ zum Panel **schlaeger**. Wir addieren daher auf die neue Position noch die aktuelle Position des Schlägers.

Die gedrückte Maustaste erhalten wir durch den Ausdruck **e.Button**. Ob es sich um die linke Maustaste handelt, lässt sich durch einen Vergleich mit der Konstanten **MouseButtons.Left** herausfinden. Die Methode **Schlaeger_MouseMove()** sieht dann so aus:

```
private void Schlaeger_MouseMove(object sender,
    MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
        ZeichneSchlaeger(e.Y + schlaeger.Top);
}
```

Code 2.6: Das Bewegen des Schlägers

Nun lässt sich der Schläger zwar sehr schön nach oben und unten bewegen, allerdings gibt es ein Problem: Wenn Sie die Maus mit gedrückter linker Taste sehr weit nach oben oder unten schieben, verschwindet der Schläger teilweise oder sogar komplett. Das liegt daran, dass die Y-Position der Maus ja auch kleiner oder größer werden kann als die Grenzen unseres Spielfelds.

Wir müssen daher noch überprüfen, ob sich die neue Position des Schlägers innerhalb des Spielfelds befindet. Dazu erweitern wir die Methode `ZeichneSchlaeger()` um eine `if`-Konstruktion:

```
void ZeichneSchlaeger(int y)
{
    //befindet sich der Schläger im Spielfeld?
    //bitte in einer Zeile eingeben
    if (((y + schlaegerGroesse) < spielfeldMaxY) &&
        (y > spielfeldMinY))
        schlaeger.Top = y;
}
```

Code 2.7: Abfrage der aktuellen Position des Schlägers in der Methode `ZeichneSchlaeger()`

Die erste Bedingung `((y + schlaegerGroesse) < spielfeldMaxY)` überprüft, ob das untere Ende des Schlägers noch im Spielfeld liegt. Die Addition von `y` und `schlaegerGroesse` ist erforderlich, da `y` für die Y-Koordinate der oberen linken Ecke des Schlägers steht. Die aktuelle Position des unteren Endes kennen wir nicht. Wir müssen sie daher durch die Addition der oberen linken Ecke und der Schlägergröße `schlaegerGroesse` erst berechnen.

Mit der zweiten Bedingung `(y > spielfeldMinY)` prüfen wir dann, ob sich das obere Ende des Schlägers ebenfalls im Spielfeld befindet. Das ist durch einen direkten Vergleich von `y` mit dem oberen Rand des Spielfelds möglich.

Übernehmen Sie jetzt die Erweiterung aus dem vorigen Code. Probieren Sie das Programm dann noch einmal aus. Sie werden sehen, der Schläger bleibt jetzt wie gewünscht im Spielfeld.

Hinweis:

Falls der Schläger nicht korrekt positioniert wird, setzen Sie die Menüleiste im Formular über das Symbol **In den Vordergrund**  in den Vordergrund und wählen Sie danach für das Panel mit dem Spielfeld noch einmal **Fill** für die Eigenschaft **Dock**.

Wenn die Bewegung des Schlägers deutlich über der unteren Begrenzung stoppt und sich der Schläger dafür unter die Menüleiste schieben lässt, haben Sie das Panel für den Schläger nicht im Panel für das Spielfeld abgelegt. Löschen Sie in diesem Fall das Panel für den Schläger und fügen Sie es neu im Panel für das Spielfeld ein. Achten Sie dabei darauf, dass Sie denselben Namen verwenden, und ordnen Sie die Methoden für die Ereignisse neu zu.

Kommen wir nun zum Bewegen des Balls.

2.5 Das Bewegen des Balls

Anders als der Schläger, der nur beim Ereignis **MouseMove** verschoben wird, muss sich der Ball ständig über das Spielfeld bewegen. Dazu benutzen wir einen Timer, der in sehr kurzen Abständen ausgelöst wird.

Zur Erinnerung:

Über einen Timer können Sie ein Ereignis **Tick** in regelmäßigen Abständen starten.



Fügen Sie jetzt bitte einen Timer in das Formular ein. Sie finden das entsprechende Steuerelement in der Gruppe **Komponenten** der Toolbox. Die Eigenschaft **Interval** können wir zum Test zunächst auf 100 stehen lassen. Später werden wir das Intervall über die Spieleinstellung noch gezielt verändern.

Im Ereignis **Tick** des Timers verändern wir dann die X- und die Y-Koordinaten des Balls und lassen ihn anschließend neu zeichnen. So entsteht der Eindruck, dass sich der Ball über das Spielfeld bewegt.

Die X-Koordinate verändern wir bei jedem Neuzeichnen um den Wert 10. Für die Veränderung der Y-Koordinaten benutzen wir das Mitglied `winkel` aus der Struktur `ballPosition`. Im Moment hat es noch den Wert 0, den wir im Konstruktor gesetzt haben. Später werden wir den Winkel in der Methode `ZeichneBall()` verändern.

Die Bewegungsrichtung machen wir von den beiden booleschen Mitgliedern in der Struktur `ballPosition` abhängig. Sie liefern uns ja über `true` beziehungsweise `false` die Informationen, ob sich der Ball nach links oder rechts beziehungsweise nach oben oder unten bewegt.

Wenn sich der Ball nach rechts bewegt, addieren wir den Wert 10 auf die letzte bekannte Position des Balls. Bewegt sich der Ball nach links, ziehen wir 10 von der letzten bekannten Position ab.

Die Bewegung nach oben oder unten erfolgt mit einer ähnlichen Technik: Bewegt sich der Ball nach oben, muss die Y-Position kleiner werden. Bewegt sich der Ball nach unten, muss die Y-Position größer werden.

Die Methode `Tick()` des Timers muss dann so aussehen:

```
private void Timer1_Tick(object sender, EventArgs e)
{
    int neuX = 0, neuY = 0;
    //abhängig von der Bewegungsrichtung die
    //Koordinaten neu setzen
    if (ballPosition.richtungX == true)
        neuX = ball.Left + 10;
    else
        neuX = ball.Left - 10;
    if (ballPosition.richtungY == true)
        neuY = ball.Top - ballPosition.winkel;
```

```

    else
        neuY = ball.Top + ballPosition.winkel;
        //den Ball neu zeichnen
        ZeichneBall(new Point(neuX, neuY));
    }
}

```

Code 2.8: Die Methode für den Timer

Übernehmen Sie die Anweisungen aus dem vorigen Code für die Methode des Timers. Aktivieren Sie dann noch den Timer, indem Sie die Eigenschaft **Enabled** auf **True** setzen, und testen Sie das Programm.

Besonders viel wird allerdings nicht passieren. Der Ball bewegt sich zwar sehr schön nach rechts, verschwindet allerdings rechts aus dem Spielfeld. Wir müssen also noch dafür sorgen, dass er bei Kontakten mit den Rändern des Spielfelds seine Richtung ändert. Dazu ergänzen wir in der Methode `ZeichneBall()` einige Abfragen:

```

void ZeichneBall(Point position)
{
    ball.Location = position;
    //wenn der Ball rechts anstößt, ändern wir die Richtung
    if ((position.X + 10) >= spielfeldMaxX)
        ballPosition.richtungX = false;
    //stößt er unten bzw. oben an, ebenfalls
    if ((position.Y + 10) >= spielfeldMaxY)
        ballPosition.richtungY = true;
    else
        if (position.Y <= spielfeldMinY)
            ballPosition.richtungY = false;
}

```

Code 2.9: Die Abfragen auf die Spielfeldränder in der Methode `ZeichneBall()`

Mit der ersten Abfrage überprüfen wir, ob der Ball mit der rechten Begrenzung „zusammengestoßen“ ist, und drehen gegebenenfalls die Bewegungsrichtung um. Die zweite Abfrage prüft dann, ob eine Kollision mit dem unteren oder oberen Rand erfolgt ist. Auch hier wird bei einem „Zusammenstoß“ die Bewegungsrichtung umgedreht.

Hinweis:

Beim rechten und beim unteren Rand müssen wir für die Abfragen die aktuelle Position + 10 benutzen, da `position.X` und `position.Y` für die obere linke Ecke des Balls stehen.

Übernehmen Sie jetzt die Anweisungen aus dem vorigen Code, speichern Sie die Dateien und testen Sie das Programm erneut. Nun sollte der Ball beim Kontakt mit der rechten Spielfeldbegrenzung wieder zurückkommen.

2.6 Das Zurückschlagen

Jetzt müssen wir noch dafür sorgen, dass der Ball bei einem „Zusammenstoß“ mit dem Schläger ebenfalls seine Richtung ändert. Das ist ein wenig „fummelig“, da wir nicht nur überprüfen müssen, ob der Ball wieder an seiner ursprünglichen X-Position am linken

Rand angekommen ist, sondern gleichzeitig auch kontrollieren müssen, ob sich der Schläger vor dem Ball befindet. Die entsprechende Abfrage in der Methode `ZeichneBall()` ist dann auch auf den ersten Blick recht komplex:

```
//ist er wieder links, prüfen wir, ob der Schläger in
//der Nähe ist
//bitte in einer Zeile eingeben
if ((position.X == spielfeldMinX) && ((schlaeger.Top <=
position.Y) && (schlaeger.Bottom >= position.Y)))
    //die Richtung ändern
    ballPosition.richtungX = true;
```

Code 2.10: Die Abfrage auf einen „Zusammenstoß“ zwischen Ball und Schläger

Der erste Ausdruck `(position.X == spielfeldMinX)` überprüft, ob der Ball wieder links angekommen ist. Der zweite Ausdruck `((schlaeger.Top <= position.Y) && (schlaeger.Bottom >= position.Y))` kontrolliert, ob sich die aktuelle Y-Position des Balls auf derselben Höhe wie ein Teil des Schlägers befindet.

Tipp:

Wenn Ihnen der zweite Ausdruck Kopfschmerzen macht, zeichnen Sie sich einfach ein Koordinatensystem mit dem Ball und dem Schläger. Markieren Sie dabei die Positionen jeweils durch die Ausdrücke aus dem vorigen Code. Sie werden sehen: Die Abfragen sind eigentlich gar nicht so kompliziert. Denken Sie bei der Skizze bitte daran, dass der Punkt 0,0 oben links im Koordinatensystem liegt.

Damit der Ball nun nicht immer nur auf einer geraden Linie von links nach rechts beziehungsweise von rechts nach links läuft, sorgen wir bei einem „Treffer“ noch dafür, dass das Mitglied `winkel` der Struktur für den Ball einen zufälligen Wert bekommt. Die erweiterte Abfrage sieht dann so aus:

```
//für die Zufallszahl
Random zufall = new Random();

...
//ist er wieder links, prüfen wir, ob der Schläger in
//der Nähe ist
//bitte in einer Zeile eingeben
if ((position.X == spielfeldMinX) && ((schlaeger.Top <=
position.Y) && (schlaeger.Bottom >= position.Y)))
{
    //die Richtung ändern
    ballPosition.richtungX = true;
    //und den Winkel
    ballPosition.winkel = zufall.Next(5);
}
```

Code 2.11: Das Verändern des Bewegungswinkels

Mit der Anweisung

```
Random zufall = new Random();
```

erzeugen wir zuerst eine neue Instanz der Klasse `Random`. Danach rufen Sie für diese Instanz die Methode `Next()` auf und übergeben die gewünschte obere Grenze als Argument. Die Zahlen werden im Bereich von 0 bis obere Grenze – 1 ermittelt – in unserem Beispiel also im Bereich 0 bis 4.

Da sich Ball und Schläger beim Start des Programms auf derselben Höhe befinden, ist die Abfrage

```
if ((position.X == spielfeldMinX) && ((schlaeger.Top <= position.Y) && (schlaeger.Bottom >= position.Y)))
```

bereits beim ersten Zeichnen des Balls wahr. Damit verändert der Ball wahrscheinlich direkt bei der ersten Bewegung seinen Winkel.

Im letzten Schritt sorgen wir noch dafür, dass der Ball wieder im Spielfeld erscheint, wenn der Spieler ihn verpasst hat. Dazu ergänzen wir eine weitere Abfrage, die überprüft, ob die aktuelle Position des Balls kleiner ist als seine X-Position direkt nach dem Start. Wenn das der Fall ist, legen wir eine kurze Pause ein und lassen den Ball anschließend mit der X-Position für den Start wieder neu zeichnen. Außerdem setzen wir auch hier die Bewegungsrichtung für die X-Koordinate wieder auf `true`.

Die vollständige Methode `ZeichneBall()` der Klasse `Form1` mit den Abfragen für die Ränder und den Schläger sieht jetzt so aus:

```
void ZeichneBall(Point position)
{
    //für die Zufallszahl
    Random zufall = new Random();
    ball.Location = position;
    //wenn der Ball rechts anstößt, ändern wir die Richtung
    if ((position.X + 10) >= spielfeldMaxX)
        ballPosition.richtungX = false;
    //stößt er unten bzw. oben an, ebenfalls
    if ((position.Y + 10) >= spielfeldMaxY)
        ballPosition.richtungY = true;
    else
        if (position.Y <= spielfeldMinY)
            ballPosition.richtungY = false;
    //ist er wieder links, prüfen wir, ob der Schläger
    //in der Nähe ist
    //bitte in einer Zeile eingeben
    if ((position.X == spielfeldMinX) &&
        (schlaeger.Top <= position.Y) && (schlaeger.Bottom
        >= position.Y)))
    {
        //die Richtung ändern
        ballPosition.richtungX = true;
        //und den Winkel
        ballPosition.winkel = zufall.Next(5);
    }
}
```

```
//ist der Ball hinter dem Schläger?
if (position.X < spielfeldMinX)
{
    //eine kurze Pause einlegen
    System.Threading.Thread.Sleep(1000);
    //und alles von vorne
    ZeichneBall(new Point(spielfeldMinX, position.Y));
    ballPosition.richtungX = true;
}
}
```

Code 2.12: Die vollständige Methode ZeichneBall()**Hinweis:**

Bei Kollisionen mit den Rändern kann es passieren, dass der Ball in die Spielfeldbegrenzungen hineingezeichnet wird, bevor er seine Richtung ändert. Das liegt daran, dass wir den Ball beim Neuzeichnen nicht exakt an den Rändern positionieren. Wenn sich der Ball später schneller bewegt, wird dieser Effekt aber kaum noch zu sehen sein. Wir nehmen diesen kleinen Makel daher bewusst in Kauf.

Falls Sie den Ball bei Kollisionen mit den Rändern exakt auf den Rand positionieren wollen, denken Sie bitte daran, dass wir beim „Zusammenstoß“ mit dem Schläger auf die genaue X-Position `spielfeldMinX` überprüfen. Wenn Sie den Ball also bei einer Kollision mit der hinteren Begrenzung exakt auf die Grenze positionieren, müssen Sie dafür sorgen, dass er bei der anschließenden Bewegung in die andere Richtung auch wieder bei der Position `spielfeldMinX` ankommt. Bei unserer Konstruktion müssen wir uns darum nicht weiter kümmern, da wir ja die X-Koordinate immer nur um den festen Wert 10 verändern. Der Ball landet also in jedem Fall immer wieder an der X-Koordinate `spielfeldMinX` – seiner ursprünglichen Startposition.

Den zufälligen Wert für die Veränderung des Winkels werden wir später noch vom Schwierigkeitsgrad für das Spiel abhängig machen.

Speichern Sie jetzt alle Änderungen und riskieren Sie eine Runde „Pong“. Dabei können Sie auch den zufälligen Wert für den Winkel aus einem größeren Zahlenraum ermitteln lassen. Je größer die Veränderung für die Y-Koordinate ist, desto unsauberer wird allerdings auch der „Zusammenstoß“ mit dem oberen und dem unteren Rand. Unter Umständen ist der Ball sogar für einen kurzen Moment gar nicht zu sehen. Das liegt ebenfalls daran, dass wir den Ball bei einem „Zusammenstoß“ nicht wieder exakt an dieser Position zeichnen lassen.

Wenn Sie sehr große Werte für die Veränderung der Y-Koordinate benutzen – zum Beispiel `zufall.Next(1000)`, funktioniert das ganze Spiel nicht mehr richtig. Der Ball läuft dann unter Umständen nur noch gerade hin und her – allerdings mit starkem Flackern. Das liegt daran, dass der Ball sehr schnell aus dem eigentlichen Spielfeld gerät und dort nicht gezeichnet wird.

Im nächsten Kapitel werden wir uns an die Funktionen zum Starten und Anhalten des Spiels machen.

Zusammenfassung

Über die Eigenschaft **StartPosition** können Sie festlegen, wie ein Formular bei der ersten Anzeige auf dem Bildschirm positioniert wird.

Einfache grafische Animationen können Sie zum Beispiel durch Panels erstellen, die Sie durch das Formular bewegen.

Mit der Klasse `Random` erzeugen Sie Zufallszahlen.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 2.1 Welchen Wert muss die Eigenschaft **StartPosition** erhalten, damit ein Formular beim ersten Öffnen auf dem Desktop zentriert wird?

- 2.2 Sie wollen eine Grafik, die als Datei gespeichert ist, durch ein Formular bewegen. Wie lässt sich diese Aufgabe möglichst einfach erledigen?

3 Starten und Anhalten

In diesem Kapitel werden wir die Methoden zum Starten und Anhalten des Spiels programmieren.

3.1 Vorüberlegungen

Grundsätzlich müssen wir beim Starten und Anhalten fünf verschiedene Aktionen unterscheiden können:

- 1) Das Spiel wird das erste Mal neu gestartet.
- 2) Bei einem laufenden Spiel soll ein neues Spiel gestartet werden.
- 3) Das Spiel wird unterbrochen.
- 4) Das Spiel wird nach einer Unterbrechung fortgesetzt.
- 5) Die Spielzeit ist abgelaufen.

Die ersten vier Aktionen starten wir über das Menü **Spiel**. Dazu erstellen wir einen Eintrag **Neues Spiel** und einen Eintrag **Pause**. Der Eintrag **Pause** soll dabei wie ein Schalter arbeiten: Wenn das Spiel läuft, wird es angehalten. Ist das Spiel dagegen angehalten, wird es fortgesetzt.

Über den Eintrag **Neues Spiel** soll nach einer Abfrage ein neues Spiel gestartet werden können. Dieser Neustart soll nicht nur direkt nach dem Aufruf des Programms möglich sein, sondern auch während eines laufenden Spiels. Damit das Spiel dabei nicht im Hintergrund weiterläuft, halten wir es zunächst einmal über die Pausenfunktion an.

Um ein Spiel nach dem Ablauf der Spielzeit von zwei Minuten automatisch zu beenden, benötigen wir noch einen zweiten Timer. Dieser Timer wird nach 120 000 Millisekunden – also nach 120 Sekunden oder zwei Minuten – ausgelöst.

Um das Spiel anzuhalten und wieder fortzusetzen, müssen wir auch noch einen dritten Timer einsetzen. Dieser Timer wird jede Sekunde ausgelöst und „zählt“ mit, wie viele Sekunden der Spielzeit bereits verstrichen sind. Denn beim Unterbrechen des Spiels müssen wir ja den Timer für die eigentliche Spielzeit anhalten. Dabei geht dann auch die bereits abgelaufene Zeit verloren. Wenn wir den Timer für die Spielzeit beim Fortsetzen einfach neu starten, würde er ja wieder von vorne zählen – also die Spielzeit wieder auf zwei Minuten setzen. Daher berechnen wir beim Fortsetzen des Spiels ein neues Intervall für den Timer der Spielzeit. Diesen neuen Wert erhalten wir, indem wir die bereits verstrichene Zeit von der ursprünglichen Spielzeit abziehen.

Das ganze Verfahren mag beim Durchlesen etwas kompliziert wirken. Allerdings lässt es sich nicht anders bewerkstelligen. Wenn Sie einen Timer anhalten und wieder starten, beginnt er von vorne zu zählen. Sie können bei einem Timer leider keine Zwischenzeit nehmen.

Zusätzlich werden wir über den dritten Timer auch noch die verbleibende Spielzeit auf dem Bildschirm anzeigen lassen. Damit beschäftigen wir uns aber erst später.

3.2 Die Vorbereitungen

Fügen Sie jetzt bitte zwei weitere Timer in das Formular ein. Setzen Sie das Intervall für den ersten neuen Timer auf 120000 und das Intervall für den zweiten neuen Timer auf 1000.

Tipp:

Damit Sie die drei Timer besser auseinanderhalten können, sollten Sie sprechende Namen verwenden – zum Beispiel `timerBall`, `timerSpiel` und `timerSekunde`.

Vereinbaren Sie dann zwei neue Felder für die Klasse des Formulars. Das erste Feld `spielPause` vom Typ `bool` soll speichern, ob das Spiel aktuell angehalten ist oder nicht. Das zweite Feld `aktuelleSpielzeit` vom Typ `int` soll die noch verbleibende Spielzeit aufnehmen.

Setzen Sie anschließend im Konstruktor des Formulars das Feld `spielPause` auf `true` und weisen Sie dem Feld `aktuelleSpielzeit` den Wert der Eigenschaft `Interval` des Timers für die Spielzeit zu. Damit Sie sich später bei der Anzeige die Umrechnung von Millisekunden in Sekunden sparen können, teilen Sie diesen Wert am besten direkt durch 1000.

Schalten Sie dann alle drei Timer zunächst ab. Dazu können Sie zum Beispiel im Konstruktor die Eigenschaft `Enabled` der Timer auf `false` setzen.

Der vollständige Konstruktor für das Formular sollte jetzt so aussehen:

```
public Form1()
{
    InitializeComponent();
    //die Breite der Linien
    spielfeldLinienbreite = 10;
    //die Größe des Schlägers
    schlaegerGroesse = 50;
    //erst einmal geht der Ball nach rechts und oben mit
    //dem Winkel 0
    ballPosition.richtungX = true;
    ballPosition.richtungY = true;
    ballPosition.winkel = 0;
    //den Pinsel erzeugen
    pinsel = new SolidBrush(Color.Black);
    //die Zeichenfläche beschaffen
    zeichenflaeche = spielfeld.CreateGraphics();
    //das Spielfeld bekommt einen schwarzen Hintergrund
    spielfeld.BackColor = Color.Black;
    //die Grenzen für das Spielfeld setzen
    SetzeSpielfeld();
    //einen neuen Ball erstellen
    NeuerBall();
    //erst einmal ist das Spiel angehalten
    spielPause = true;
    //die Spielzeit in Sekunden setzen
    aktuelleSpielzeit = timerSpiel.Interval / 1000;
```

```
//alle drei Timer sind zunächst angehalten
timerBall.Enabled = false;
timerSpiel.Enabled = false;
timerSekunde.Enabled = false;
}
```

Code 3.1: Der erweiterte Konstruktor für das Formular

Im nächsten Schritt sorgen wir jetzt dafür, dass die verbleibende Spielzeit ausgegeben wird. Dazu erstellen wir eine neue Methode `ZeichneZeit()`, die die restliche Zeit als Argument erhält und dann über die Methode `DrawString()` der Klasse `Graphics` rechts oben in die Zeichenfläche des Spielfelds „malt“.

Damit die Ausgaben nicht übereinanderliegen, löschen wir vor einer neuen Ausgabe eine bereits vorhandene Ausgabe, indem wir ein Rechteck in der Hintergrundfarbe des Spielfelds an die Ausgabeposition zeichnen lassen.

Die Methode `ZeichneZeit()` sieht so aus:

```
void ZeichneZeit(string restzeit)
{
    //zuerst die alte Anzeige "überschreiben"
    pinsel.Color = spielfeld.BackColor;
    //bitte in einer Zeile eingeben
    zeichenflaeche.FillRectangle(pinsel, spielfeldMaxX
        - 50, spielfeldMinY + 20, 30, 20);
    //in weißer Schrift
    pinsel.Color = Color.White;
    //die Auszeichnungen für die Schrift werden beim
    //Erstellen des Spielfelds gesetzt
    //bitte in einer Zeile eingeben
    zeichenflaeche.DrawString(restzeit, schrift,
        pinsel, new Point(spielfeldMaxX - 50, spielfeldMinY
        + 20));
}
```

Code 3.2: Die Methode `ZeichneZeit()`

Über die Eigenschaft `Color` wird im vorigen Code allerdings nur die Farbe für die Textausgabe festgelegt. Da wir später noch weitere Texte im Formular ausgeben werden, setzen wir die eigentlichen Textformatierungen im Konstruktor der Klasse. In unserem Beispiel soll der Text in der Schriftart Arial in 12 Punkt und fett erscheinen.

Vereinbaren Sie bitte ein neues Feld `schrift` mit dem Typ `Font`. Weisen Sie diesem Feld dann im Konstruktor mit der folgenden Anweisung die gewünschte Formatierung zu.

```
schrift = new Font("Arial", 12, FontStyle.Bold);
```

Damit jetzt unmittelbar nach dem Start die Spielzeit angezeigt wird, rufen wir die Methode `ZeichneZeit()` im Ereignis `Paint` des Panels für das Spielfeld auf. Als Argument übergeben wir dabei den Wert des Feldes `aktuelleSpielzeit`. Da es sich dabei um einen `int`-Typ handelt, wandeln wir diesen Wert noch in den Typ `string` um.

Die entsprechende Anweisung sieht so aus:

```
ZeichneZeit(Convert.ToString(aktuelleSpielzeit));
```

Speichern Sie jetzt die Änderungen und lassen Sie die Anwendung dann ausführen. Das Formular sollte nun ungefähr so aussehen:

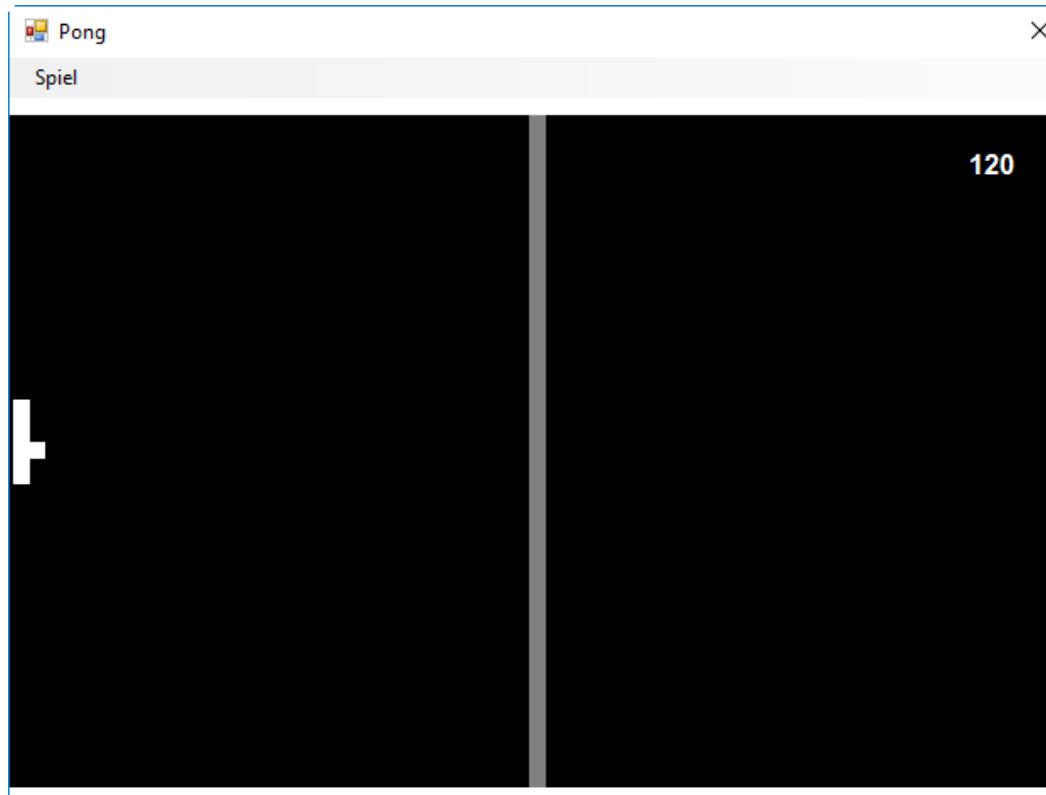


Abb. 3.1: Die Anzeige der Spielzeit oben rechts

Im letzten Schritt der Vorbereitungen lassen wir noch im Ereignis **Tick** für den Timer `timerSekunde` die Anzeige der Spielzeit aktualisieren.

Die entsprechenden Anweisungen sehen so aus:

```
private void TimerSekunde_Tick(object sender, EventArgs e)
{
    //eine Sekunde abziehen
    aktuelleSpielzeit = aktuelleSpielzeit - 1;
    //die Restzeit ausgeben
    ZeichneZeit(Convert.ToString(aktuelleSpielzeit));
}
```

Code 3.3: Die Anweisungen für den Timer `timerSekunde`

Hinweis:

Wenn Sie die Ausgabe der Restzeit testen möchten, kommentieren Sie die Anweisung im Konstruktor, die den entsprechenden Timer abschaltet, einfach aus und setzen Sie die Eigenschaft **Enabled** für den Timer auf `True`. Die Zeit sollte dann jede Sekunde heruntergezählt werden.

Die Ausgabe der Spielzeit im Ereignis **Paint** des Spielfelds ist erforderlich, da der Timer ja erst nach einer Sekunde ausgelöst wird. Wenn Sie die Zeit also nur über den Timer ausgeben lassen, erscheint die Anzeige mit einer Sekunde Verzögerung.

Damit sind die Vorbereitungen abgeschlossen und wir können uns um die Methoden zum Starten und Anhalten kümmern. Da wir auch für das Neustarten eines Spiels zunächst einmal eine Methode benötigen, die ein laufendes Spiel gegebenenfalls anhält, beginnen wir dabei mit der Methode zum Unterbrechen und Fortsetzen.

3.3 Unterbrechen und Fortsetzen

Das Unterbrechen und Fortsetzen soll so funktionieren: Beim ersten Klick auf den Menüeintrag soll das Spiel angehalten und der Eintrag im Menü mit einem Häkchen markiert werden. Beim nächsten Anklicken soll das Spiel fortgesetzt und das Häkchen wieder entfernt werden.

Die Anzeige des Häckchens lässt sich recht einfach über die Eigenschaft `Checked` eines Menüeintrags realisieren. Wenn sie den Wert `true` hat, wird das Häkchen links vor dem Eintrag angezeigt. Hat sie den Wert `false`, wird das Häkchen nicht angezeigt. Zusätzlich müssen wir beim Anklicken des Eintrags prüfen, ob das Spiel gerade läuft oder nicht, und dann die entsprechenden Aktionen zum Unterbrechen beziehungsweise Fortsetzen ausführen lassen. Dazu fragen wir den Wert des Felds `spielPause` ab.

Erstellen Sie jetzt bitte einen neuen Eintrag **Pause** im Menü **Spiel** unserer Anwendung. Geben Sie dazu den Text **&Pause** im Feld **Hier eingeben** ein und verschieben Sie den neuen Eintrag dann mit gedrückter linker Maustaste an die erste Position im Menü. Wenn Sie wollen, können Sie zwischen den Einträgen **Pause** und **Beenden** auch noch eine Trennlinie im Menü einfügen. Wählen Sie dazu den Eintrag **Separator** im Kombinationsfeld aus.

Hinweis:

Der neue Menüeintrag erhält automatisch den Namen `pauseToolStripMenuItem`. Diesen Namen werden wir auch in den folgenden Codes verwenden.

Geben Sie dann für das Ereignis **Click** des Menüeintrags die Anweisungen aus dem Code 3.4 ein.

```
private void PauseToolStripMenuItem_Click(object sender,
EventArgs e)
{
    //erst einmal prüfen wir den Status
    //läuft das Spiel?
    if (spielPause == false)
    {
        //alle Timer anhalten
        timerBall.Enabled = false;
        timerSekunde.Enabled = false;
        timerSpiel.Enabled = false;
        //die Markierung im Menü einschalten
        pauseToolStripMenuItem.Checked = true;
        //den Text in der Titelleiste ändern
        this.Text = "Pong - Das Spiel ist angehalten!";
    }
}
```

```
    spielPause = true;
}
else
{
    //das Intervall für die verbleibende Spielzeit
    //setzen
    //bitte in einer Zeile eingeben
    timerSpiel.Interval = aktuelleSpielzeit *
    1000;
    //alle Timer wieder an
    timerBall.Enabled = true;
    timerSekunde.Enabled = true;
    timerSpiel.Enabled = true;
    //die Markierung im Menü abschalten
    pauseToolStripMenuItem.Checked = false;
    //den Text in der Titelleiste ändern
    this.Text = "Pong";
    spielPause = false;
}
```

Code 3.4: Die Anweisungen für das Klicken auf den Menüeintrag **Pause**

Mit der `if`-Abfrage prüfen wir, ob `spielPause` den Wert `false` hat. Wenn das der Fall ist, läuft das Spiel gerade und muss angehalten werden. Dazu stoppen wir alle drei Timer. Danach wird das Häkchen vor dem Menüeintrag gesetzt, ein Text in der Titelleiste ausgegeben und dem Feld `spielPause` der Wert `true` zugewiesen. Damit ist das Spiel unterbrochen.

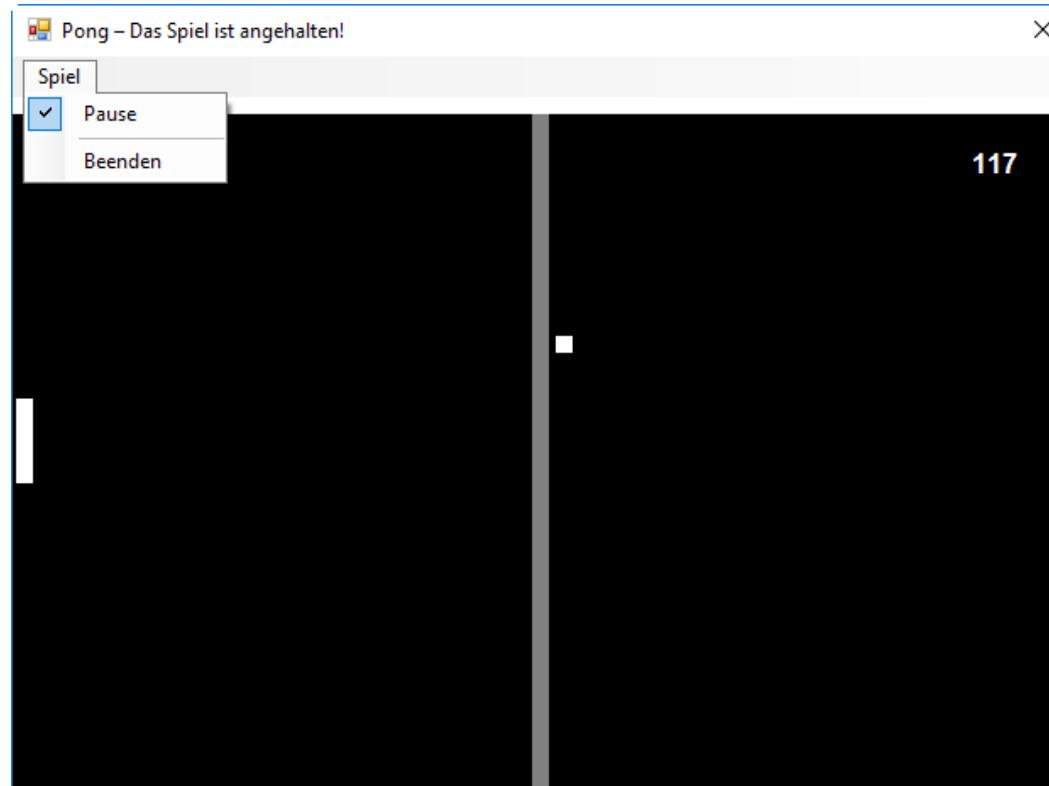


Abb. 3.2: Das unterbrochene Spiel

Falls `spielPause` dagegen den Wert `true` hat, ist das Spiel gerade unterbrochen und muss fortgesetzt werden. Dazu setzen wir im ersten Schritt den Wert des Timers für die Spielzeit auf die noch verbleibende Zeit. Da die Eigenschaft `Interval` einen Wert in Millisekunden erwartet, multiplizieren wir den Wert des Feldes `aktuelleSpielzeit` mit 1 000. Danach werden alle drei Timer wieder aktiviert, der Text in der Titelleiste überschrieben und `spielPause` auf `false` gesetzt. Das Spiel läuft also wieder.

Hinweis:

Damit der Schläger bei einer Spielpause nicht bewegt werden kann, müssen Sie im Ereignis `MouseMove` für den Schläger noch eine Abfrage ergänzen, ob das Spiel an gehalten ist oder nicht. Dazu können Sie noch vor der Abfrage der linken Maustaste überprüfen, ob `spielPause` den Wert `true` hat, und die Methode dann direkt wieder verlassen. Die entsprechenden Anweisungen sehen so aus:

```
if (spielPause == true)
    return;
```

So viel zum Unterbrechen eines Spiels. Kommen wir nun zum Neustart.

3.4 Neu starten

Beim Starten eines Spiels müssen wir zunächst prüfen, ob gerade ein aktuelles Spiel läuft. Wenn das der Fall ist, wird es über die Methode `PauseToolStripMenuItem_Click()` angehalten. Danach fragen wir in einem Dialog ab, ob tatsächlich ein neues Spiel gestartet werden soll. Klickt der Anwender hier auf die Schaltfläche **Ja**, werden alle Einstellungen wieder auf die Startwerte gesetzt und das komplette Spielfeld neu gezeichnet.

Danach rufen wir die Methode `PauseToolStripMenuItem_Click()` noch einmal auf. Dieser Aufruf erfolgt in jedem Fall – also auch, wenn der Anwender kein neues Spiel starten möchte. Dann führt die Methode dazu, dass das vorher unterbrochene Spiel fortgesetzt wird.

Wenn kein Spiel läuft, soll beim Anklicken des Eintrags **Neues Spiel** im Menü **Spiel** direkt der Dialog zum Neustart erscheinen. Klickt der Anwender hier auf die Schaltfläche **Nein**, passiert nichts weiter. Beim Anklicken der Schaltfläche **Ja** werden ebenfalls die Einstellungen für das Spiel auf die Startwerte gesetzt und dann das Spiel über die Methode `PauseToolStripMenuItem_Click()` gestartet.

Da wir den Dialog zum Neustarten des Spiels mehrfach unter verschiedenen Bedingungen einsetzen müssen, erstellen wir zunächst einmal eine Methode `NeuesSpiel()`. Diese Methode erzeugt den Dialog, setzt gegebenenfalls die Einstellungen für ein neues Spiel und liefert uns zurück, ob der Anwender auf die Schaltfläche **Ja** oder **Nein** geklickt hat. Die Anweisungen der Methode könnten so aussehen:

```
//erzeugt einen Dialog zum Neustart und liefert das
//Ergebnis zurück
bool NeuesSpiel()
{
    bool ergebnis = false;
```

```
//bitte in einer Zeile eingeben
if (MessageBox.Show("Neues Spiel starten?", "Neues
Spiel", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes)
{
    //die Spielzeit neu setzen
    aktuelleSpielzeit = 120;
    //alles neu zeichnen
    ZeichneSpielfeld();
    NeuerBall();
    ZeichneZeit(Convert.ToString(aktuelleSpielzeit));
    ergebnis = true;
}
return ergebnis;
}
```

Code 3.5: Die Methode NeuesSpiel()

Beim Anklicken der Schaltfläche Ja setzen wir die aktuelle Spielzeit wieder auf 120. Danach werden das Spielfeld, der Ball und der Schläger sowie die Zeit in das Formular gezeichnet.

Jetzt können wir uns um die eigentliche Funktion zum Starten eines neuen Spiels kümmern. Legen Sie bitte einen Eintrag **Neues Spiel** im Menü **Spiel** unserer Anwendung an. Geben Sie dann für das Ereignis **Click** des neuen Eintrags die folgenden Anweisungen ein:

```
private void NeuesSpielToolStripMenuItem_Click(object
sender, EventArgs e)
{
    //läuft ein Spiel?
    //dann erst einmal pausieren
    if (spielPause == false)
    {
        PauseToolStripMenuItem_Click(sender, e);
        //den Dialog anzeigen
        NeuesSpiel();
        //und weiter spielen
        PauseToolStripMenuItem_Click(sender, e);
    }
    //wenn kein Spiel läuft, starten wir ein neues,
    //wenn im Dialog auf Ja geklickt wurde
    else
        if (NeuesSpiel() == true)
            PauseToolStripMenuItem_Click(sender, e);
}
```

Code 3.6: Die Anweisungen für das Anklicken des Menüeintrags **Neues Spiel**

Wenn ein Spiel läuft, wird es zunächst gestoppt. Danach wird die Methode `NeuesSpiel()` aufgerufen und das Spiel fortgesetzt. Welche Einstellungen für das Fortsetzen des Spiels verwendet werden, hängt davon ab, ob der Anwender im Dialog für den Neustart **Ja** oder **Nein** angeklickt hat. Bei **Ja** werden die Werte für ein neues Spiel benutzt, bei **Nein** dagegen die Werte aus dem laufenden Spiel.

Läuft dagegen kein Spiel, wird nur dann ein neues Spiel gestartet, wenn der Anwender im Dialog für das neue Spiel auf **Ja** klickt. Andernfalls geschieht nichts.

3.5 Das Spielende

Jetzt fehlt nur noch die Methode zum Beenden des Spiels, wenn die Spielzeit abgelaufen ist. Dazu halten wir im Ereignis **Tick** des Timers für die Spielzeit zunächst das Spiel über die Methode `PauseToolStripMenuItem_Click()` an, geben eine Meldung aus und fragen dann über unsere Methode `NeuesSpiel()`, ob der Anwender ein neues Spiel starten möchte. Wenn das der Fall ist, wird das Spiel mit den Starteinstellungen fortgesetzt. Andernfalls wird es über die Methode `BeendenToolStripMenuItem_Click()` beendet.

Die entsprechenden Anweisungen könnten so aussehen:

```
private void TimerSpiel_Tick(object sender, EventArgs e)
{
    //das Spiel anhalten
    pauseToolStripMenuItem_Click(sender, e);
    //eine Meldung anzeigen
    //bitte in einer Zeile eingeben
    MessageBox.Show("Die Zeit ist um", "Spielende",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
    //Abfrage, ob ein neues Spiel gestartet werden soll
    if (NeuesSpiel() == true)
        //das Spiel fortsetzen
        pauseToolStripMenuItem_Click(sender, e);
    else
        //sonst beenden
        beendenToolStripMenuItem_Click(sender, e);
}
```

Code 3.7: Das Beenden des Spiels

So weit funktioniert das Spiel bisher ganz ordentlich. Sie können es anhalten, fortsetzen und auch neu starten.

Wenn ein Anwender allerdings direkt nach dem Start des Spiels die Funktion **Pause** im Menü **Spiel** anklickt, beginnt ein neues Spiel. Wir müssen also dafür sorgen, dass die Funktion **Pause** nur dann aufgerufen werden kann, wenn gerade ein Spiel läuft.

Dazu deaktivieren wir den Menüeintrag direkt nach dem Start und schalten ihn erst dann ein, wenn ein neues Spiel gestartet wurde. Ergänzen Sie dazu bitte im Konstruktor des Formulars die Anweisung

```
pauseToolStripMenuItem.Enabled = false;
```

Die Anweisung zum Setzen der aktuellen Spielzeit können Sie jetzt übrigens aus dem Konstruktor löschen. Die Zeit wird ja ebenfalls beim Neustart eingestellt.

Ergänzen Sie dann in der Methode `NeuesSpiel()` beim Anklicken der Schaltfläche **Ja** die Anweisung

```
pauseToolStripMenuItem.Enabled = true;
```

Damit wird der Menüeintrag **Pause** wieder aktiviert.

Nun bleibt nur noch ein kleines Problemchen: Wenn das Fenster unserer Anwendung durch eine andere Anwendung verdeckt wird, läuft das Spiel im Hintergrund weiter. Dieses Problem lösen wir aber recht rabiat. Wir sorgen ähnlich wie beim Bildbetrachter dafür, dass das Fenster immer im Vordergrund bleibt. Dazu setzen Sie die Eigenschaft **TopMost** des Formulars auf `True`. Sie finden diese Eigenschaft in der Gruppe **Fensterstil** im Eigenschaftenfenster.

Damit sind auch die Funktionen zum Starten und Anhalten des Spiels komplett. Im nächsten Kapitel werden wir uns um die Bestenliste kümmern.

Zusammenfassung

Wenn Sie einen Timer unterbrechen und später fortsetzen wollen, müssen Sie die bereits abgelaufene Zeit zwischenspeichern.

Über die Eigenschaft `Checked` können Sie ein Häkchen vor einem Menüeintrag ein- beziehungsweise ausblenden.

Aufgaben zur Selbstüberprüfung

- 3.1 Worauf müssen Sie achten, wenn Sie unterschiedliche Texte mit der Methode `DrawString()` hintereinander an derselben Position ausgeben lassen?

- 3.2 Wie können Sie dafür sorgen, dass ein Formular ständig im Vordergrund angezeigt wird?

- 3.3 Sie wollen über die Methode `DrawString()` einen fett formatierten Text ausgeben lassen. Wie setzen Sie diese fette Darstellung?

- 3.4 Sie wollen einen Timer in einer Anwendung unterbrechen und wieder fortsetzen. Dabei soll der Timer aber nicht neu gestartet werden, sondern die bereits verstrichene Zeit soll berücksichtigt werden. Beschreiben Sie bitte, wie Sie vorgehen.

4 Die Bestenliste

In diesem Kapitel programmieren wir die Bestenliste für das Spiel. Dazu sind nur ein paar Anweisungen erforderlich, die aber ein wenig anspruchsvoller sind.

Bevor wir mit der Programmierung anfangen, zunächst wieder einige Vorüberlegungen.

4.1 Vorüberlegungen

Nahezu die gesamte Logik der Bestenliste lagern wir in eine eigene Klasse aus, die grundsätzlich auch von anderen Anwendungen genutzt werden können soll. Die Klasse für die Bestenliste soll dabei folgende Aufgaben übernehmen:

- das Verändern des Punktestands für das aktuelle Spiel,
- das Zurücksetzen der Punkte beim Neustart eines Spiels,
- das Verwalten einer Liste mit Punkten und Namen und
- die Ausgabe der Liste.

Für das Verändern des Punktestands gelten in der Standardeinstellung folgende Regeln:

- Wenn der Spieler den Ball mit dem Schläger zurückspielt, erhält er einen Punkt.
- Verpasst er den Ball, bekommt er fünf Punkte abgezogen.

Die Ausgabe des neuen Punktestands führen wir direkt nach einer Änderung durch. Damit wir die Punktanzeige korrekt im Spielfeld positionieren können, erfolgt die Ausgabe über eine Methode der Klasse `Form1`.

Auch die weitere Verarbeitung der Bestenliste übernimmt die eigentliche Anwendung – also die Klasse für das Formular. Hier überprüfen wir später, ob die erzielten Punkte für einen neuen Eintrag in der Liste ausreichen, und stellen eine Funktion für die Anzeige der Liste zur Verfügung. Dazu werden jeweils entsprechende Methoden aus der Klasse für die Bestenliste aufgerufen.

Der kniffligste Teil ist dabei die Überprüfung, ob die erzielten Punkte für einen neuen Eintrag in der Bestenliste ausreichen. Dazu müssen wir die Einträge in der Liste überprüfen und gegebenenfalls austauschen. Um uns das Leben zu erleichtern, benutzen wir eine Liste mit absteigend sortierten Einträgen. Der Eintrag mit der höchsten Punktzahl steht also ganz am Anfang der Liste und der Eintrag mit der niedrigsten Punktzahl am Ende. Falls die aktuelle Punktzahl größer ist als beim Eintrag am Ende der Liste, überschreiben wir den letzten Eintrag mit den aktuellen Daten und sortieren die Liste danach neu.

So viel zum grundsätzlichen Vorgehen. Beginnen wir jetzt mit der praktischen Umsetzung.

4.2 Die Klasse Score

Zunächst einmal erstellen wir eine neue Klasse `Score` für die Verwaltung der Bestenliste und fügen dort die einfacheren Methoden ein.

Legen Sie bitte über die Funktion **Projekt/Klasse hinzufügen ...** eine neue Klasse an. Im Fenster **Neues Element hinzufügen – Pong** geben Sie in das Feld **Name:** bitte `Score` ein.

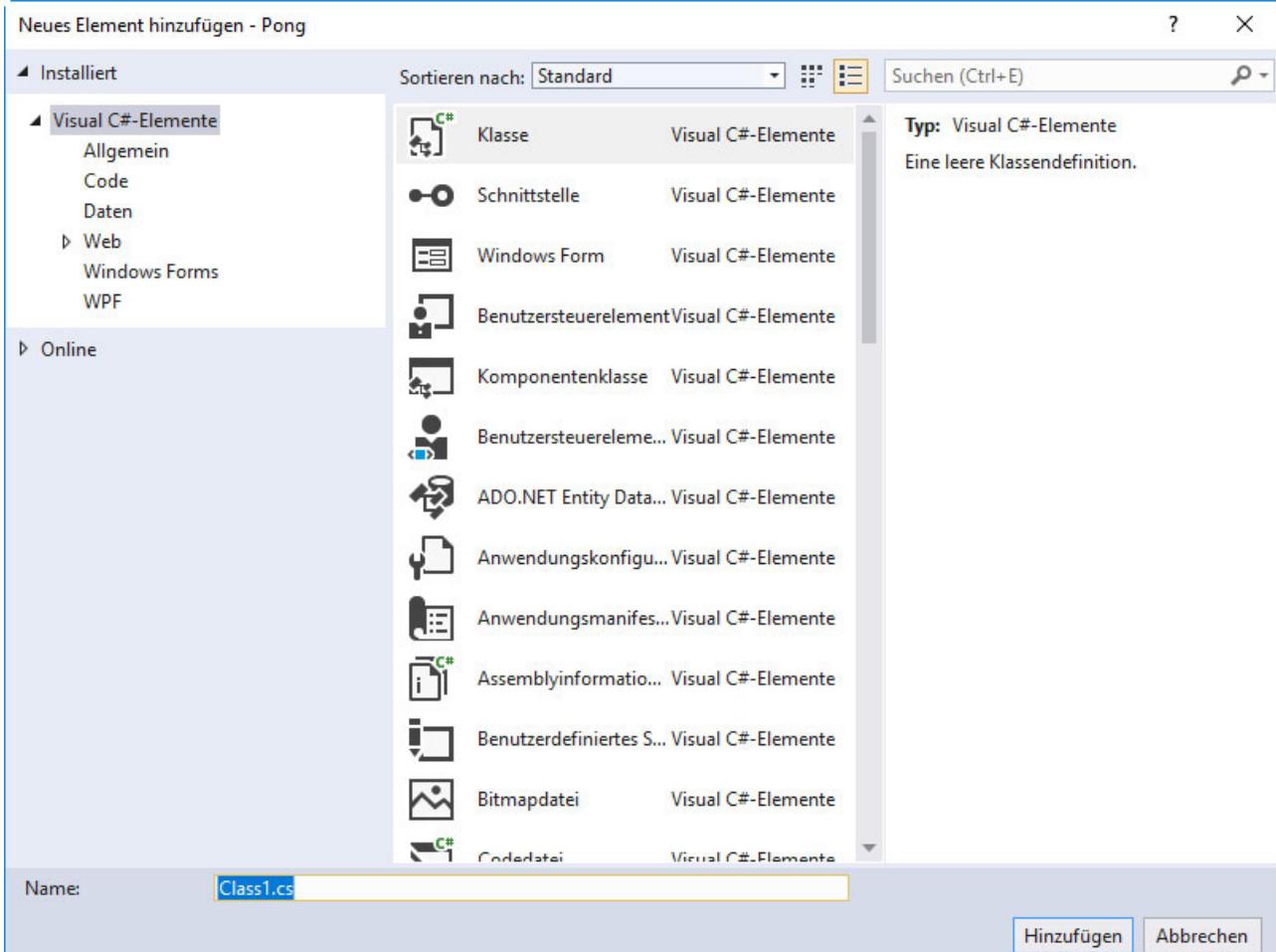


Abb. 4.1: Das Anlegen einer neuen Klasse

Ergänzen Sie dann für die Klasse die Felder und Methoden aus dem Code 4.1.

```
using System;

namespace Pong
{
    class Score
    {
        //die Felder
        int punkte;

        //die Methoden
        //der Konstruktor
        public Score()
```

```

{
    punkte = 0;
}

//zum Verändern der Punkte
public int VeraenderePunkte(int anzahl)
{
    punkte = punkte + anzahl;
    return punkte;
}

//zum Zurücksetzen der Punkte
public void LoeschePunkte()
{
    punkte = 0;
}
}

```

Code 4.1: Die Vereinbarung der Klasse Score

Vereinbaren Sie dann in der Klasse `Form1` ein Feld vom Typ `Score` mit dem Namen `spielpunkte` und erzeugen Sie im Konstruktor der Klasse `Form1` über dieses Feld eine neue Instanz der Klasse `Score`. Dazu verwenden Sie wie gewohnt den Operator `new`.

4.3 Die Ausgabe der Punkte

Im nächsten Schritt können wir uns jetzt um die Ausgabe der Punktzahl kümmern.

Die eigentliche Ausgabe ist dabei recht einfach.

Die entsprechende Methode für die Klasse `Form1` könnte zum Beispiel so aussehen:

```

void ZeichnePunkte(string punkte)
{
    //zuerst die alte Anzeige überschreiben
    pinsel.Color = spielfeld.BackColor;
    //bitte in einer Zeile eingeben
    zeichenflaeche.FillRectangle(pinsel, spielfeldMaxX
        - 50, spielfeldMinY + 40, 30, 20);
    //in weißer Schrift
    pinsel.Color = Color.White;
    //die Einstellungen für die Schrift werden beim
    //Erstellen des Spielfelds gesetzt
    //bitte in einer Zeile eingeben
    zeichenflaeche.DrawString(punkte, schrift, pinsel,
        new Point(spielfeldMaxX - 50, spielfeldMinY + 40));
}

```

Code 4.2: Die Methode `ZeichnePunkte()`

Genau wie bei der Ausgabe der Zeit überschreiben wir auch hier erst wieder den Ausgabebereich mit einem schwarzen Rechteck, damit die vorherige Ausgabe verschwindet.

Der Aufruf der Methode `ZeichnePunkte()` erfolgt dann in der Methode `ZeichneBall()`. Wenn der Ball den Schläger berührt, erhöhen wir die Punktzahl um 1 und lassen die Punkte ausgeben. Läuft der Ball dagegen am Schläger vorbei, ziehen wir fünf Punkte ab.

```
...
//ist er wieder links, prüfen wir, ob der Schläger in
//der Nähe ist
if ((position.X == spielfeldMinX) && ((schlaeger.Top <=
position.Y) && (schlaeger.Bottom >= position.Y)))
{
    if (ballPosition.richtungX == false)
        //einen Punkt dazu und die Punkte ausgeben
        //bitte in einer Zeile eingeben
        ZeichnePunkte(Convert.ToString
        (spielpunkte.VeraenderePunkte(1)));
    //die Richtung ändern
    ballPosition.richtungX = true;
    //und den Winkel
    ballPosition.winkel = zufall.Next(5);
}
//ist der Ball hinter dem Schläger?
if (position.X < spielfeldMinX)
{
    //fünf Punkte abziehen und die Punkte ausgeben
    //bitte in einer Zeile eingeben
    ZeichnePunkte(Convert.ToString
    (spielpunkte.VeraenderePunkte(-5)));
}
...
```

Code 4.3: Das Verändern der Punktzahl

Ergänzen Sie bitte die fett markierten Anweisungen aus dem Code 4.3 in der Methode `ZeichneBall()`. Die `if`-Abfrage

```
if (ballPosition.richtungX == false)
```

vor dem Erhöhen der Punkte sorgt dabei dafür, dass die Punkte nur dann hochgezählt werden, wenn sich der Ball in Richtung Schläger bewegt hat. Andernfalls würden bei einem Kontakt die Punkte doppelt gezählt, da der Ball ja erst nach der Berührung seine Richtung ändert und dabei auch noch einmal den Schläger passiert.

Speichern Sie dann die Änderungen und führen Sie einen Test durch. Beim ersten Zurücksschlagen sollten oben rechts im Spielfeld unter der Zeit jetzt auch die Punkte erscheinen.

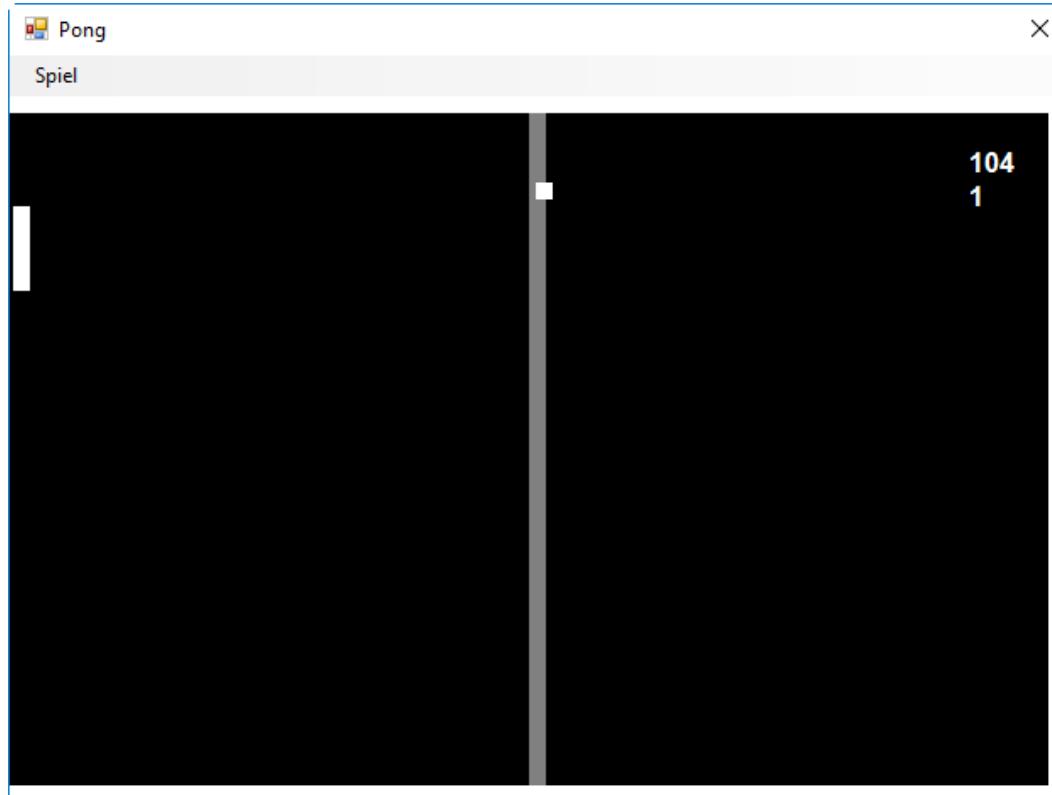


Abb. 4.2: Die Punktanzeige

Damit die Punkte direkt nach dem Start eines neuen Spiels angezeigt werden und auch wieder zurückgesetzt werden, ergänzen Sie jetzt noch in der Methode `NeuesSpiel()` der Klasse `Form1` nach den Anweisungen zur Ausgabe der Spielzeit die beiden Anweisungen

```
spielpunkte.LoeschePunkte();
ZeichnePunkte("0");
```

Damit ist auch die Ausgabe der Punkte vollständig und wir können uns um die eigentliche Liste kümmern.

4.4 Die Verwaltung der Liste

Eine Möglichkeit wäre es, eine Struktur mit den Punkten und einem Namen für die Einträge anzulegen, dann ein Array für diese Struktur zu erstellen und die Liste anschließend über die Methode `Array.Sort()` sortieren zu lassen. Was sich in der Theorie noch ganz brauchbar anhört, klappt in der Praxis leider nicht. Denn die Methode `Array.Sort()` weiß nicht, wonach das Array sortiert werden soll, da wir zum einen Zahlen speichern, zum anderen aber auch Zeichenketten.

Wir erstellen daher eine eigene Klasse und legen selber fest, wie die Sortierung erfolgen soll. Aus dieser Klasse bauen wir uns anschließend ein Array, das wir dann nach den Punkten sortieren lassen können.

Damit das Ganze nicht zu kompliziert wird, nehmen wir die Vereinbarung unserer Sortierklasse in die Klasse `Score` auf und beschränken uns bei der Sortierung auch nur auf die Reihenfolge, die wir für die Bestenliste benötigen – nämlich eine absteigende Sortierung, die beim größten Wert beginnt.

Die Vereinbarung einer Klasse `Liste` für die Sortierung finden Sie im Code 4.4. Was dort genau passiert, erklären wir Ihnen im Anschluss.

```
//die Klasse für die Liste
//Sie muss die Schnittstelle IComparable implementieren
class Liste : IComparable
{
    //die Felder
    int listePunkte;
    string listeName;

    //die Methoden
    //der Konstruktor
    public Liste()
    {
        //er setzt die Punkte und den Namen auf
        //Standardwerte
        listePunkte = 0;
        listeName = "Nobody";
    }

    //die Vergleichsmethode
    public int CompareTo(object objekt)
    {
        Liste tempListe = (Liste)(objekt);
        if (this.listePunkte < tempListe.listePunkte)
            return 1;
        if (this.listePunkte > tempListe.listePunkte)
            return -1;
        else
            return 0;
    }

    //die Methode zum Setzen von Einträgen
    public void SetzeEintrag(int punkte, string name)
    {
        listePunkte = punkte;
        listeName = name;
    }

    //die Methode zum Liefern der Punkte
    public int GetPunkte()
    {
        return listePunkte;
    }
}
```

```
//die Methode zum Liefern des Namens
public string GetName()
{
    return listeName;
}
```

Code 4.4: Die Klasse für die Liste

Zunächst einmal sorgen wir dafür, dass unsere Klasse `Liste` die Schnittstelle `IComparable`³ implementiert. Dadurch können wir unsere eigene Sortierung erstellen. Welche Schnittstelle die Klasse implementiert, geben Sie wie bei der Vererbung hinter einem Doppelpunkt an.



Schnittstellen sind – etwas vereinfacht ausgedrückt – Klassen, die die **Signatur** – den Namen und die Typen der Parameterliste – von bestimmten Methoden enthalten. Die Methoden selbst – also die eigentliche Funktionalität – müssen Sie in Ihren eigenen Klassen im Detail implementieren, und zwar sämtliche Methoden, die die Schnittstelle vorgibt.

Schnittstellen werden auch **Interfaces** genannt.

Anschließend folgen die Felder der Klasse und der Konstruktor. Dabei gibt es keine Besonderheiten.

Spannend wird dann die Methode `CompareTo()`. Sie erstellt die Standardmethode `IComparable.CompareTo()`, die zwei Objekte desselben Typs vergleichen kann, neu – und zwar so, dass nach der Anzahl der Punkte sortiert wird.



Bitte beachten Sie:

Der Rückgabetyp der Methode `CompareTo()` muss ein `int` sein. Der Parameter muss vom Typ `object` sein.

Die erste Anweisung der Methode

```
Liste tempListe = (Liste)(objekt);
```

wandelt zunächst einmal das übergebene Objekt wieder in ein Objekt vom Typ `Liste` um. Danach folgen dann mehrere Vergleiche, die den Rückgabewert der Methode festlegen. Wann welcher Wert geliefert werden muss, hängt von der gewünschten Reihenfolge der Sortierung ab.

Bei aufsteigender Sortierung muss ein Wert größer als 0 geliefert werden, wenn der aktuelle Wert kleiner ist als der andere. Ist der andere Wert kleiner als der erste, muss ein Wert kleiner als 0 geliefert werden. Bei absteigender Sortierung ist es genau andersherum: Ist der aktuelle Wert größer als der zweite, muss ein Wert kleiner als 0 geliefert werden, andernfalls ein Wert größer als 0. Sind die beiden Werte gleich, muss sowohl bei aufsteigender als auch bei absteigender Sortierung der Wert 0 zurückgegeben werden.

3. *Comparable* bedeutet übersetzt so viel wie „vergleichbar“.

Wenn Sie jetzt die Ausdrücke wie

```
if (this.listePunkte < tempListe.listePunkte)
```

irritieren: Als Argument erhält die Methode nicht die aktuelle Instanz, sondern die andere Instanz – also die Werte, die mit den aktuellen Werten verglichen werden soll. Welche Instanz das genau ist, spielt für Sie als Programmierer eigentlich keine Rolle, da sich die Methode später beim Sortieren automatisch so lange selbst aufruft, bis die Sortierung komplett durchgeführt wurde.

Die drei Methoden `SetzeEintrag()`, `GetPunkte()` und `GetName()` der Klasse `Liste` benötigen wir, um nicht von außen direkt auf die Felder der Klasse zugreifen zu müssen.

Hinweis:

Sie können auf die drei Methoden auch verzichten, wenn Sie die Felder für die Punkte und den Namen öffentlich vereinbaren. Damit verletzen Sie aber das Prinzip der Datenkapselung.

Nachdem wir die Klasse `Liste` jetzt vereinbart haben, wird es noch einmal ein wenig knifflig. Um die Klasse in unserer Klasse `Score` zu verwenden, vereinbaren wir zunächst einmal ein Feld für ein Array der Klasse `Liste`. Damit sich die Liste einfacher erweitern lässt, legen wir außerdem noch ein Feld `anzahl` mit dem Wert 10 an.

Diese Vereinbarungen in der Klasse `Score` könnten so aussehen:

```
//die Anzahl der Einträge in der Liste
int anzahl = 10;
//für die Liste
Liste[] bestenliste;
```

Code 4.5: Die neuen Felder für die Klasse Score

Zur Auffrischung:

Wenn Sie einem Feld direkt bei der Vereinbarung einen Wert zuweisen, erhält jede Instanz der Klasse denselben Wert für das Feld. Das ist in unserem Fall aber durchaus beabsichtigt, da die Liste ja immer 10 Elemente haben soll.



Im Konstruktor der Klasse `Score` erzeugen wir dann eine neue Instanz für die Liste und sorgen dafür, dass auch die Elemente der Liste erstellt werden. Die neuen Anweisungen müssen so aussehen:

```
//eine neue Instanz der Liste erstellen
bestenliste = new Liste[anzahl];
//die Elemente initialisieren
for (int i = 0; i < anzahl; i++)
    bestenliste[i] = new Liste();
```

Code 4.6: Die neuen Anweisungen für den Konstruktor



Bitte beachten Sie:

Es reicht nicht aus, nur die Liste selbst zu erzeugen. Denn bei der Liste handelt es sich ja um eine Liste mit Instanzen der Klasse `Liste`. Und diese Instanzen müssen Sie ebenfalls erzeugen. Das erledigt im vorigen Code die `for`-Schleife. Sie weist den einzelnen Elementen des Arrays `bestenliste` jeweils eine neue Instanz der Klasse `Liste` zu. Über den Konstruktor der Klasse werden dabei die Punkte auf 0 gesetzt und der Name auf „Nobody“.

Um die Liste nun auch benutzen zu können, erstellen wir zwei weitere Methoden für die Klasse `Score`. Die Methode `NeuerEintrag()` mit dem Rückgabetyp `bool` soll für das Einfügen neuer Einträge in die Liste zuständig sein und die Methode `ListeAusgeben()` für die Ausgabe der Liste. Damit wir die Ausgabe möglichst flexibel gestalten können, übergeben wir an die Methode die Zeichenfläche selbst als Argument vom Typ `Graphics` und die Dimensionen der Zeichenfläche als Argument vom Typ `RectangleF`.

Bevor wir uns an die Programmierung der Methoden machen, legen Sie bitte noch ein neues Formular mit dem Namen `NameDialog` für die Eingabe eines Namens an. Fügen Sie in dieses Formular ein Eingabefeld und eine Schaltfläche ein. Sorgen Sie dafür, dass das Formular nur über die Schaltfläche geschlossen werden kann, und setzen Sie die Eigenschaft `TopMost` für das Formular auf `True`. Andernfalls liegt das Fenster nämlich später möglicherweise unter dem Fenster des Spiels. Außerdem sollten Sie das Formular zentriert im Spielfeld anzeigen lassen. Dazu setzen Sie die Eigenschaft `StartPosition` in der Gruppe `Layout` auf `CenterParent`⁴.

Das Formular könnte zum Beispiel so aussehen:

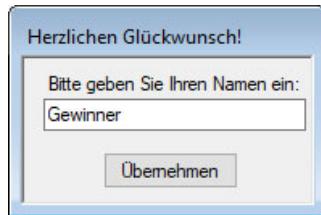


Abb. 4.3: Das Formular zur Eingabe des Namens

Jetzt gibt es lediglich noch ein kleineres Problem zu lösen: den Zugriff von außen auf den eingegebenen Namen.



Noch einmal zur Auffrischung:

Sie können aus einem Formular nicht direkt auf die Eigenschaften der Steuerelemente in einem anderen Formular zugreifen, da die Steuerelemente in der Regel mit der Sichtbarkeit `private` erzeugt werden. Daher wird aus dem einen Formular normalerweise eine öffentliche Methode in dem anderen Formular aufgerufen, die dann Werte setzt beziehungsweise liefert. Der Aufruf der Methode erfolgt dabei über eine Referenz auf das andere Formular.

4. `CenterParent` bedeutet wörtlich übersetzt „Zentriert in Eltern“. Eltern meint dabei das übergeordnete Formular – in unserem Fall also das Spiel.

In unserem Beispiel arbeiten wir dazu mit der Eigenschaft **DialogResult**⁵ des Formulars für die Eingabe des Namens. Über diese Eigenschaft können Sie gezielt auf das Anklicken einer Schaltfläche reagieren. Dabei wird das Formular nicht geschlossen, sondern lediglich ausgeblendet. Das heißt, Sie können nach wie vor aus einem anderen Formular auf Methoden im Formular zugreifen.

Der grundsätzliche Ansatz zur Kommunikation zwischen den beiden Formularen unseres Spiels könnte also so aussehen:

Wir erzeugen wie gewohnt im Formular für das Spiel eine neue Instanz des Formulars für die Eingabe des Namens und lassen es modal anzeigen.

Für die Schaltfläche **Übernehmen** im Formular für die Eingabe des Namens legen wir fest, dass sie die Eigenschaft **DialogResult** des Formulars auf **OK** setzen soll. Dazu ändern Sie die Eigenschaft **DialogResult** der Schaltfläche von **None** auf **OK**. Sie finden diese Eigenschaft in der Gruppe **Verhalten** im Eigenschaftenfenster.

Beim Ereignis **Click** der Schaltfläche **Übernehmen** im Formular für die Eingabe des Namens unternehmen wir im Formular selbst gar nichts. Wir überprüfen stattdessen im Formular für das Spiel, welches Ergebnis die Eigenschaft **DialogResult** für das Formular für die Eingabe des Namens liefert hat, und rufen dann eine Methode im Formular für die Eingabe des Namens auf, die uns den eingegebenen Namen als Zeichenkette zurückliefert.

Was sich in der Theorie vielleicht ein wenig verwirrend und kompliziert anhört, ist in der Praxis nicht weiter schwierig. Prüfen Sie noch einmal, ob Sie die Eigenschaft **DialogResult** für die Schaltfläche im Formular für die Eingabe des Namens korrekt gesetzt haben. Erstellen Sie dann eine Methode **LiefereName()** für dieses Formular. Sie sollte so aussehen:

```
public string LiefereName()
{
    return textBox1.Text;
}
```

Code 4.7: Die Methode **LiefereName()** für das Formular **NameDialog**

In der Methode **NeuerEintrag()** der Klasse **Score** überprüfen wir, ob die aktuelle Punktzahl größer ist als die Punktzahl im letzten Element der Liste. Wenn das der Fall ist, weisen wir die aktuellen Punkte dem letzten Element der Liste zu und fragen über das gerade neu erstellte Formular einen Namen ab. Diese Abfrage erledigen die folgenden Anweisungen:

```
NameDialog neuerName = new NameDialog();
if (neuerName.ShowDialog() ==
System.Windows.Forms.DialogResult.OK)
    tempName = neuerName.LiefereName();
neuerName.Close();
```

5. Übersetzt bedeutet **DialogResult** so viel wie „Dialogergebnis“.

Mit der Zeile

```
if (neuerName.ShowDialog() ==  
    System.Windows.Forms.DialogResult.OK)
```

lassen wir den Dialog modal anzeigen und prüfen, ob er den Wert `System.Windows.Forms.DialogResult.OK` liefert hat. Wenn ja, beschaffen wir uns den Namen und speichern ihn zwischen. Danach schließen wir den Dialog zur Eingabe des Namens. Das ist erforderlich, da der Dialog ja nur ausgeblendet wird.



Bitte beachten Sie:

Der Einsatz der Eigenschaft **DialogResult** ist nur dann sinnvoll, wenn Sie ein Formular modal anzeigen lassen. Denn dann wird die Verarbeitung ja so lange unterbrochen, bis das Formular geschlossen beziehungsweise ausgeblendet wird. Und das geschieht beim Setzen der Eigenschaft **DialogResult** automatisch, sobald die entsprechende Schaltfläche angeklickt wurde.

Wenn Sie das Formular dagegen normal mit der Methode `Show()` anzeigen lassen, wird die Verarbeitung quasi im Hintergrund fortgeführt und die Eigenschaft **DialogResult** unmittelbar nach dem Öffnen des Dialogs überprüft. Das führt in unserem Beispiel dazu, dass das Klicken auf die Schaltfläche **Übernehmen** keinerlei sichtbare Wirkung mehr hat.

Denken Sie beim Einsatz der Eigenschaft **DialogResult** bitte daran, dass Sie das Formular selbst mit einer entsprechenden Anweisung wieder schließen müssen. Beim Anklicken einer Schaltfläche, die einen Wert für die Eigenschaft **DialogResult** liefert, wird ein Formular lediglich ausgeblendet.

Nach der Eingabe des Namens lassen wir dann die Liste ganz einfach über die Methode `Array.Sort()` neu sortieren und liefern den Wert `true` zurück.

Falls die aktuelle Punktzahl kleiner ist als die Punktzahl im letzten Element der Liste, lassen wir die Methode `NeuerEintrag()` den Wert `false` zurückliefern.

Der komplette Quelltext für die Methode `NeuerEintrag()` der Klasse `Score` sieht so aus:

```
public bool NeuerEintrag()  
{  
    string tempName = string.Empty;  
    //wenn die aktuelle Punktzahl größer ist als der  
    //letzte Eintrag der Liste, wird der letzte Eintrag  
    //der Liste überschrieben und die Liste neu sortiert  
    if (punkte > bestenliste[anzahl-1].GetPunkte())  
    {  
        //den Namen beschaffen  
        NameDialog neuerName = new NameDialog();  
        //bitte in einer Zeile eingeben  
        if (neuerName.ShowDialog() ==  
            System.Windows.Forms.DialogResult.OK)  
            tempName = neuerName.LiefereName();  
    }  
}
```

```

        neuerName.Close();
        bestenliste[anzahl-1].SetzeEintrag(punkte, tempName);
        Array.Sort(bestenliste);
        return true;
    }
    else
        return false;
}

```

Code 4.8: Die Methode NeuerEintrag() der Klasse Score**Bitte beachten Sie:**

Wenn Sie die Klassen `Score` und `Liste` in anderen Projekten wiederverwenden wollen, müssen Sie auch die Datei mit dem Formular für das Eintragen eines Namens in das andere Projekt einbinden. Denn auf dieses Formular greift die Klasse `Score` ja beim Anlegen eines neuen Eintrags zu.

Jetzt benötigen wir noch die Methode, die die Bestenliste auf dem Bildschirm ausgibt. In unserem Beispiel soll diese Ausgabe direkt im Spielfeld erfolgen – also im Panel `spielfeld`. Die Zeichenfläche und auch den Zeichenbereich des Panels übergeben wir dabei beim Aufruf als Argumente. Die Methode `ListeAusgeben()` der Klasse `Score` könnte zum Beispiel so aussehen:

```

public void ListeAusgeben(System.Drawing.Graphics
zeichenflaeche, System.Drawing.RectangleF flaeche)
{
    //ein temporärer Pinsel
    //bitte jeweils in einer Zeile eingeben
    System.Drawing.SolidBrush tempPinsel = new
    System.Drawing.SolidBrush (System.Drawing.Color.White);
    //die Schriftart setzen
    System.Drawing.Font tempSchrift = new
    System.Drawing.Font("Arial", 12,
    System.Drawing.FontStyle.Bold);
    //für die zentrierte Ausgabe
    System.Drawing.StringFormat ausrichtung = new
    System.Drawing.StringFormat();
    //Koordinaten für die Ausgabe
    float punkteX, nameX, y;
    punkteX = flaeche.Left + 50;
    nameX = flaeche.Left + 250;
    y = flaeche.Top + 50;
    //die Ausrichtung ist zentriert
    //bitte in einer Zeile eingeben
    ausrichtung.Alignment =
    System.Drawing.StringAlignment.Center;
    //die Zeichenfläche löschen
    zeichenflaeche.Clear(System.Drawing.Color.Black);
    //den Titel ausgeben
    //bitte in einer Zeile eingeben
    zeichenflaeche.DrawString("Bestenliste",
    tempSchrift, tempPinsel, flaeche.Width/2, y, ausrichtung);
}

```

```
//und nun die Liste selbst
for (int i=0; i < anzahl; i++)
{
    y = y + 20;
    //bitte jeweils in einer Zeile eingeben
    zeichenflaeche.DrawString
    (Convert.ToString(bestenliste[i].
    GetPunkte()), tempSchrift, tempPinsel,
    punkteX, y);
    zeichenflaeche.DrawString
    (bestenliste[i].GetName(), tempSchrift,
    tempPinsel, nameX, y);
}
```

Code 4.9: Die Methode `ListeAusgeben()` der Klasse Score

Im ersten Schritt erstellen wir zunächst einmal einen Pinsel in der Farbe Weiß.

Danach setzen wir die Schriftart und erstellen eine neue Instanz `ausrichtung` der Klasse `System.Drawing.StringFormat`. Über diese Instanz können wir gleich den Text sehr einfach zentriert ausgeben.

Anschließend vereinbaren wir mehrere Variablen und setzen einige Koordinaten für die Ausgabe der Texte. Damit der Compiler keine Fehler wegen nicht möglicher Konvertierungen meldet, benutzen wir hier den Typ `float` und lassen die Fläche – das zweite Argument – als Typ `RectangleF` übergeben. Dieser Typ speichert die gleichen Daten wie der Typ `Rectangle`, allerdings als Typ `float` beziehungsweise `Single`.

Danach löschen wir dann die Zeichenfläche und geben den Titel „Bestenliste“ zentriert aus. Die Startposition berechnen wir dabei über den Ausdruck `flaeche.Width / 2`.



Bitte beachten Sie:

Bei der zentrierten Ausgabe wird der Text an der Startposition zentriert ausgegeben – und nicht etwa im Zeichenbereich.

Abschließend werden dann die Einträge der Bestenliste ausgegeben.

Damit sollte die Bestenliste jetzt funktionieren. Wir müssen nur noch dafür sorgen, dass beim Ende des Spiels überprüft wird, ob ein neuer Eintrag in der Liste angelegt werden soll. Außerdem bauen wir in das Menü `Spiel` unserer Anwendung noch einen Eintrag ein, mit dem die Bestenliste angezeigt werden kann.

4.5 Die Anzeige der Liste

Speichern Sie alle Änderungen. Wechseln Sie dann in den Quelltext der Klasse `Form1` und ergänzen Sie in der Methode für den Timer des Spiels die folgenden Anweisungen vor der Abfrage, ob ein neues Spiel gestartet werden soll:

```
//nachsehen, ob ein neuer Eintrag in der Bestenliste
//erfolgen kann
if (spielpunkte.NeuerEintrag() == true)
```

```
{  
    //Ball und Schläger verstecken  
    ball.Hide();  
    schlaeger.Hide();  
    //die Liste ausgeben  
    //bitte in einer Zeile eingeben  
    spielpunkte.ListeAusgeben(zeichenflaeche,  
    spielfeldGroesse);  
    //fünf Sekunden warten  
    System.Threading.Thread.Sleep(5000);  
    //die Zeichenfläche löschen  
    zeichenflaeche.Clear(spielfeld.BackColor);  
    //Ball und Schläger wieder anzeigen  
    ball.Show();  
    schlaeger.Show();  
}
```

Code 4.10: Abfrage und Anzeige der Bestenliste

Falls die erzielten Punkte für einen Eintrag in der Bestenliste ausreichen, wird die Liste für knapp fünf Sekunden angezeigt. Damit die Anzeige nicht gestört wird, blenden wir vorher den Ball und den Schläger über die Methode `Hide()`⁶ aus und lassen sie erst nach der Anzeige wieder erscheinen.

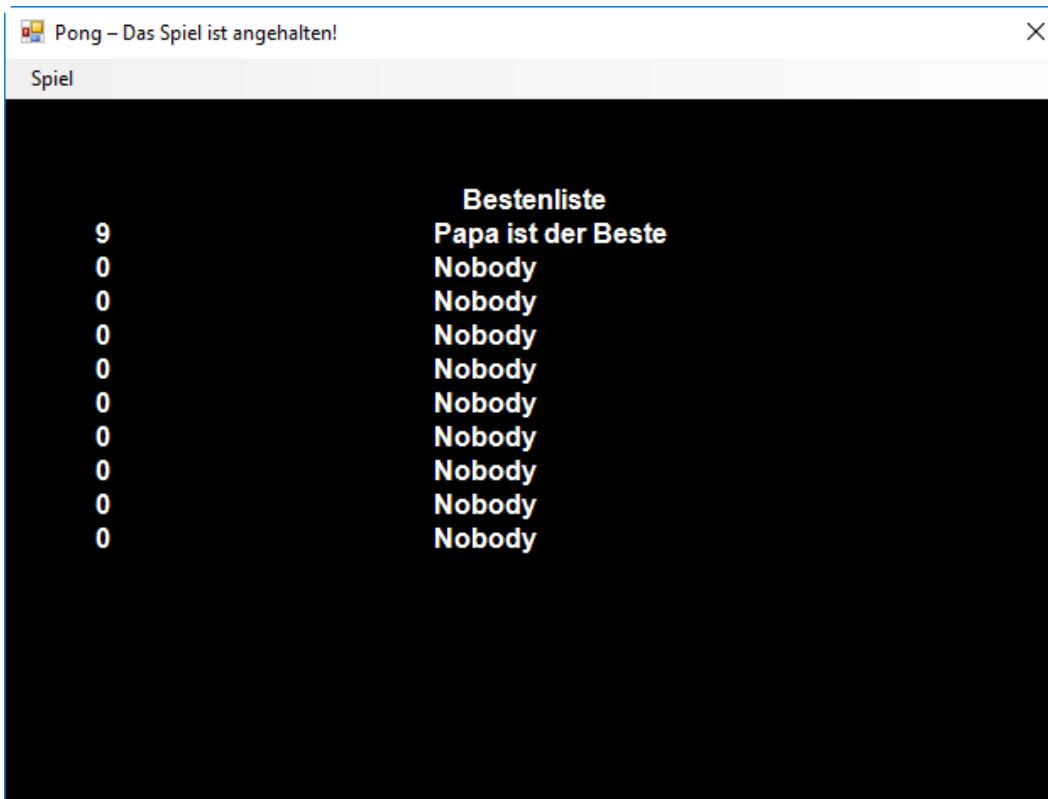


Abb. 4.4: Die Bestenliste

6. *Hide* bedeutet übersetzt „Verstecke“.

Der Menüeintrag für die Anzeige der Bestenliste sollte Sie nicht vor große Schwierigkeiten stellen. Denken Sie aber bitte daran, dass Sie bei einem laufenden Spiel das Spiel vor der Anzeige anhalten müssen. Denn sonst läuft es ja quasi im Hintergrund weiter. Die Methode für die Anzeige der Bestenliste über den Menüeintrag könnte zum Beispiel so aussehen:

```
private void BestenlisteToolStripMenuItem_Click(object
sender, EventArgs e)
{
    //zur Unterscheidung zwischen einem laufenden und
    //einem nicht gestarteten Spiel
    bool weiter = false;
    //läuft ein Spiel? dann erst einmal pausieren
    if (spielPause == false)
    {
        PauseToolStripMenuItem_Click(sender, e);
        weiter = true;
    }
    //Ball und Schläger verstecken
    ball.Hide();
    schlaeger.Hide();
    //die Liste anzeigen
    spielpunkte.ListeAusgeben(zeichenflaeche, spieldfeldGroesse);
    //fünf Sekunden warten
    System.Threading.Thread.Sleep(5000);
    //die Zeichenfläche löschen
    zeichenflaeche.Clear(spieldfeld.BackColor);
    //Ball und Schläger wieder anzeigen
    ball.Show();
    schlaeger.Show();
    //das Spiel wieder fortsetzen, wenn wir es angehalten haben
    if (weiter == true)
        PauseToolStripMenuItem_Click(sender, e);
}
```

Code 4.11: Die Anzeige der Bestenliste über den Menüeintrag

Hier gibt es lediglich eine Besonderheit. Beim Anhalten des Spiels setzen wir eine Variable `weiter`. Über diese Variable überprüfen wir am Ende der Methode, ob wir das Spiel selbst unterbrochen haben, und lassen es nur dann wieder fortsetzen. Wenn Sie in dieser Abfrage den Wert von `spielPause` direkt abfragen, stürzt das Programm ab, falls die Bestenliste unmittelbar nach dem Start des Programms angezeigt wird. Denn dann läuft das Spiel ja nicht und wird durch die Methode

`PauseToolStripMenuItem_Click()` gestartet. Dabei wird allerdings auch das Intervall für den Timer der Spielzeit über das Feld `aktuelleSpielzeit` gesetzt – und der Wert für `aktuelleSpielzeit` ist direkt nach dem Start des Programms 0.

Richtig spannend ist das Spiel bisher aber nicht. Da der Ball recht langsam ist, ist es eher schwierig, ihn nicht zu treffen. Wir werden daher im nächsten Kapitel unterschiedliche Schwierigkeitsgrade einbauen und so für ein wenig mehr Leben im Spiel sorgen.

Zusammenfassung

Um eigene Sortierungen – zum Beispiel für Arrays – festzulegen, müssen Sie eine Klasse erstellen, die die Schnittstelle `IComparable` implementiert.

In dieser Klasse legen Sie eine Methode `CompareTo()` an, die die Sortierung durchführt.

Die Methode muss drei Werte zurückgeben können: einen Wert größer als 0, einen Wert kleiner als 0 und den Wert 0 selbst. Wann welcher Wert zurückgeliefert wird, hängt von der gewünschten Sortierreihenfolge ab.

Die Methode `CompareTo()` erhält nur ein einziges Argument.

Beim Erzeugen einer Liste, die Instanzen einer Klasse enthalten soll, reicht es nicht aus, nur die Liste selbst zu erzeugen. Sie müssen auch sicherstellen, dass die Elemente der Liste korrekt erzeugt werden.

Über die Klasse `System.Drawing.StringFormat` können Sie die Ausrichtung eines Textes festlegen, der über `DrawString()` ausgegeben wird. Dazu setzen Sie die Eigenschaft `Alignment` auf den gewünschten Wert. Die Ausrichtung bezieht sich nicht auf die Zeichenfläche, sondern auf die Startposition der Ausgabe.

Aufgaben zur Selbstüberprüfung

- 4.1 Sie erstellen eine eigene Klasse für Sortierungen, die die Schnittstelle `IComparable` implementiert. Welche Vorgaben müssen Sie in dieser Klasse für die Methode `CompareTo()` beachten?

- 4.2 Welche Werte muss die Methode `CompareTo()` für eigene Vergleiche liefern, wenn Sie die Werte aufsteigend sortieren lassen wollen?

- 4.3 Die Vergleichsmethode `CompareTo()` erhält nur ein Argument, vergleicht aber trotzdem zwei Werte. Beschreiben Sie kurz, wie das möglich ist.

- 4.4 Sie haben eine Klasse vereinbart, die eine Liste abbildet, und legen ein Array mit 10 Elementen für diese Liste an. Worauf müssen Sie beim Zugriff auf die einzelnen Elemente des Arrays besonders achten?



5 Die Spieleinstellungen

In diesem Kapitel beschäftigen wir uns mit den Einstellungen für das Spiel. Der Anwender soll sowohl den Schwierigkeitsgrad als auch die Größe des Spielfelds ändern können.

Beginnen wir mit dem Schwierigkeitsgrad.

5.1 Der Schwierigkeitsgrad

Für den Schwierigkeitsgrad benutzen wir fünf verschiedenen Einstellungen – angefangen bei sehr einfach über mittel bis zu sehr schwer. Je nach Einstellung verändern sich die Geschwindigkeit des Balls und auch der mögliche Winkel, in dem sich der Ball bewegt. Bei der Einstellung „sehr schwer“ soll außerdem der Schläger kleiner werden.

Zusätzlich sollen sich je nach Schwierigkeitsgrad auch die Punkte verändern, die der Spieler beim Zurückschlagen erhält beziehungsweise die ihm bei einem verpassten Ball abgezogen werden.

Die Einstellungen des Schwierigkeitsgrads erfolgen über ein Menü, das für jeden der fünf Schwierigkeitsgrade einen eigenen Eintrag anbietet. Wenn der Anwender einen Eintrag anklickt, soll eine Methode mit Argumenten für die Einstellungen aufgerufen und zusätzlich die Eigenschaft `Interval` des Timers für die Ballbewegung verändert werden.

Da wir bisher die meisten Einstellungen für das Spiel über konstante Werte setzen, müssen wir im ersten Schritt einige zusätzliche Felder in der Klasse `Form1` vereinbaren. Diese neuen Felder könnten so aussehen:

```
//für die Punkte
int punkteMehr, punkteWeniger;
//für die Veränderung des Winkels
int winkelZufall;
```

Code 5.1: Die neuen Felder für die Klasse `Form1`

Außerdem benötigen wir noch eine Methode, die die Werte für die Einstellungen über Parameter erhält und dann die Werte der Felder entsprechend setzt. Diese Methode könnte so aussehen:

```
//setzt die Einstellungen für den Schwierigkeitsgrad
void SetzeEinstellungen(int schlaeger, int mehr, int
weniger, int winkel)
{
    schlaegerGroesse = schlaeger;
    punkteMehr = mehr;
    punkteWeniger = weniger;
    winkelZufall = winkel;
}
```

Code 5.2: Die Methode `SetzeEinstellungen()`

Hinweis:

Das Feld `schlaegerGroesse` für den Schläger haben wir bereits ganz zu Beginn des Projekts vereinbart.

Damit die geänderten Einstellungen später auch Wirkung zeigen, müssen wir nun noch in der Methode `zeichneBall()` an den passenden Stellen die konstanten Werte durch die Felder ersetzen. Das betrifft das Ändern des Winkels sowie das Erhöhen beziehungsweise Verringern der Punkte. Die entsprechenden Stellen sind im folgenden Fragment fett markiert.

```

...
//ist er wieder links, prüfen wir, ob der Schläger in
//der Nähe ist
if ((position.X == spielfeldMinX) && ((schlaeger.Top <=
position.Y) && (schlaeger.Bottom >= position.Y)))
{
    if (ballPosition.richtungX == false)
        //Punkte dazu und die Punkte ausgeben
        //bitte in einer Zeile eingeben
        ZeichnePunkte(Convert.ToString
            (spielpunkte.VeraenderePunkte(punkteMehr)));
    //die Richtung ändern
    ballPosition.richtungX = true;
    //und den Winkel
    ballPosition.winkel = zufall.Next(winkelZufall);
}
//ist der Ball hinter dem Schläger?
if (position.X < spielfeldMinX)
{
    //Punkte abziehen und die Punkte ausgeben
    //bitte in einer Zeile eingeben
    ZeichnePunkte(Convert.ToString
        (spielpunkte.VeraenderePunkte(punkteWeniger)));
}
```

Code 5.3: Die geänderte Methode `ZeichneBall()`

Anschließend sollten Sie im Konstruktor sicherstellen, dass die neu vereinbarten Felder in jedem Fall einen Wert erhalten. Dazu können Sie zum Beispiel dem Feld `punkteMehr` den Wert 1 zuweisen, dem Feld `punkteWeniger` den Wert -5 und dem Feld `winkelZufall` den Wert 5. Diese Einstellungen nehmen wir für den Schwierigkeitsgrad „einfach“.

Im nächsten Schritt legen Sie nun die Menüeinträge für das Ändern der Schwierigkeitsgrade fest und sorgen im Ereignis `Click` jedes Eintrags dafür, dass die Eigenschaft `Interval` des Timers für den Ball geändert und die Methode `SetzeEinstellungen()` mit den erforderlichen Argumenten aufgerufen wird. Das ist aber eher Fleißarbeit.

Legen Sie jetzt bitte in der Menüleiste unseres Spiels einen neuen Eintrag **Einstellungen** an. Erstellen Sie für diesen Eintrag zwei Untermenüs. Das eine Untermenü können Sie zum Beispiel **Schwierigkeitsgrad** nennen und das andere **Spielfeld**. Legen Sie dann im Untermenü **Schwierigkeitsgrad** fünf Einträge für die verschiedenen Schwierigkeitsgrade an.

Damit der Anwender sofort erkennen kann, welche Einstellung ausgewählt ist, soll der aktuelle Schwierigkeitsgrad besonders markiert werden. Dazu setzen wir für den Eintrag **Einfach** zunächst die Eigenschaft **Checked** auf `True` und schalten dann beim Anklicken eines anderen Eintrags die Markierung um.

Im letzten Schritt müssen Sie nun noch die Methoden für das Anklicken jedes einzelnen Menüeintrags programmieren. Hier setzen Sie die Eigenschaft **Checked** für den angeklickten Eintrag auf `true`, verändern das Intervall des Timers und rufen die Methode `SetzeEinstellungen()` auf. Damit immer nur ein Eintrag markiert ist, müssen Sie außerdem die Markierungen in den anderen Menüeinträgen abschalten.

Für das Anklicken des Menüeintrags **Schwer** könnten diese Anweisungen so aussehen:

```
private void SchwerToolStripMenuItem_Click(object
    sender, EventArgs e)
{
    //das Intervall für den Ball setzen
    timerBall.Interval = 25;
    //die Einstellungen setzen
    SetzeEinstellungen(50, 10, -5, 25);
    //und die Markierungen
    schwerToolStripMenuItem.Checked = true;
    sehrEinfachToolStripMenuItem.Checked = false;
    einfacheToolStripMenuItem.Checked = false;
    mittelToolStripMenuItem.Checked = false;
    sehrSchwerToolStripMenuItem.Checked = false;
}
```

Code 5.4: Die Anweisungen für das Anklicken des Menüeintrags **Schwer**

Über die Argumente der Methode `SetzeEinstellungen()` erhält die Schlägergröße den Wert 50, das Feld `punkteMehr` den Wert 10, das Feld `punkteWeniger` den Wert -5 und das Feld `winkelZufall` den Wert 25.

Erstellen Sie jetzt bitte auch noch die entsprechenden Methoden für die anderen Menüeinträge. Vorschlagswerte für die Änderungen finden Sie in der folgenden Tabelle. Sie können aber auch eigene Werte verwenden. Denken Sie in den Methoden bitte auch an das Umschalten der Markierung für die Menüeinträge.

Tab. 5.1: Vorschlagswerte für den Schwierigkeitsgrad

Schwierigkeit	Intervall des Timers	Schlägergröße	Punkte +	Punkte -	Winkel
Sehr einfach	200	100	1	-20	2
Einfach	100	50	1	-5	5
Mittel	50	50	3	-5	15
Sehr schwer	10	20	0	-5	40

Damit die Spielsteuerung nicht durcheinandergerät, sollten Sie außerdem noch dafür sorgen, dass die Einstellungen nur dann geändert werden können, wenn gerade kein Spiel stattfindet. Dazu können Sie zum Beispiel die beiden Einträge im Menü **Einstellungen** deaktivieren, wenn ein Spiel gestartet wurde.

Speichern Sie jetzt alle Änderungen und testen Sie die Einstellungen. Sie werden sehen, beim Schwierigkeitsgrad **Sehr schwer** werden Sie mit den Werten aus der vorigen Tabelle Probleme haben, den Ball wieder zurückzuspielen.

Kommen wir nun zur Veränderung der Spielfeldgröße.

5.2 Die Spielfeldgröße

Dazu erstellen wir ein neues Formular, in dem der Anwender verschiedene feste Einstellungen und die maximale Bildschirmauflösung wählen kann. Die ausgewählte Größe lassen wir zurückliefern und passen dann in der Anwendung die Größe des Formulars an. Damit auch das Spielfeld an die neue Größe angepasst wird, lassen wir es danach neu zeichnen.

Erstellen Sie jetzt ein neues Formular mit dem Namen **EinstellungenDialog**. Das Formular soll eine **GroupBox**, eine Schaltfläche zum Übernehmen und eine Schaltfläche zum Abbrechen enthalten. Setzen Sie für die Schaltfläche zum Übernehmen die Eigenschaft **DialogResult** auf **OK**. Beim Anklicken der Schaltfläche **Abbrechen** lassen Sie das Formular wie gewohnt über `Close()` schließen. Sorgen Sie außerdem dafür, dass die Symbole in der Titelleiste nicht mehr angezeigt werden, dass das Fenster immer oben liegt und dass es zentriert im Formular des Spielfelds erscheint.

Legen Sie dann in der **GroupBox** mehrere **RadioButtons** an. Ein RadioButton soll dabei für die maximale Auflösung stehen. Markieren Sie anschließend über die Eigenschaft **Checked** einen Eintrag als Standardwert.

Das Formular könnte zum Beispiel so aussehen:

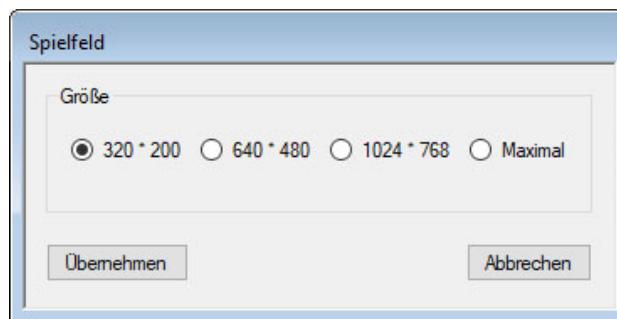


Abb. 5.1: Das Formular **EinstellungenDialog**

Erstellen Sie dann die Methode, die den ausgewählten Wert zurückliefert. Sie könnte so aussehen:

```
//die Methode liefert den ausgewählten Wert
public Point LiefereWert()
{
    Point rueckgabe = new Point(0, 0);
```

```
if (radioButton320.Checked == true)
    rueckgabe = new Point(320, 200);
if (radioButton640.Checked == true)
    rueckgabe = new Point(640, 480);
if (radioButton1024.Checked == true)
    rueckgabe = new Point(1024, 768);
if (radioButtonMaximal.Checked == true)
    //bitte in einer Zeile eingeben
    rueckgabe = new
    Point(Screen.PrimaryScreen.Bounds.Width,
    Screen.PrimaryScreen.Bounds.Height);
return rueckgabe;
}
```

Code 5.5: Die Anweisungen zum Zurückliefern der Einstellungen (Die Bezeichner hängen von den Namen ab, die Sie in Ihrem Projekt verwendet haben)

Damit wir nicht zwei unterschiedliche Methoden für die Breite und die Höhe erstellen müssen, lassen wir von der Methode den Typ `Point` zurückgeben. Ein Teil enthält die neue Breite und der andere die neue Höhe.

Zur Auffrischung:

Eine Methode kann mit Standardtechniken nur einen Wert zurückgeben.



Bei der maximalen Größe beschaffen wir uns die Werte über `Screen.PrimaryScreen.Bounds.Width` beziehungsweise `Screen.PrimaryScreen.Bounds.Height`. Diese beiden Eigenschaften liefern die Breite beziehungsweise die Höhe der aktuellen Bildschirmeinstellungen.

Beim Anklicken des Eintrags **Spielfeld** im Menü **Einstellungen** lassen Sie dann den neuen Dialog anzeigen, beschaffen die Werte und setzen die Einstellungen für das Formular und das Spielfeld neu.

Die entsprechende Methode sieht so aus:

```
private void SpielfeldToolStripMenuItem_Click(object
sender, EventArgs e)
{
    Point neueGroesse = new Point(0, 0);
    EinstellungenDialog neueWerte = new EinstellungenDialog();
    //wenn der Dialog über die "OK"-Schaltfläche
    //beendet wird
    if (neueWerte.ShowDialog() == DialogResult.OK)
    {
        //die neue Größe holen
        neueGroesse = neueWerte.LiefereWert();
        //den Dialog wieder schließen
        neueWerte.Close();
        //das Formular ändern
        this.Width = neueGroesse.X;
        this.Height = neueGroesse.Y;
        //neu ausrichten
    }
}
```

```

//bitte jeweils in einer Zeile eingeben
this.Left = (Screen.PrimaryScreen.Bounds.Width
- this.Width) / 2;
this.Top = (Screen.PrimaryScreen.Bounds.Height
- this.Height) / 2;
//die Zeichenfläche neu beschaffen
zeichenflaeche = spielfeld.CreateGraphics();
//das Spielfeld neu setzen
SetzeSpielfeld();
//Spielfeld löschen
zeichenflaeche.Clear(spielfeld.BackColor);
//und einen neuen Ball und einen neuen Schläger zeichnen
NeuerBall();
//das Spielfeld neu zeichnen
ZeichneSpielfeld();
}
}

```

Code 5.6: Die Methode SpielfeldToolStripMenuItem_Click()

Zuerst wird die Größe des Formulars verändert und das Formular neu auf dem Desktop zentriert. Danach wird die Variable `zeichenflaeche` neu auf die Zeichenfläche des Spielfelds positioniert und die neuen Dimensionen des Spielfelds über die Methode `SetzeSpielfeld()` gesetzt. Anschließend wird das Spielfeld gelöscht, ein neuer Ball sowie ein neuer Schläger gezeichnet und das Spielfeld neu erstellt.

Hinweis:

Das etwas umständliche Neupositionieren des Formulars „per Hand“ ist erforderlich, da Sie über die Eigenschaft `StartPosition` nur die Position beim ersten Anzeigen eines Formulars setzen können.

Damit ist unser Spiel zunächst einmal fertig programmiert.

Abschließend noch ein paar Ideen für Erweiterungen und Verbesserungen:

- Die Anzeige der Zeit kann unter Umständen flackern. Das liegt daran, dass das Panel immer wieder neu gezeichnet wird. Sie können die Anweisung zum Zeichnen der Zeit aber einfach aus der Methode `Spielfeld_Paint()` löschen. Die Anzeige erfolgt ja automatisch nach dem Start eines neuen Spiels.
- Zurzeit verschwindet die Anzeige der Punkte noch, wenn der Ball darüberfährt oder wenn die Bestenliste angezeigt wurde. Sie erscheint erst wieder, wenn neue Punkte erzielt werden. Die verschwundene Anzeige lässt sich aber sehr leicht beheben. Nehmen Sie das Zeichnen der Punkte mit in das Ereignis `Paint` des Spielfelds auf. Das kann dann aber wieder zu einem Flackern der Anzeige führen.
- Die Anzeige der Bestenliste haben wir etwas lieblos programmiert. Hier können Sie die grafische Gestaltung zum Beispiel noch ein wenig aufwendiger machen.
- Bei einem sehr kleinen Spielfeld wird die Anzeige der Bestenliste verstümmelt, da der Platz nicht ausreicht. Sie könnten hier die Anzeige entweder so umprogrammieren, dass sie auch bei einem kleinen Spielfeld vollständig zu sehen ist, oder aber einfach keine sehr kleinen Spielfelder zulassen.

- Auch die Einstellungen für den Schwierigkeitsgrad und die Größe des Spielfelds sind zurzeit ja nur einmal direkt nach dem Starten des Spiels möglich. Auch hier können Sie die Anweisungen so ändern, dass die Einstellungen nach jedem Spiel möglich sind.
- Bisher kann man das aktuelle Spiel nur durch den Neustart eines anderen Spiels oder durch das Schließen der Anwendung beenden. Hier könnten Sie zum Beispiel eine weitere Methode erstellen, über die lediglich das aktuelle Spiel beendet wird.
- Auch die Vergabe der Punkte können Sie noch verfeinern – zum Beispiel, indem jede Berührung mit einer Wand ebenfalls zum Erhöhen des Punktestands führt. Damit würde dann auch der zusätzliche Schwierigkeitsgrad berücksichtigt, wenn der Ball mit einem kleinen Winkel im Zick-Zack über das Spielfeld wandert.
- Sie können eine Methode einbauen, über die auch die Spieldauer verändert werden kann, und dann mehrere Bestenlisten für die unterschiedlichen Spieldauern erstellen.
- Sie können das Spiel mit akustischen Effekten versehen – zum Beispiel bei Kontakten des Balls mit dem Schläger oder einer Wand und beim Ausfahren des Balls.
- An einigen Stellen lässt sich das Spiel noch etwas kompakter programmieren, da wir vor allem auf die Verständlichkeit geachtet haben. So könnten Sie ja zum Beispiel die Anzeige der Bestenliste nach dem Eintragen eines Wertes und nach dem Anzeigen über das Menü zusammenfassen.

Sicherlich wird Ihnen selbst auch noch die eine oder andere Erweiterung einfallen. Ein paar zusätzliche Funktionen warten wieder als Einsendeaufgabe auf Sie.

Zusammenfassung

Über den Ausdruck `Screen.PrimaryScreen` können Sie Informationen zum Bildschirm beschaffen.

Aufgaben zur Selbstüberprüfung

- 5.1 Sie wollen in einer Anwendung die aktuelle Bildschirmauflösung ermitteln.
Wie erhalten Sie diese Werte?

- 5.2 Sie haben ein Menü erstellt, in dem der aktuelle Eintrag durch ein Häkchen markiert werden soll. Was müssen Sie beim Anklicken der Einträge selbst sicherstellen?

Schlussbetrachtung

Herzlichen Glückwunsch!

Sie haben ein recht aufwendiges Spiel erstellt und auf mehrere Klassen verteilt. Diese Verteilung führt zwar in einigen Teilen zu deutlich mehr Aufwand, sorgt aber dafür, dass Sie einmal programmierte Funktionalität später ohne Schwierigkeiten immer wieder verwenden können. So lässt sich ja die Bestenliste aus unserem Pong-Spiel ohne Weiteres auch in anderen Spielen einsetzen.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 2

- 2.1 Um ein Formular beim ersten Öffnen auf dem Desktop zu zentrieren, muss die Eigenschaft **StartPosition** den Wert `CenterScreen` erhalten.
- 2.2 Um eine Grafik in einem Formular zu animieren, verwenden Sie ein Steuerelement, das eine Grafik darstellen kann – zum Beispiel ein **Panel** oder auch eine **PictureBox**. Für die Bewegung verändern Sie dann die Position des Steuerelements – zum Beispiel in einer Schleife.

Kapitel 3

- 3.1 Sie müssen sicherstellen, dass die vorige Ausgabe vor der neuen Ausgabe „gelöscht“ wird. Dazu können Sie zum Beispiel ein Rechteck in der Hintergrundfarbe an die Ausgabeposition zeichnen lassen.
- 3.2 Damit ein Formular ständig im Vordergrund angezeigt wird, setzen Sie die Eigenschaft **TopMost** auf `True`.
- 3.3 Um einen Text über die Methode `DrawString()` fett auszugeben, erzeugen Sie eine neue Instanz der Klasse `Font`. An den Konstruktor übergeben Sie dabei die Schriftart, die Größe und die fette Formatierung über den Ausdruck `FontStyle.Bold`.
- 3.4 Sie benötigen einen zweiten Timer, der die bereits verstrichene Zeit mitzählt. Beim Unterbrechen und Fortsetzen des ersten Timers berechnen Sie die noch verbleibende Zeit aus der ursprünglichen Dauer des Timers und der bereits verstrichenen Zeit. Diesen Wert weisen Sie dann der Eigenschaft `Interval` des ersten Timers zu.

Kapitel 4

- 4.1 Die Methode `CompareTo()` muss als Rückgabe einen `int`-Typ liefern und einen Typ `object` als Parameter erhalten.
- 4.2 Bei aufsteigender Sortierung muss die Methode einen Wert größer als 0 liefern, wenn der erste Wert größer ist als der zweite. Ist der zweite Wert größer als der erste, muss ein Wert kleiner als 0 geliefert werden. Sind die beiden Werte gleich, muss der Wert 0 zurückgegeben werden.
- 4.3 Die Methode erhält als Argument nicht die aktuelle Instanz, sondern die andere Instanz – also die Instanz, die mit der aktuellen Instanz verglichen werden soll.
- 4.4 Vor dem Zugriff müssen Sie für jedes Element der Liste ebenfalls eine Instanz erzeugen. Denn das Array enthält zunächst einmal nur eine Instanz der Liste, aber keine Instanzen für die einzelnen Einträge in der Liste.

Kapitel 5

- 5.1 Die aktuelle Bildschirmauflösung erhalten Sie über
`Screen.PrimaryScreen.Bounds.Width` beziehungsweise
`Screen.PrimaryScreen.Bounds.Height`.
- 5.2 Beim Anklicken der Einträge müssen Sie zum einen dafür sorgen, dass das Häkchen für den Eintrag gesetzt wird. Zum anderen muss bei den anderen Einträgen das Häkchen wieder entfernt werden.

B. Glossar

Client-Bereich	Der Client-Bereich ist der innere Bereich eines Steuerelements ohne Rahmen oder Ähnliches. Bei einem Formular stellt der Client-Bereich zum Beispiel das Innere des Formulars dar.
GDI+	GDI+ ist eine geräteunabhängige Programmierschnittstelle für die Ausgabe von Informationen auf grafikfähigen Geräten.
GDI+ API	Siehe GDI+
Graphics Device Interface+ Application Programming Interface	Siehe GDI+
Interface	Siehe Schnittstelle
Modales Fenster	Ein modales Fenster steht immer im Vordergrund und kann auch nicht minimiert werden. Mit der eigentlichen Anwendung kann erst dann weitergearbeitet werden, wenn das modale Fenster wieder geschlossen wird. Typische Beispiele für modale Fenster sind unter anderem die Öffnen- und Speicherndialoge von Windows.
Nicht visuelle Steuerelemente	Nicht visuelle Steuerelemente werden beim Ausführen einer Anwendung nicht direkt im Formular angezeigt. Es handelt sich also nicht um Bedienelemente im eigentlichen Sinne. Ein typisches nicht visuelles Steuerelement ist zum Beispiel das Steuerelement OpenFileDialog für den Standardöffnendialog oder das Steuerelement Timer .
Pixel	Ein Pixel ist ein Bildpunkt.
Schnittstelle	Eine Schnittstelle ist – etwas vereinfacht ausgedrückt – eine Klasse, die die Signatur von bestimmten Methoden enthält. Die Methoden selbst – also die eigentliche Funktionalität – müssen in eigenen Klassen im Detail implementiert werden – und zwar sämtliche Methoden, die die Schnittstelle vorgibt.
Signatur	Eine Signatur beschreibt den Namen und die Typen der Parameterliste einer Methode.
Timer	Ein <i>Timer</i> löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis aus, in dem Sie Anweisungen verarbeiten lassen können. Ein <i>Timer</i> ermöglicht damit die zeitgesteuerte Verarbeitung von Anweisungen.

Zeichenfläche

Die Zeichenfläche ist der Bereich eines Steuerelements, in dem Sie selbst zeichnen können.

Nicht alle Steuerelemente verfügen über eine Zeichenfläche.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmiertechniken.* 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in Visual C# 2019. Ideal für Programmieranfänger geeignet.* 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Das Pong-Spiel	4
Abb. 2.1	Die Menüleiste im Formular	6
Abb. 2.2	Der Elementsammlungs-Editor	7
Abb. 2.3	Der neu eingefügte Menüeintrag	8
Abb. 2.4	Der Elementsammlungs-Editor für die Einträge im Menü Spiel	9
Abb. 2.5	Das geöffnete Menü Spiel im Designer	9
Abb. 2.6	Das Spielfeld	15
Abb. 3.1	Die Anzeige der Spielzeit oben rechts	26
Abb. 3.2	Das unterbrochene Spiel	28
Abb. 4.1	Das Anlegen einer neuen Klasse	35
Abb. 4.2	Die Punktanzeige	38
Abb. 4.3	Das Formular zur Eingabe des Namens	42
Abb. 4.4	Die Bestenliste	47
Abb. 5.1	Das Formular EinstellungenDialog	54
Abb. H.1	Das Formular für die Farbeinstellungen	72

E. Tabellenverzeichnis

Tab. 5.1 Vorschlagswerte für den Schwierigkeitsgrad.....	53
--	----

F. Codeverzeichnis

Code 2.1	Die Felder für das Spiel	11
Code 2.2	Der Konstruktor	12
Code 2.3	Die Methode SetzeSpielfeld()	12
Code 2.4	Die Methode ZeichneSpielfeld()	13
Code 2.5	Die Methoden zum „Zeichnen“ von Ball und Schläger	14
Code 2.6	Das Bewegen des Schlägers	15
Code 2.7	Abfrage der aktuellen Position des Schlägers in der Methode ZeichneSchlaeger()	16
Code 2.8	Die Methode für den Timer	18
Code 2.9	Die Abfragen auf die Spielfeldränder in der Methode ZeichneBall()	18
Code 2.10	Die Abfrage auf einen „Zusammenstoß“ zwischen Ball und Schläger ..	19
Code 2.11	Das Verändern des Bewegungswinkels.....	19
Code 2.12	Die vollständige Methode ZeichneBall()	21
Code 3.1	Der erweiterte Konstruktor für das Formular	25
Code 3.2	Die Methode ZeichneZeit()	25
Code 3.3	Die Anweisungen für den Timer timerSekunde	26
Code 3.4	Die Anweisungen für das Klicken auf den Menüeintrag Pause	28
Code 3.5	Die Methode NeuesSpiel()	30
Code 3.6	Die Anweisungen für das Anklicken des Menüeintrags Neues Spiel	30
Code 3.7	Das Beenden des Spiels	31
Code 4.1	Die Vereinbarung der Klasse Score	36
Code 4.2	Die Methode ZeichnePunkte()	36
Code 4.3	Das Verändern der Punktzahl	37
Code 4.4	Die Klasse für die Liste	40
Code 4.5	Die neuen Felder für die Klasse Score.....	41
Code 4.6	Die neuen Anweisungen für den Konstruktor	41
Code 4.7	Die Methode LiefereName() für das Formular NameDialog	43
Code 4.8	Die Methode NeuerEintrag() der Klasse Score	45
Code 4.9	Die Methode ListeAusgeben() der Klasse Score	46
Code 4.10	Abfrage und Anzeige der Bestenliste	47
Code 4.11	Die Anzeige der Bestenliste über den Menüeintrag	48
Code 5.1	Die neuen Felder für die Klasse Form1	51
Code 5.2	Die Methode SetzeEinstellungen()	51
Code 5.3	Die geänderte Methode ZeichneBall()	52

Code 5.4	Die Anweisungen für das Anklicken des Menüeintrags Schwer	53
Code 5.5	Die Anweisungen zum Zurückliefern der Einstellungen (Die Bezeichner hängen von den Namen ab, die Sie in Ihrem Projekt verwendet haben)	55
Code 5.6	Die Methode SpielfeldToolStripMenuItem_Click()	56

G. Sachwortverzeichnis

A

- Anzeige der Liste 46
Ausgabe der Punkte 36

B

- Bestenliste 34
Bewegen
 des Balls 17
 des Schlägers 15
Bildschirmeinstellung 55

C

- CompareTo() 40

E

- Eigenschaft
 TopMost 32
Elementsammlungs-Editor 7
Ereignis
 MouseMove 15
 Tick 17

H

- Hide() 47

I

- IComparable.CompareTo() 40

K

- Klasse
 Score 35
Kommunikation
 zwischen den beiden Formularen 43

M

- Menüleiste 5
MenuStrip 5
Methode
 CompareTo() 40
 Hide() 47
MouseMove 15

N

- Neu starten 29

S

- Schnittstelle
 IComparable 40
Schwierigkeitsgrad 51
Score 35
Signatur 40
Spielende 31
Spielfeld
 erstellen 5
Spielfeldgröße 54
Spielregel 3
Spielzeit
 ausgeben 25
Standardmethode
 IComparable.CompareTo() 40
Starten und Anhalten 23
Steuerelement
 MenuStrip 5

T

- Tick 17
Timer 17
TopMost 32

U

- Unterbrechen und Fortsetzen 27

V

- Verwaltung der Liste 38

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH11D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Schicken Sie das vollständige Projekt mit allen Unterordnern und Dateien ein.

Da alle Aufgaben zum Spiel gehören, müssen Sie nur ein Projekt einsenden. Bitte markieren Sie deutlich, zu welcher Aufgabe die neuen Anweisungen gehören – zum Beispiel durch Kommentare.

Beschreiben Sie außerdem, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Steuerelemente Sie in dem Formular verwenden und welche Besonderheiten im Spiel berücksichtigt werden müssen.

Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt vor dem Versand mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

1. Beim Setzen der Größe für das Spielfeld wird bisher immer die Einstellung 320×200 Punkte markiert beziehungsweise das Optionsfeld, das Sie im Formular aktiviert haben.

Ändern Sie das Spiel so, dass im Formular für die Größenänderung des Spielfelds die jeweils aktuell gesetzte Einstellung des Spielfelds korrekt markiert wird.

30 Pkt.

2. Erweitern Sie das Pong-Spiel um eine Einstellung der Farben. Der Anwender soll dabei die Farben für den Rahmen und das Spielfeld selbst festlegen können. Für den Schläger und den Ball sowie die Anzeige der Punkte und der Spielzeit soll die Farbe des Rahmens benutzt werden.

Im Formular für die Farbauswahl soll neben der Auswahl auch eine Art Vorschau für die Änderungen angezeigt werden. Die Vorschau soll aktualisiert werden, sobald eine andere Farbe ausgewählt wurde. Dazu können Sie das Neuzeichnen des Formulars über die Methode `Refresh()` erzwingen.

Die weitere Gestaltung des Formulars ist Ihnen freigestellt. Sie können zum Beispiel das Formular für die Einstellung der Spielfeldgröße erweitern. Es könnte dann so aussehen:

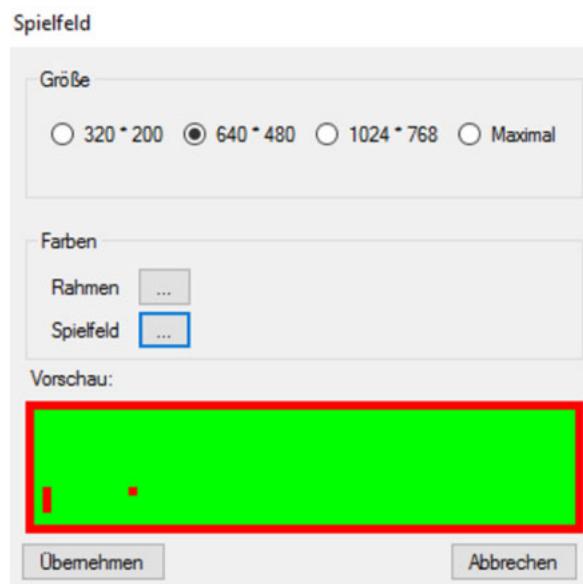


Abb. H.1: Das Formular für die Farbeinstellungen

Sorgen Sie dafür, dass im Formular die aktuellen Farben des Spielfelds benutzt werden. Diese Farben sollen sowohl bei der Auswahl der Farben als auch in der Vorschau angezeigt werden.

70 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit der Registrierung und XML-Dateien

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0919N01

CSHP12D

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit der Registrierung und XML-Dateien

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit der Registrierung und XML-Dateien

Inhaltsverzeichnis

Einleitung	1
1 Arbeiten mit der Registrierung	3
1.1 Der Aufbau der Registrierung	3
1.2 Die Klassen Registry und RegistryKey	7
1.3 Ein einfaches Beispiel	9
1.4 Eine Kennwortabfrage	13
1.5 Eine Listenanzeige für die Registrierung	18
Zusammenfassung	22
2 Arbeiten mit XML-Dateien	24
2.1 Der Aufbau von XML-Dateien	24
2.2 XML-Dateien mit dem Editor erstellen	26
2.3 Lesen aus einer XML-Datei	28
2.4 Schreiben in eine XML-Datei	32
2.5 Ein etwas anspruchsvolleres Beispiel	37
Zusammenfassung	42
3 Speichern der Einstellungen für das Pong-Spiel	44
3.1 Vorüberlegungen	44
3.2 Spielfeldgröße und Schwierigkeitsgrad	45
3.3 Die Bestenliste	50
Zusammenfassung	53
Schlussbetrachtung	55

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	56
B.	Glossar	58
C.	Literaturverzeichnis	60
D.	Abbildungsverzeichnis	61
E.	Tabellenverzeichnis	62
F.	Codeverzeichnis	63
G.	Medienverzeichnis	64
H.	Sachwortverzeichnis	65
I.	Einsendeaufgabe	67

Einleitung

Bisher haben wir Daten aus einer Anwendung immer über Methoden der entsprechenden Steuerelemente geladen – zum Beispiel die Bilder in unserem Bildbetrachter. Dabei mussten Sie nicht viel mehr machen, als die richtige Methode aufzurufen und den Dateinamen anzugeben.

Für sehr viele Steuerelemente gibt es aber keine eigenen Methoden, um Daten zu speichern und zu laden – zum Beispiel für ein Eingabefeld. Aber auch solche Daten können Sie dauerhaft auf einem Datenträger speichern und wieder zurücklesen. Damit werden wir uns jetzt beschäftigen.

In diesem Studienheft stellen wir Ihnen den Umgang mit der Windows-Registrierung und XML-Dateien vor. Im Einzelnen lernen Sie:

- wofür die Registrierung von Windows eingesetzt wird,
- wie die Registrierung von Windows aufgebaut ist,
- warum Sie bei Zugriffen auf die Registrierung sehr vorsichtig sein müssen,
- wie Sie über die Klassen `Registry` und `RegistryKey` auf die Registrierung zugreifen,
- wie Sie Eingabefelder für Kennwörter erstellen,
- wie Sie eine Anwendung aus einem beliebigen Formular der Anwendung beenden können,
- wie XML-Dateien aufgebaut sind,
- wie Sie Daten mit der Klasse `XmlReader` aus einer XML-Datei lesen,
- wie Sie Daten mit der Klasse `XmlWriter` in einer XML-Datei speichern,
- wie Sie die Einstellungen für das Schreiben in eine XML-Datei setzen,
- wie Sie den Namen einer Datei aus dem Namen beziehungsweise dem Pfad der Anwendung ableiten und
- wie Sie die Bestenliste und die Einstellungen aus dem Pong-Spiel in XML-Dateien speichern und wieder lesen.

Christoph Siebeck

1 Arbeiten mit der Registrierung

In diesem Kapitel lernen Sie, wie Sie Daten in der Registrierung von Windows speichern und lesen.

Die **Registrierung** – auch **Registry** genannt – ist eine Art Datenbank, in der alle wichtigen Einstellungen von Windows zentral zusammenlaufen. Sie wurde mit Windows 95 eingeführt.



1.1 Der Aufbau der Registrierung

Die Registrierung besteht – etwas vereinfacht ausgedrückt – aus einzelnen Einträgen mit eindeutigen Namen, denen jeweils ein Wert zugeordnet wird.

Die einzelnen Einträge werden dabei über Schlüssel zusammengefasst, die sich in einer hierarchischen Struktur befinden. Auf der obersten Ebene dieser Struktur liegen unter anderem folgende fünf Hauptschlüssel:

Tab. 1.1: Hauptschlüssel der Registrierung¹

Name	Bedeutung
HKEY_CLASSES_ROOT	Dieser Schlüssel enthält Daten über installierte Programme und Dateierweiterungen.
HKEY_CURRENT_USER	In diesem Schlüssel werden Einstellungen für den aktuell angemeldeten Anwender gespeichert.
HKEY_LOCAL_MACHINE	Dieser Schlüssel speichert vor allem Daten zur verwendeten Hardware.
HKEY_USERS	In diesem Schlüssel werden die Daten aller angelegten Benutzer gespeichert.
HKEY_CURRENT_CONFIG	Dieser Schlüssel speichert die Hardwarekonfiguration des aktuell angemeldeten Anwenders.

Unterhalb der Hauptschlüssel befinden sich dann weitere Schlüssel, die selbst ebenfalls wieder Schlüssel enthalten. Jeder einzelne Schlüssel kann dabei mehrere Einträge speichern. Ein Eintrag selbst setzt sich aus einem Namen, einem Datentyp und dem Wert zusammen.

Die Struktur der Registrierung ähnelt damit der Struktur des Dateisystems. Die einzelnen Schlüssel entsprechen dabei den Ordnern und die Einträge den Dateien.

Die Daten in der Registrierung werden in binärer Form gespeichert. Der Zugriff erfolgt mit einem speziellen Windows-Programm – dem **Registrierungs-Editor**.

1. Das **H** vorn im Namen eines Hauptschlüssels steht für *hive* (übersetzt: „Bienenstock“). **Key** bedeutet übersetzt „Schlüssel“. Die Namen der einzelnen Hauptschlüssel bedeuten übersetzt etwa „Wurzelklassen“ (**CLASSES_ROOT**), „aktueller Anwender“ (**CURRENT_USER**), „lokaler Rechner“ (**LOCAL_MACHINE**), „Anwender“ (**USERS**) und „aktuelle Konfiguration“ (**CURRENT_CONFIG**).

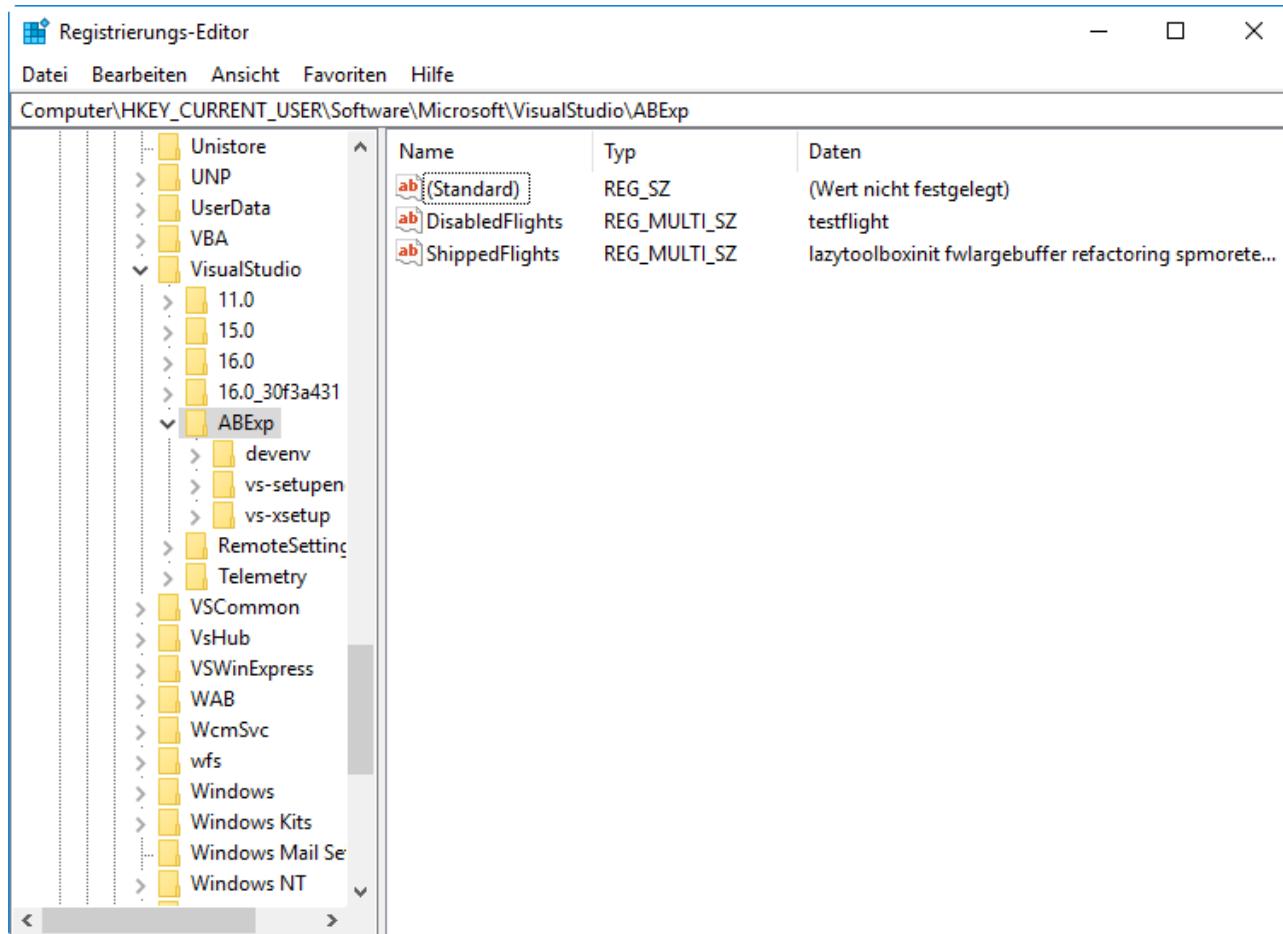


Abb. 1.1: Der Registrierungs-Editor

Im linken Bereich des Registrierungs-Editors werden die Schlüssel in der hierarchischen Struktur angezeigt, im rechten Bereich sehen Sie die einzelnen Einträge des aktuell markierten Schlüssels.



Bitte beachten Sie unbedingt:

Die Registrierung von Windows sollten Sie außerordentlich sorgfältig behandeln. Nehmen Sie auf keinen Fall Änderungen an irgendwelchen Einträgen vor, wenn Sie nicht genau wissen, was sich hinter dem Eintrag verbirgt und welche Auswirkungen die Änderungen haben. Unter Umständen funktioniert Windows nach Änderungen nicht mehr. Das kann sogar so weit gehen, dass bereits der Start von Windows scheitert.

Da die Registrierung sehr empfindlich ist, „versteckt“ sich der Registrierungs-Editor auch ein wenig. Am einfachsten starten Sie das Programm, wenn Sie im Suchfeld im Startmenü oder in der Taskleiste **regedit** eingeben.

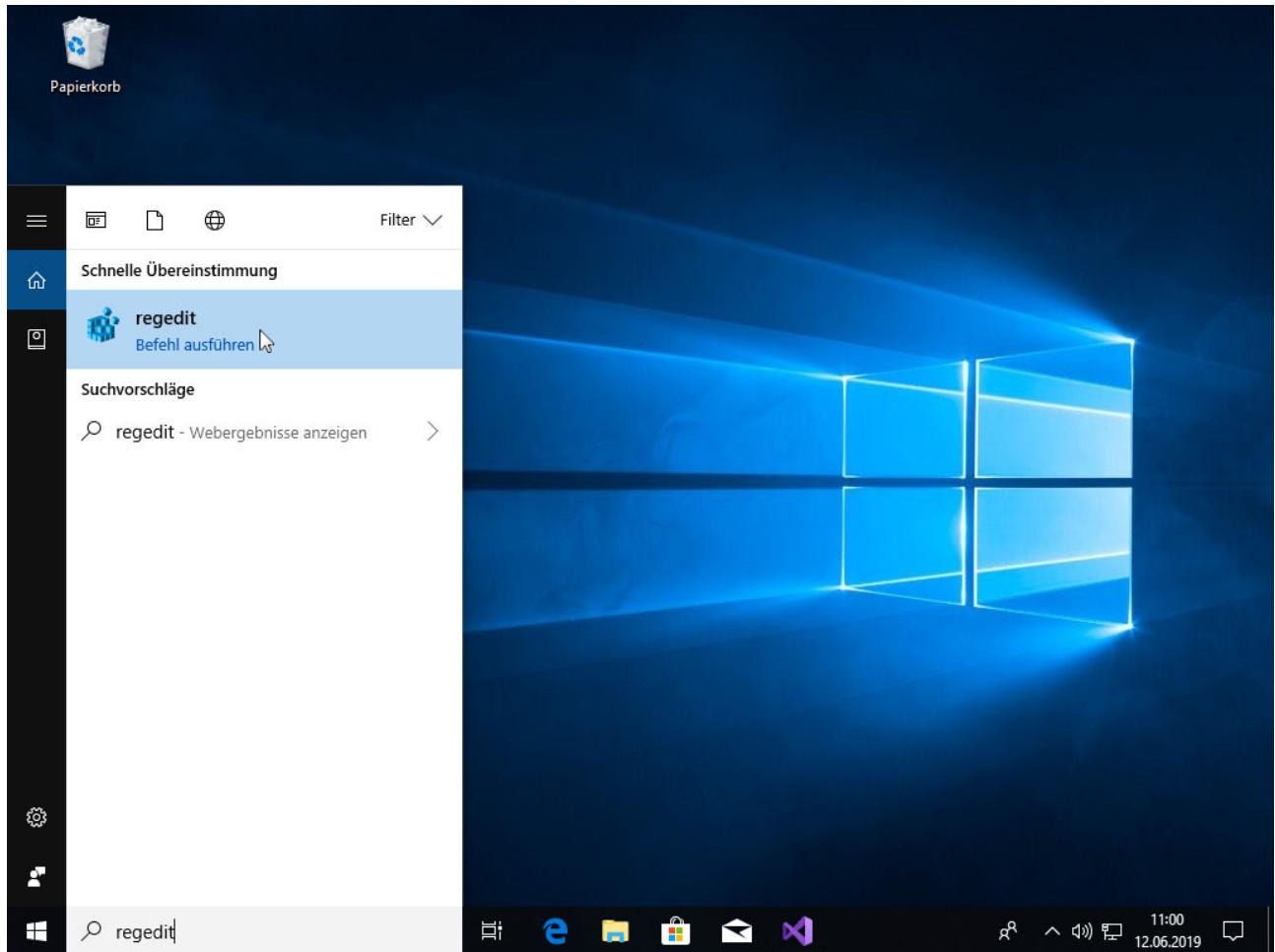


Abb. 1.2: Der Registrierungs-Editor in der Liste der Suchergebnisse

Aus der Liste der Suchergebnisse können Sie den Registrierungs-Editor anschließend durch einen Mausklick aufrufen. Direkt nach dem Start müssen Sie der Aktion in der Regel noch einmal ausdrücklich zustimmen. Klicken Sie dazu im Dialog **Benutzerkontensteuerung** auf die Schaltfläche **Ja**.

An dieser Stelle aber noch einmal die eindringliche Warnung:

Nehmen Sie keine Änderungen an Einträgen vor, wenn Sie nicht genau wissen, was geschieht.



Zur Sicherheit sollten Sie vor dem Zugriff auf die Registrierung immer eine Sicherungskopie der Daten anlegen. Dazu klicken Sie im Menü **Datei** des Registrierungs-Editors auf die Funktion **Exportieren ...**. Markieren Sie dann im Fenster **Registrierungsdatei exportieren** den Eintrag **Alles** unten links.

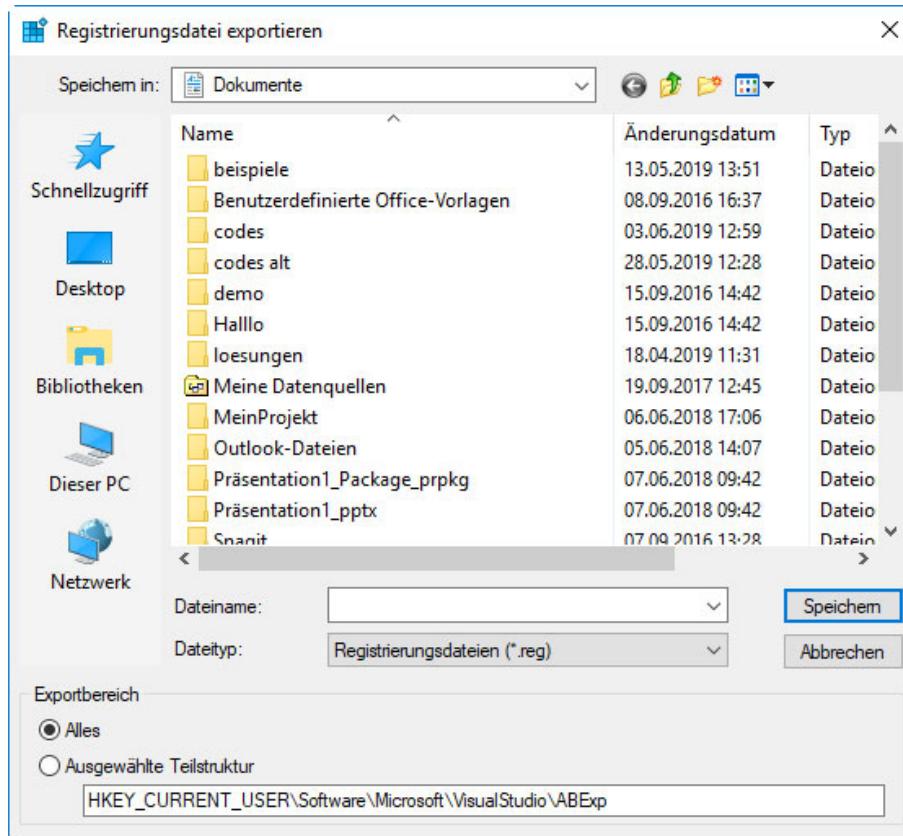
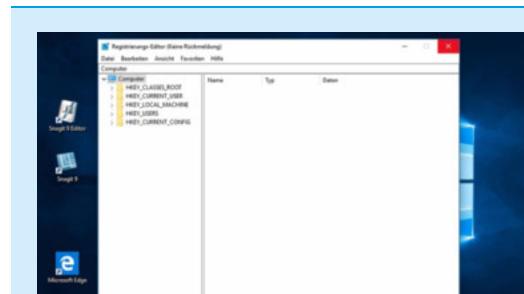


Abb. 1.3: Das Exportieren der Registrierungsdatei

Legen Sie anschließend einen Ordner sowie einen Dateinamen fest und sichern Sie die Daten über die Schaltfläche **Speichern**. Bitte beachten Sie, dass die Sicherungskopie abhängig vom Umfang Ihrer Registrierung recht groß werden kann. Das Speichern kann also durchaus einen Moment dauern.

Im Fall der Fälle können Sie die Einträge aus der Sicherungskopie mit der Funktion **Importieren ...** im Menü **Datei** des Registrierungs-Editors wieder zurücklesen. Denken Sie aber bitte daran, dass Sie dazu den Registrierungs-Editor auch starten müssen. Arbeiten Sie daher auch bei einer vorhandenen Sicherungskopie äußerst sorgfältig und verzichten Sie auf Experimente.



In diesem Video zeigen wir Ihnen, wie Sie den Registrierungs-Editor starten und wie Sie eine Sicherungskopie der Registrierung erstellen.

www.dfz.media/jjjxmf

Video 1.1: Sicherungskopie der Registrierung erstellen

Schauen wir uns nun an, wie Sie mit C# auf die Einträge in der Registrierung zugreifen.

1.2 Die Klassen Registry und RegistryKey

Der Zugriff erfolgt über die Klassen `Registry` und `RegistryKey`. Die Klasse `Registry` stellt dabei Felder vom Typ `RegistryKey` zur Verfügung. Diese Felder stehen für die Hauptschlüssel der Registrierung. Eine Zuordnung der Felder zum jeweiligen Hauptschlüssel der Registrierung finden Sie in der Tab. 1.2.

Tab. 1.2: Felder der Klasse Registry

Feld	Hauptschlüssel
<code>ClassesRoot</code>	<code>HKEY_CLASSES_ROOT</code>
<code>CurrentUser</code>	<code>HKEY_CURRENT_USER</code>
<code>LocalMachine</code>	<code>HKEY_LOCAL_MACHINE</code>
<code>Users</code>	<code>HKEY_USERS</code>
<code>CurrentConfig</code>	<code>HKEY_CURRENT_CONFIG</code>

Bitte beachten Sie, dass sowohl die Klasse `Registry` als auch die Klasse `RegistryKey` im Namensraum `Microsoft.Win32` vereinbart sind.

Der Zugriff auf einen Schlüssel in der Registrierung erfolgt über die Methoden `OpenSubKey()`, `GetValue()` und `SetValue()` der Klasse `RegistryKey`.

Die Methode `OpenSubKey()`² öffnet dabei einen bereits vorhandenen Schlüssel. Der Name des Schlüssels muss als Zeichenkette übergeben werden. Das folgende Fragment erzeugt zum Beispiel eine Instanz `regSchluessel` der Klasse `RegistryKey` und öffnet dann über die Methode `OpenSubKey()` den Schlüssel `RegistryDemo` im Zweig `SOFTWARE` des Hauptschlüssels `HKEY_CURRENT_USER`.

```
RegistryKey regSchluessel;
regSchluessel = Registry.CurrentUser.
OpenSubKey("Software\\RegistryDemo");
```

Code 1.1: Zugriff auf einen Schlüssel in der Registrierung

Hinweis:

Wenn der Schlüssel nicht vorhanden ist oder der Zugriff scheitert, liefert die Methode `OpenSubKey()` den Wert `null` zurück.

Denken Sie bitte daran, dass Sie das Trennzeichen `\` doppelt angeben müssen. Alternativ können Sie auch das Zeichen `@` vor das erste Anführungszeichen der Zeichenkette setzen.

Über die Methode `GetValue()`³ können Sie dann den Wert eines Eintrags in dem Schlüssel lesen. Dazu übergeben Sie den Namen des Eintrags als Zeichenkette. Da die Methode `GetValue()` den Wert als Typ `object` zurückliefert, müssen Sie den gelesenen Wert in der Regel noch in den gewünschten Typ konvertieren.

2. `OpenSubKey` lässt sich mit „Öffne untergeordneten Schlüssel“ übersetzen.

3. `GetValue` bedeutet so viel wie „Hole Wert“.

Die folgende Anweisung liest zum Beispiel über die Methode `GetValue()` den Wert eines Eintrags `Eintrag` und gibt ihn über eine **MessageBox** aus:

```
MessageBox.Show(Convert.ToString(regSchluessel.  
GetValue("Eintrag")));
```

Das Schreiben von Daten in die Registrierung erfolgt über die Methode `SetValue()`⁴. Die Methode erwartet zuerst den Namen des Eintrags und dann den Wert, der für den Eintrag gespeichert werden soll. Die folgende Anweisung würde zum Beispiel den Text aus einem Eingabefeld in dem Eintrag `Eintrag` speichern.

```
regSchluessel.SetValue("Eintrag", textBoxSchreiben.Text);
```

Allerdings werden die Schlüssel über die Methode `OpenSubKey()` in der Standardeinstellung immer nur zum Lesen geöffnet. Wenn Sie die Daten auch speichern möchten, müssen Sie beim Aufruf der Methode `OpenSubKey()` als zweites Argument noch den Wert `true` angeben. Die folgende Anweisung würde den Schlüssel also sowohl zum Lesen als auch zum Schreiben öffnen:

```
regSchluessel = Registry.CurrentUser.  
OpenSubKey("Software\\RegistryDemo", true);
```

Etwas einfacher ist ein schreibender Zugriff auf einen Eintrag, wenn Sie den Schlüssel über die Methode `CreateSubKey()`⁵ öffnen. Diese Methode öffnet nicht nur einen vorhandenen Schlüssel zum Lesen und Schreiben, sondern legt ihn auch automatisch an, falls er nicht vorhanden ist.

Damit alles seine Ordnung hat, sollten Sie die Registrierung nach den Zugriffen auch wieder schließen. Dazu verwenden Sie die Methode `Close()` von `RegistryKey`.



Bitte beachten Sie:

Das Schließen ist vor allem nach schreibenden Zugriffen sehr wichtig. Denn nur beim Schließen der Registrierung können Sie sicher sein, dass auch alle Änderungen tatsächlich übernommen werden.

Microsoft empfiehlt ausdrücklich, alle Ressourcen der Klasse `RegistryKey` direkt nach dem Zugriff wieder freizugeben. Dazu können Sie entweder die Methode `Dispose()` in einer `try ... catch ... finally`-Konstruktion aufrufen oder – sehr viel einfacher – eine `using`-Konstruktion benutzen.



Über eine `using`-Konstruktion legen Sie die Lebensdauer eines Objekts fest.

4. `SetValue` lässt sich mit „Setze Wert“ übersetzen.

5. `CreateSubKey` bedeutet übersetzt so viel wie „Erzeuge untergeordneten Schlüssel“.

Was sich vielleicht kompliziert anhört, ist ganz einfach. Sie geben hinter dem Schlüsselwort `using` in runden Klammern die Anweisung zum Erzeugen des Objekts an. Dann folgt in geschweiften Klammern der Anweisungsblock, für den dieses Objekt gelten soll. Die Anweisungen

```
using (RegistryKey regSchluessel =  
Registry.CurrentUser.OpenSubKey("Software\\RegistryDemo"))  
{  
...  
}
```

erzeugen so zum Beispiel eine Instanz `regSchluessel` der Klasse `RegistryKey`. Diese Instanz existiert nur für die Anweisungen, die im folgenden Anweisungsblock stehen. Beim Verlassen des Blocks werden die Ressourcen freigegeben und auch die Registrierung wieder geschlossen.

Bitte beachten Sie:

Die Anweisungen für das Erzeugen der Instanz in den runden Klammern dürfen Sie nicht mit einem Semikolon abschließen.



Schauen wir uns den Umgang mit der Registrierung nun an einigen praktischen Beispielen an.

1.3 Ein einfaches Beispiel

Beginnen wir mit einem ganz einfachen Programm. Wir wollen den Eintrag aus einem Eingabefeld in der Registrierung ablegen und über ein Label wieder zurücklesen.

Legen Sie bitte eine neue Windows Forms-Anwendung an. Setzen Sie dann in das Formular ein Label, ein Eingabefeld und zwei Schaltflächen. Über die eine Schaltfläche starten wir das Lesen aus der Registrierung und über die andere das Schreiben.

Die beiden Methoden für das Anklicken der beiden Schaltflächen finden Sie im Code 1.2:

```
private void ButtonLesen_Click(object sender, EventArgs e)  
{  
    //den Schlüssel  
    //HKEY_CURRENT_USER\Software\RegistryDemo öffnen  
    //bitte in einer Zeile eingeben  
    using (RegistryKey regSchluessel =  
        Registry.CurrentUser.OpenSubKey("Software\\RegistryDemo"))  
    {  
        //wenn der Schlüssel nicht vorhanden ist, eine  
        //Meldung ausgeben  
        if (regSchluessel == null)  
        {  
            //bitte in einer Zeile eingeben  
            MessageBox.Show("Der Schlüssel konnte nicht  
                geöffnet werden!");  
            label1.Text = "Nicht vorhanden";  
        }  
    }  
}
```

```

        else
            //sonst den Eintrag lesen und label1 zuweisen
            //bitte in einer Zeile eingeben
            label1.Text = Convert.ToString(regSchluessel.
                GetValue("Eintrag"));
    }

private void ButtonSchreiben_Click(object sender, EventArgs e)
{
    //den Schlüssel HKEY_CURRENT_USER\Software\RegistryDemo
    //anlegen bzw. öffnen
    //bitte in einer Zeile eingeben
    using (RegistryKey regSchluessel =
        Registry.CurrentUser.CreateSubKey("Software\\RegistryDemo"))
    {
        //den Wert aus dem Eingabefeld in den Eintrag schreiben
        regSchluessel.SetValue("Eintrag", textBox1.Text);
    }
}

```

Code 1.2: Die Methoden ButtonLesen_Click() und ButtonSchreiben_Click()

Hinweis:

Die Namen der Methoden hängen vom Namen der Schaltflächen ab. Auch die Namen für das Label und das Eingabefeld müssen Sie gegebenenfalls an Ihr Projekt anpassen.



Bitte beachten Sie:

Denken Sie daran, den Namensraum `Microsoft.Win32` bekannt zu machen. Ergänzen Sie dazu zu Beginn des Quelltextes von `Form1` die Anweisung `using Microsoft.Win32;` unterhalb der anderen `using`-Anweisungen. Wenn Sie diese Anweisung weglassen, müssen Sie den Namensraum immer explizit in den einzelnen Anweisungen für die Klassen `Registry` und `RegistryKey` angeben.

Beim Lesen überprüfen wir mit der Anweisung

```
if (regSchluessel == null)
```

ob der Schlüssel geöffnet werden konnte. Wenn das nicht der Fall ist, lassen wir eine Meldung ausgeben und setzen den Text in dem Label.

Probieren Sie die Anwendung jetzt aus. Versuchen Sie dazu zunächst einmal, den Wert direkt nach dem Start zu lesen. Hier muss die Meldung erscheinen, dass der Schlüssel nicht geöffnet werden konnte.

Nach dem ersten Schreiben sollte dann beim Lesen auch der Wert im Label angezeigt werden.

Neben den Daten aus Steuerelementen können Sie zum Beispiel auch die Position und die Größe eines Formulars in der Registrierung speichern und wieder auslesen. So können Sie zum Beispiel dafür sorgen, dass ein Formular immer in der Größe und an der Position erscheint, die es beim Verlassen der Anwendung hatte. Dazu speichern Sie beim

Schließen des Formulars im Ereignis **FormClosed** die aktuellen Daten in der Registrierung und lesen sie zum Beispiel beim Öffnen des Formulars im Ereignis **Shown** wieder ein.

Die entsprechenden Methoden könnten so aussehen:

```
private void Form1_FormClosed(object sender,
FormClosedEventArgs e)
{
    //beim Schließen speichern wir die Position und die
    //Größe in der Registrierung
    //den Schlüssel HKEY_CURRENT_USER\Software\RegistryDemo
    //anlegen bzw. öffnen
    //bitte in einer Zeile eingeben
    using (RegistryKey regSchluessel =
Registry.CurrentUser.CreateSubKey("Software\\RegistryDemo"))
    {
        //die Werte speichern
        regSchluessel.SetValue("Top", this.Top);
        regSchluessel.SetValue("Left", this.Left);
        regSchluessel.SetValue("Height", this.Height);
        regSchluessel.SetValue("Width", this.Width);
    }
}

private void Form1_Shown(object sender, EventArgs e)
{
    //beim Anzeigen lesen wir die Daten wieder zurück
    //den Schlüssel HKEY_CURRENT_USER\Software\RegistryDemo öffnen
    //bitte in einer Zeile eingeben
    using (RegistryKey regSchluessel =
Registry.CurrentUser.OpenSubKey("Software\\RegistryDemo"))
    {
        //wenn der Schlüssel nicht vorhanden ist, Meldung ausgeben
        if (regSchluessel == null)
            //bitte in einer Zeile eingeben
            MessageBox.Show("Der Schlüssel konnte nicht
geöffnet werden!");
        else
        {
            //sonst die Einträge lesen und zuweisen
            this.Top = Convert.ToInt32(regSchluessel.GetValue("Top"));
            //bitte jeweils in einer Zeile eingeben
            this.Left = Convert.ToInt32
(regSchluessel.GetValue("Left"));
            this.Height = Convert.ToInt32
(regSchluessel.GetValue("Height"));
            this.Width = Convert.ToInt32
(regSchluessel.GetValue("Width"));
        }
    }
}
```

Code 1.3: Das Speichern und Lesen der Positionsdaten für das Formular

Hinweis:

In der Methode `Form1_Shown()` überprüfen wir lediglich, ob der Schlüssel `RegistryDemo` vorhanden ist, und kontrollieren nicht, ob die einzelnen Einträge für die Positionsdaten auch tatsächlich gelesen werden können. Beim ersten Start des Programms kann es daher passieren, dass das Formular sehr klein oben links auf dem Bildschirm erscheint, da das Einlesen keine brauchbaren Werte liefert.



Im praktischen Einsatz finden Sie die beiden Methoden im Projekt **RegDemo01** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Sehen Sie nach dem Test des Programms auch einmal über den Registrierungs-Editor nach, wo die Einträge genau abgelegt wurden. Sie sollten sich im Zweig `HKEY_CURRENT_USER\Software\RegistryDemo` befinden.

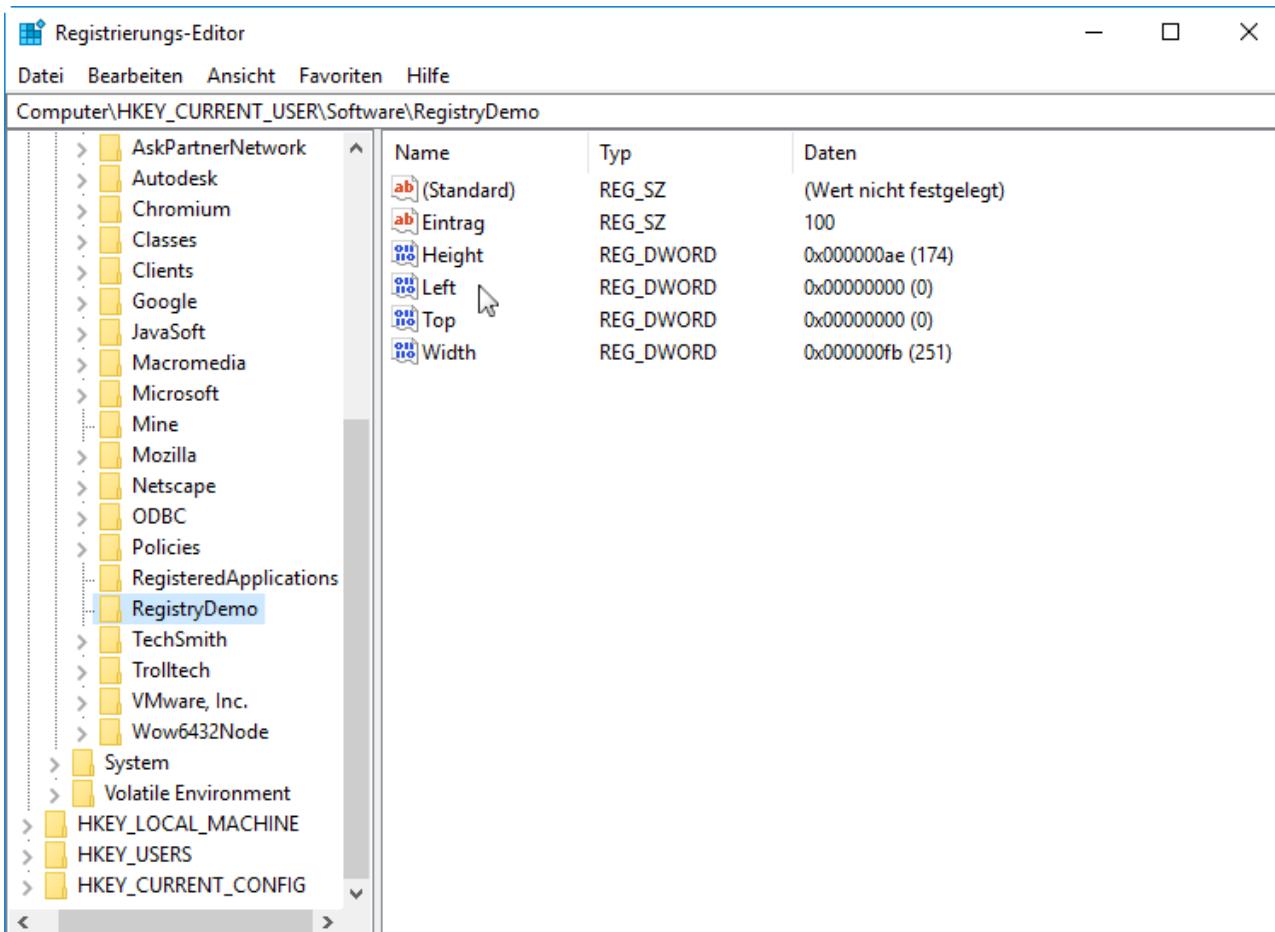


Abb. 1.4: Die neuen Schlüssel in der Registrierung

Hinweis:

Über die Funktion **Löschen** im Kontext-Menü des Schlüssels können Sie den kompletten Schlüssel auch wieder aus der Registrierung entfernen. Achten Sie dabei aber sehr sorgfältig darauf, dass Sie nicht versehentlich einen anderen Schlüssel löschen.

1.4 Eine Kennwortabfrage

Schauen wir uns den Einsatz der Registrierung nun an einem etwas anspruchsvollerem Beispiel an.

Eine Anwendung soll nur dann gestartet werden können, wenn der Anwender ein Kennwort eingibt. Dieses Kennwort kann der Anwender beim ersten Start der Anwendung selbst festlegen. Es wird dann in der Registrierung gespeichert und bei allen weiteren Starts der Anwendung abgefragt. Auf Wunsch soll der Anwender das Kennwort auch wieder aus der Registrierung löschen können.

Das Ganze lässt sich recht einfach über eine Anwendung mit insgesamt drei Formularen realisieren. Das erste Formular steht dabei wie gewohnt für die eigentliche Anwendung. Das zweite Formular verwenden wir für die erste Eingabe des Kennworts und das dritte Formular für die Eingabe, wenn bereits ein Kennwort vorhanden ist.

Direkt nach dem Start des Programms überprüfen wir, ob bereits ein Eintrag für das Kennwort in der Registrierung vorhanden ist. Wenn das der Fall ist, lassen wir das Formular zur Kennworteingabe anzeigen. Andernfalls soll das Formular zum Festlegen des Kennworts erscheinen.

Legen Sie jetzt bitte eine neue leere Windows Forms-Anwendung an. Erstellen Sie dann zwei weitere Formulare. Diese Formulare könnten zum Beispiel so aussehen:

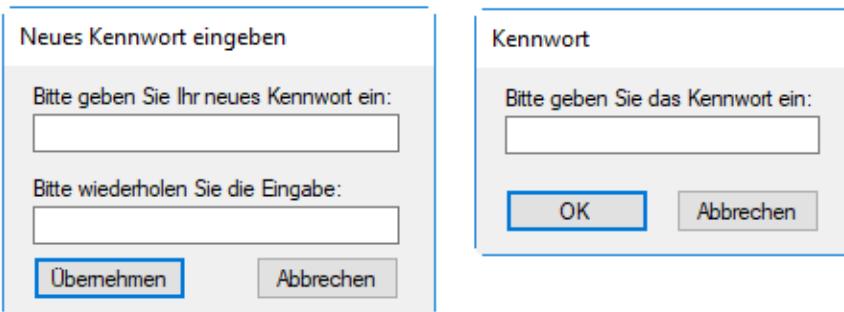


Abb. 1.5: Das Formular zur Kennworteingabe (rechts) und das Formular zum Festlegen des Kennworts (links)

Hinweis:

In unserem Beispielprogramm verwenden wir für das Formular zur Kennworteingabe den Namen `FrmKennwortDialog`. Das andere Formular hat den Namen `FrmNeuesKennwort`.

Damit die Formulare nur über die Schaltflächen geschlossen werden können, sorgen Sie bitte dafür, dass die Standardsymbole in der Titelleiste und auch das Systemmenü nicht angezeigt werden. Setzen Sie dazu die Eigenschaft `ControlBox` auf `False`. Damit die Formulare immer im Vordergrund angezeigt werden, sollten Sie außerdem die Eigenschaft `TopMost` in `True` ändern.

Tipp:

Damit die Eingaben in den Kennwortfeldern nicht im Klartext erscheinen, können Sie die Eigenschaft **UseSystemPasswordChar**⁶ der Textboxen auf `True` setzen. Dann wird für jede Eingabe das Zeichen angezeigt, das auch Windows für Kennwörter benutzt. Über die Eigenschaft **PasswordChar** können Sie auch ein eigenes Zeichen festlegen. Dieses Zeichen wird aber nur dann benutzt, wenn die Eigenschaft **UseSystemPasswordChar** den Wert `False` hat.

Im Ereignis **Shown** der eigentlichen Anwendung prüfen wir mit der Methode `OpenSubKey()`, ob bereits ein Eintrag für das Kennwort in der Registrierung vorhanden ist. Abhängig vom Ergebnis der Überprüfung lassen wir dann entweder das eine oder das andere Formular modal anzeigen. Damit die eigentliche Anwendung dabei nicht im Hintergrund aktiv ist, blenden wir das Formular der Anwendung direkt wieder aus und lassen es erst wieder anzeigen, wenn die anderen Dialoge geschlossen wurden.

Die vollständige Methode für das Ereignis **Shown** der Anwendung sieht dann so aus:

```
private void Form1_Shown(object sender, EventArgs e)
{
    //bitte in einer Zeile eingeben
    using (RegistryKey regSchluessel =
    Registry.CurrentUser.OpenSubKey("Software\\RegistryDemo2"))
    {
        //das Formular der eigentlichen Anwendung ausblenden
        this.Hide();
        //wenn es den Schlüssel nicht gibt, das Formular
        //zum Festlegen des Kennworts anzeigen
        if (regSchluessel == null)
        {
            FrmNeuesKennwort dialogNeu = new FrmNeuesKennwort();
            dialogNeu.ShowDialog();
        }
        //sonst das Formular zur Eingabe des Kennworts anzeigen
        else
        {
            FrmKennwortDialog dialogEingabe = new FrmKennwortDialog();
            dialogEingabe.ShowDialog();
        }
    }
    //das Formular der eigentlichen Anwendung wieder zeigen
    this.Show();
}
```

Code 1.4: Die Methode `Form1_Shown()`

Hinweis:

Denken Sie daran, dass Sie in allen Formularen den Namensraum `Microsoft.Win32` bekannt machen müssen.

6. `UseSystemPasswordChar` lässt sich mit „Benutze das Kennwortzeichen des Systems“ übersetzen.

Besonderheiten gibt es in der Methode eigentlich nicht.

Auch die Anweisungen für die beiden anderen Formulare sind nicht weiter kompliziert. Im Formular für die Vergabe eines neuen Kennworts überprüfen wir beim Anklicken der Schaltfläche **Übernehmen** zunächst einmal, ob sich im ersten Feld überhaupt ein Eintrag befindet. Wenn das der Fall ist, kontrollieren wir noch, ob die Einträge im ersten und im zweiten Eingabefeld identisch sind, und schreiben das neue Kennwort dann in die Registrierung. Dabei benutzen wir den Zweig **HKEY_CURRENT_USER\RegistryDemo2** und den Eintrag **Kennwort**. Danach wird das Formular für die Kennworteingabe geschlossen.

Die Anweisungen für das Ereignis **Click** der Schaltfläche **Übernehmen** im Formular für die Vergabe eines neuen Kennworts könnten so aussehen:

```
//ist das erste Feld leer?  
if (textBoxKennwort1.Text == string.Empty)  
{  
    //bitte in einer Zeile eingeben  
    MessageBox.Show("Das Kennwort muss mindestens ein  
    Zeichen lang sein!");  
    //den Fokus auf das erste Feld setzen  
    textBoxKennwort1.Select();  
    return;  
}  
//stimmen die beiden Einträge überein?  
if (textBoxKennwort1.Text == textBoxKennwort2.Text)  
{  
    //den Wert in die Registrierung schreiben  
    //bitte in einer Zeile eingeben  
    using (RegistryKey regSchluessel = Registry.CurrentUser.  
    CreateSubKey("Software\\RegistryDemo2"))  
    {  
        regSchluessel.SetValue("Kennwort", textBoxKennwort1.Text);  
    }  
    //das Formular schließen  
    Close();  
}  
else  
{  
    MessageBox.Show("Die Eingaben stimmen nicht überein!");  
    //den Fokus auf das zweite Eingabefeld setzen  
    textBoxKennwort2.Select();  
}
```

Code 1.5: Die Anweisung für das Ereignis **Click** der Schaltfläche **Übernehmen**

Hinweis:

Für das erste Eingabefeld haben wir in dem Beispiel den Namen `textBoxKennwort1` benutzt und für das zweite Eingabefeld den Namen `textBoxKennwort2`.

Beim Anklicken der Schaltfläche **Abbrechen** im Formular zum Festlegen eines neuen Kennworts soll nicht nur das aktive Formular geschlossen, sondern die gesamte Anwendung beendet werden. Dazu verwenden wir die Anweisung `Application.Exit()`. Sie hat eine ähnliche Wirkung wie die Methode `Close()` für das Hauptformular einer Anwendung.

Kommen wir nun zum Formular zur Eingabe des Kennworts.

Hier lesen wir beim Anklicken der Schaltfläche **OK** zunächst den Wert aus der Registrierung ein und vergleichen ihn mit der Eingabe in dem Feld. Wenn die beiden Werte übereinstimmen, blenden wir die eigentliche Anwendung ein und schließen das Formular für die Eingabe wieder.

Damit der Anwender sich auch einen Tippfehler erlauben kann, lassen wir insgesamt drei falsche Eingaben zu. Falls auch danach nicht das richtige Kennwort eingegeben wurde, schließen wir die gesamte Anwendung über die Anweisung `Application.Exit()`.

Die entsprechenden Anweisungen könnten dann so aussehen:

```
...
//ein eigenes Feld zum Zählen
//es wird im Konstruktor auf 0 gesetzt
int zaehler;
...

private void ButtonOK_Click(object sender, EventArgs e)
{
    string kennwort;
    //den Wert aus der Registrierung lesen und in
    //kennwort ablegen
    //bitte jeweils in einer Zeile eingeben
    using (RegistryKey regSchluessel = Registry.CurrentUser.
        OpenSubKey("Software\\RegistryDemo2"))
    {
        kennwort = Convert.ToString
    }
    //stimmen die Eingabe und das Kennwort überein?
    if (textBoxKennwort.Text == kennwort)
        //das Formular schließen
        Close();
    else
    {
        MessageBox.Show("Falsches Kennwort!");
        zaehler=zaehler+1;
        //drei falsche Eingaben?
        if (zaehler == 3)
        {
            MessageBox.Show("Die Anwendung wird geschlossen!");
            //die gesamte Anwendung beenden
            Application.Exit();
        }
    }
}
```

```

        else
            //den Fokus auf das Eingabefeld setzen
            textBoxKennwort.Select();
    }

private void ButtonAbbrechen_Click(object sender,
EventArgs e)
{
    //die gesamte Anwendung beenden
    Application.Exit();
}

```

Code 1.6: Die Methoden für das Formular zur Kennworteingabe**Hinweis:**

`zaehler` ist ein Feld der Klasse für das Formular zur Kennworteingabe. Sie sollten es im Konstruktor auf den Wert 0 setzen.

Das Eingabefeld hat in unserem Beispiel den Namen `textBoxKennwort`. Die beiden Schaltflächen tragen die Namen `buttonOK` und `buttonAbbrechen`.

Jetzt fehlt nur noch eine Funktion zum Löschen des Kennworts. Dazu bauen wir im Formular der eigentlichen Anwendung eine Schaltfläche ein und rufen über diese Schaltfläche die Methode `DeleteSubKeyTree()`⁷ für `Registry.CurrentUser` auf. Diese Methode löscht einen kompletten Zweig in der Registrierung mit allen untergeordneten Zweigen und Einträgen. Als Argument übergeben Sie den Namen des gewünschten Zweigs.

Die Methode für das Anklicken der Schaltfläche in der eigentlichen Anwendung sieht dann so aus:

```

private void ButtonLoeschen_Click(object sender,
EventArgs e)
{
    //den gesamten Zweig in der Registrierung löschen
    //bitte in einer Zeile eingeben
    Registry.CurrentUser.DeleteSubKeyTree("Software\\"
    RegistryDemo2");
}

```

Code 1.7: Das Löschen des kompletten Zweigs**Bitte beachten Sie:**

Gehen Sie beim Löschen von Zweigen in der Registrierung sehr sorgfältig vor. Es erfolgen keinerlei Sicherheitsabfragen.



7. `DeleteSubKeyTree` lässt sich mit „Lösche den Zweig eines untergeordneten Schlüssels“ übersetzen.

Übernehmen Sie die Anweisungen für die Formulare aus den einzelnen Codes. Probieren Sie das Programm dann aus. Sie werden sehen, beim ersten Start erscheint der Dialog zur Eingabe eines neuen Kennworts. Bei allen folgenden Aufrufen wird das Kennwort abgefragt.



Das vollständige Projekt für dieses Beispiel finden Sie unter dem Namen **RegDemo02** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Hinweis:

Das Programm ist nicht sonderlich stabil, da wir beim Starten lediglich überprüfen, ob der Zweig vorhanden ist, und nicht kontrollieren, ob es den Eintrag **Kennwort** gibt. Sie könnten also mit dem Registrierungs-Editor den Zweig per Hand anlegen und das Programm so in die Irre führen.

Wenn Sie ganz sicher gehen wollen, können Sie das Programm ja noch so erweitern, dass Sie beim Start nicht nur den Zweig überprüfen, sondern zusätzlich auch versuchen, den Eintrag **Kennwort** zu lesen. Falls das Lesen scheitert, wird ebenfalls der Wert `null` zurückgegeben.

Das Programm lässt sich auch sehr einfach austricksen, wenn das Formular zur Kennworteingabe über die Tastenkombination **Alt + F4** geschlossen wird. Dann erscheint direkt das Fenster der eigentlichen Anwendung. Das können Sie zum Beispiel verhindern, indem Sie auch beim Schließen des Formulars überprüfen, ob das richtige Kennwort eingegeben wurde.

Auch das Schließen über `Application.Exit()` kann zu Problemen führen, da möglicherweise das Hauptformular angezeigt werden soll, aber bereits freigegeben ist. Stabiler ist es, wenn Sie die Anwendung nicht aus den untergeordneten Formularen schließen, sondern einen Wert zurückliefern lassen, der markiert, ob die Anwendung beendet oder ausgeführt werden soll. Das eigentliche Schließen erfolgt dann aus dem Hauptformular. Wenn Sie Lust haben, können Sie die Anwendung ja in Eigenregie entsprechend umbauen.

1.5 Eine Listenanzeige für die Registrierung

Neben dem gezielten Zugriff auf einzelne Schlüssel und Einträge können Sie über die Methoden der Klasse `RegistryKey` auch ganze Gruppen verarbeiten.

Schauen wir uns auch dazu ein Beispiel an. In einer Anwendung sollen alle Schlüssel aus dem Hauptschlüssel **HKEY_CURRENT_USER**, alle Schlüssel aus dem Zweig **HKEY_CURRENT_USER\Software** und dann alle Einträge aus dem Zweig **HKEY_CURRENT_USER\Software\Registrierungsdemo** angezeigt werden. Die Anzeige soll dabei jeweils in einem eigenen Listenfeld erfolgen.

Die fertige Anwendung könnte so aussehen:

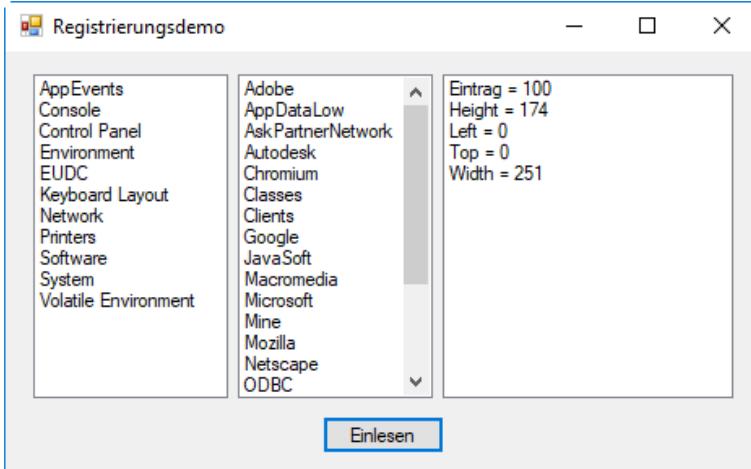


Abb. 1.6: Eine Listenanzeige für die Registrierung

Für die Umsetzung verwenden wir die Methoden `GetSubKeyNames()` und `GetValueNames()` der Klasse `RegistryKey`. Die Methode `GetSubKeyNames()` liefert eine Liste mit Zeichenketten für alle Unterschlüssel zurück und die Methode `GetValueNames()` entsprechend eine Liste mit allen Einträgen. Diese Listen können wir sehr einfach über eine `foreach`-Schleife verarbeiten und die Werte in die jeweiligen Listenfelder eintragen.

Legen Sie jetzt bitte eine neue Windows Forms-Anwendung an. Fügen Sie in das Formular drei Listenfelder und eine Schaltfläche ein. Beim Anklicken der Schaltfläche lassen Sie dann die Anweisungen aus dem folgenden Code ausführen. Was dort genau passiert, erklären wir Ihnen wie gewohnt im Anschluss.

```
//ein Array mit Zeichenketten für die Schlüssel
string[] regSchluesselListe;
//ein Array mit Zeichenketten für die Einträge
string[] regEintragListe;
//den Inhalt der drei Listenfelder löschen
listBox1.Items.Clear();
listBox2.Items.Clear();
listBox3.Items.Clear();

//die Schlüssel aus HKEY_CURRENT_USER holen
//bitte in einer Zeile eingeben
regSchluesselListe =
Registry.CurrentUser.GetSubKeyNames();
//und in das erste Listenfeld eintragen
foreach (string eintrag in regSchluesselListe)
    listBox1.Items.Add(eintrag);
```

```

//den Schlüssel Software öffnen
//bitte in einer Zeile eingeben
using (RegistryKey regSchluessel =
Registry.CurrentUser.OpenSubKey("Software"))
{
    //und jetzt alle Unterschlüssel für Software lesen
    regSchluesselListe = regSchluessel.GetSubKeyNames();
}
//in das zweite Listenfeld eintragen
foreach (string eintrag in regSchluesselListe)
    listBox2.Items.Add(eintrag);

//den Schlüssel \Software\Registrierungsdemo öffnen
//bitte in einer Zeile eingeben
using (RegistryKey regSchluessel = Registry.CurrentUser.
OpenSubKey("Software\\Registrierungsdemo"))
{
    //die Einträge lesen
    regEintragListe = regSchluessel.GetValueNames();
    //die Namen und die Werte in das dritte Listenfeld schreiben
    foreach (string eintrag in regEintragListe)
        //bitte in einer Zeile eingeben
        listBox3.Items.Add(eintrag + " = " +
Convert.ToString(regSchluessel.GetValue(eintrag)));
}

```

Code 1.8: Das Beschaffen der Einträge für die Listenanzeige

Zunächst einmal vereinbaren wir zwei Arrays vom Typ `string` für die Listen. Das übernehmen die Zeilen

```

string[] regSchluesselListe;
string[] regEintragListe;

```

Danach löschen wir zur Sicherheit alle Einträge in den drei Listenfeldern. Das ist zwar direkt nach dem Start des Programms eigentlich nicht erforderlich, sorgt aber dafür, dass beim erneuten Klicken auf die Schaltfläche die Einträge nicht unten an die Listen angehängt werden.

Im nächsten Schritt beschaffen wir uns sämtliche Unterschlüssel aus dem Hauptschlüssel **HKEY_CURRENT_USER** und lassen sie im ersten Listenfeld anzeigen. Das erledigen die Anweisungen

```

regSchluesselListe = Registry.CurrentUser.GetSubKeyNames();
foreach (string eintrag in regSchluesselListe)
    listBox1.Items.Add(eintrag);

```

Bitte achten Sie dabei darauf, dass es sich bei `regSchluesselListe` nicht um eine Instanz der Klasse `RegistryKey` handelt, sondern um ein Array vom Typ `string`. Denn die Methode `GetSubKeyNames()` liefert uns ja keinen einzelnen Schlüssel, sondern sämtliche vorhandene Schlüssel in Listenform.

Danach öffnen wir den Schlüssel **Software**, beschaffen für diesen Schlüssel ebenfalls alle Unterschlüssel und lassen sie im zweiten Listenfeld anzeigen. Dabei gibt es keine Besonderheiten.

Interessanter wird dann wieder das Einlesen der Einträge. Hier öffnen wir zunächst den Schlüssel **HKEY_CURRENT_USER\Software\Registrierungsdemo** und beschaffen uns mit der Methode `GetValueNames()` die Namen aller Einträge. Das erledigen die beiden Anweisungen

```
using (RegistryKey regSchluessel =  
Registry.CurrentUser.OpenSubKey("Software\\Registrierungsdemo"))  
{  
    regEintragListe = regSchluessel.GetValueNames();  
}
```

Danach schreiben wir mit der `foreach`-Schleife

```
foreach (string eintrag in regEintragListe)  
    listBox3.Items.Add(eintrag + " = " +  
        Convert.ToString(regSchluessel.GetValue(eintrag)));
```

den Namen jedes Eintrags in das dritte Listenfeld und hängen zusätzlich den Wert des Eintrags an. Diesen Wert erhalten wir über den Ausdruck `Convert.ToString(regSchluessel.GetValue(eintrag))`.

Wenn Sie das Programm jetzt ausprobieren und die Ergebnisse mit der Anzeige im Registrierungs-Editor vergleichen, fallen Ihnen vielleicht zwei Sachen auf:

Zum einen ist die Reihenfolge der Anzeige nicht identisch. Das liegt einfach daran, dass die Anzeige im Registrierungs-Editor alphabetisch sortiert erfolgt, die Einträge aber nicht in jedem Fall auch tatsächlich in dieser Reihenfolge in der Registrierung abgelegt sind. Diese Abweichung lässt sich sehr leicht beheben, indem Sie die Eigenschaft **Sorted** der Listenfelder jeweils auf `True` setzen.

Zum anderen werden im dritten Listenfeld nur die Einträge des Schlüssels angezeigt, aber keine eventuell vorhandenen weiteren Unterschlüssel. Das liegt daran, dass die Methode `GetValueNames()` nur echte Einträge verarbeitet, aber keine Unterschlüssel. Wenn Sie auch die Unterschlüssel anzeigen lassen möchten, müssten Sie gegebenenfalls vor der Anzeige der Einträge noch einmal selbst die Methode `GetSubKeyNames()` aufrufen.

Sollte bei Ihnen kein Eintrag im dritten Listenfeld angezeigt werden, liegen unter Umständen gar keine Einträge für diesen Zweig vor. Lassen Sie dann einen anderen Zweig aus der Registrierung anzeigen. Da wir in dem Beispiel nicht schreibend auf die Registrierung zugreifen, ist das ohne Probleme möglich. Sehen Sie einfach mit dem Registrierungs-Editor nach, in welchem Zweig auf der dritten Ebene einzelne Einträge vorhanden sind.

Das vollständige Projekt für dieses Beispiel mit den beiden Erweiterungen finden Sie unter dem Namen **RegDemo03** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.



Hinweis:

Die Listenanzeige ist bisher nicht sonderlich flexibel, da ja nur feste Schlüssel verarbeitet werden. Eine Erweiterung des Programms wartet aber bei den Einsendeaufgaben auf Sie.

Die Klasse `RegistryKey` kennt noch zahlreiche weitere Methoden, die wir Ihnen hier nicht einzeln vorstellen wollen. Detaillierte Informationen können Sie in der Hilfe nachschlagen.

Bevor Sie sich nun an eigene Experimente begeben, noch einmal die eindringliche Warnung: Bereits kleine Programmierfehler beim Zugriff können dafür sorgen, dass Teile der Registrierung oder sogar die komplette Registrierung unbrauchbar werden. Im schlimmsten Fall können Sie dann Windows nicht mehr starten und haben keinen direkten Zugriff mehr auf Ihre Daten auf dem Rechner.

Erstellen Sie daher vor Ihren Versuchen immer eine komplette Sicherungskopie auf einem externen Datenträger – zum Beispiel einer mobilen Festplatte. Ganz sicher gehen Sie, wenn Sie für die Experimente einen eigenen Rechner benutzen, den Sie nur für Experimente verwenden. Hier können Sie im Zweifelsfall immer noch Windows komplett neu installieren. Sehr gut geeignet zum Test sind auch virtuelle Maschinen. Hier können Sie ebenfalls gefahrlos experimentieren.

So viel zum Zugriff auf die Registrierung. Im nächsten Kapitel werden wir uns mit XML-Dateien beschäftigen.

Zusammenfassung

Die Registrierung ist eine Art Datenbank, in der alle wichtigen Einstellungen von Windows zentral zusammenlaufen.

Die Registrierung arbeitet mit Schlüsseln in einer hierarchischen Struktur. Zu den Schlüsseln können Einträge mit Werten angelegt werden.

Die Daten in der Registrierung werden in binärer Form gespeichert. Die Anzeige und die Bearbeitung erfolgen mit dem Registrierungs-Editor von Windows.

Über die Klassen `Registry` und `RegistryKey` können Sie auf Einträge in der Registrierung zugreifen.

Arbeiten Sie beim Zugriff auf die Registrierung äußerst sorgfältig. Löschen oder ändern Sie auf keinen Fall Einträge, wenn Sie nicht genau wissen, wofür der Eintrag gut ist.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 In welchem Hauptschlüssel der Registrierung werden Daten für den aktuell angemeldeten Anwender gespeichert?

- 1.2 Aus welchen Bestandteilen setzt sich ein Eintrag in der Registrierung zusammen?

- 1.3 Welche statische Instanz der Klasse `RegistryKey` entspricht dem Haupt-schlüssel **HKEY_LOCAL_MACHINE** der Registrierung?

- 1.4 Mit welcher Eigenschaft eines Eingabefelds können Sie dafür sorgen, dass die eingegebenen Zeichen nicht im Klartext angezeigt werden?

- 1.5 Mit welcher Methode können Sie die komplette Anwendung aus einem beliebigen Formular der Anwendung beenden?

- 1.6 Sie wollen über eine Instanz `registrierung` der Klasse `RegistryKey` auf den Schlüssel **HKEY_CURRENT_USER\Software** zugreifen. Welche Anwei-sungen sind dazu erforderlich?

- 1.7 Mit welcher Methode der Klasse `RegistryKey` schreiben Sie einen Wert in die Registrierung? Mit welcher Methode lesen Sie einen Wert aus der Regis-trierung? Worauf müssen Sie beim Lesen besonders achten?

2 Arbeiten mit XML-Dateien

In diesem Kapitel beschäftigen wir uns mit dem Lesen und Schreiben von Daten in XML-Dateien.



XML steht für eXtensible Markup Language^{a)}. Es handelt sich um ein textbasiertes, strukturiertes Format für die Beschreibung von Daten.

a) *Extensible Markup Language* bedeutet übersetzt „erweiterbare Auszeichnungssprache“.

XML ist lose mit der Auszeichnungssprache HTML verwandt und stellt sämtliche Informationen als normalen Text dar. Anders als das HTML-Format, das vor allem für die Anzeige in Browern gedacht ist, können XML-Dokumente auch mit vielen Standardanwendungen erzeugt und geöffnet werden. XML eignet sich damit ideal zum anwendungsübergreifenden Austausch von Daten.

Bevor wir uns an die praktische Arbeit mit XML-Dateien machen, wollen wir uns zunächst einmal kurz den Aufbau einer solchen Datei ansehen.

2.1 Der Aufbau von XML-Dateien

Sämtliche Informationen in einer XML-Datei werden als einfacher Text in einer hierarchischen Baumstruktur gespeichert. Diese Struktur besteht aus einem **Wurzelement** und nahezu beliebigen **Unterelementen** – den **Knoten**.

Für die Beschreibung und Strukturierung werden **Tags**⁸ benutzt. Sie stehen in spitzen Klammern < >. Jedes Tag besteht in der Regel aus dem Start-Tag und dem End-Tag. Das End-Tag beginnt dabei immer mit dem Schrägstrich /. Zwischen dem Start- und dem End-Tag können dann Texte oder auch weitere Tags stehen.



Bitte beachten Sie:

Jedes Tag muss korrekt beendet werden. Das heißt, zu jedem Start-Tag muss es auch ein passendes End-Tag geben.

Eine typische Zeile in einer XML-Datei könnte zum Beispiel so aussehen:

Start-Tag	Text	End-Tag
<code><eintrag></code>	<code>Ich bin der erste Eintrag</code>	<code></eintrag></code>

Abb. 2.1: Eine typische Zeile in einer XML-Datei

Wie Sie die Tags benennen, ist relativ beliebig. Lediglich der Name `xml` ist reserviert und kann nicht verwendet werden. Außerdem muss der Name mit einem Buchstaben oder dem Unterstrich `_` beginnen. Achten Sie aber in jedem Fall darauf, dass Sie den Namen des Tags im Start- und End-Tag gleich schreiben. Dabei werden auch Groß- und Kleinschreibung berücksichtigt. Das End-Tag darf sich also lediglich durch den Schrägstrich / vom Start-Tag unterscheiden.

8. Tag bedeutet übersetzt so viel wie „Kennzeichen“.

Damit ein XML-Dokument korrekt erkannt wird, wird es mit der **XML-Deklaration** – einem Teil des **Prologs** – eingeleitet. Sie beschreibt unter anderem die verwendete XML-Version und den verwendeten Zeichensatz. Die Deklaration beginnt mit den Zeichen <?xml und endet mit den Zeichen ?>.

```
Version          Zeichensatz
|               |
<?xml version="1.0" encoding="UTF-8"?>
```

Abb. 2.2: Die Deklaration eines XML-Dokuments

Bitte beachten Sie:

Die Angaben zur Version und zum Zeichensatz müssen in Anführungszeichen gesetzt werden. Auch die Reihenfolge ist verbindlich. Sie können also nicht erst den Zeichensatz angeben und dann die Version.



Neben Tags und Texten kann ein XML-Dokument auch Kommentare enthalten. Sie werden mit <!-- --> eingeleitet und enden mit -->.

Ein vollständiges einfaches XML-Dokument könnte zum Beispiel so aussehen:

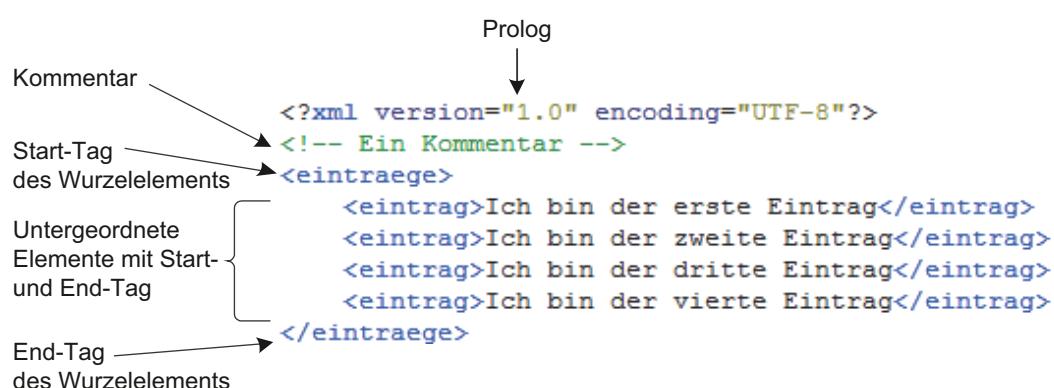


Abb. 2.3: Ein vollständiges XML-Dokument

<eintraege> bildet dabei das Wurzelement. Es enthält insgesamt vier untergeordnete Knoten <eintrag> mit unterschiedlichen Werten.

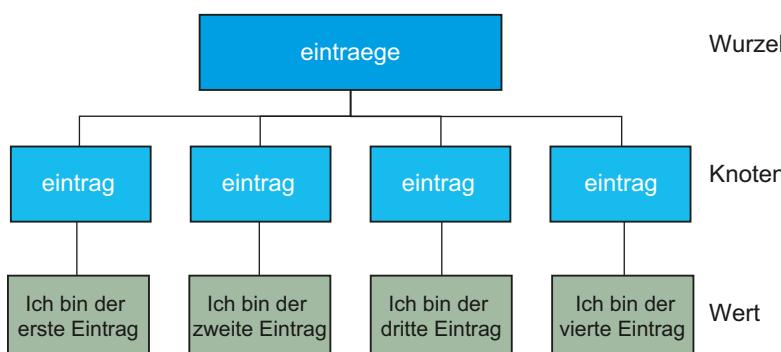


Abb. 2.4: Die hierarchische Struktur der XML-Datei

Hinweis:

XML bietet noch sehr viele Möglichkeiten, die wir Ihnen hier nicht im Detail vorstellen können. Weitere Informationen finden Sie zum Beispiel im Buch von Hauser bei den Literaturempfehlungen.

2.2 XML-Dateien mit dem Editor erstellen

Da wir in unserem ersten Beispiel zunächst nur lesend auf eine XML-Datei zugreifen werden, erstellen Sie jetzt bitte eine Testdatei. Das können Sie direkt in Visual Studio erledigen.

Legen Sie dazu bitte eine neue Windows Forms-Anwendung an. Öffnen Sie dann das Menü **Projekt** und klicken Sie auf **Neues Element hinzufügen ...**. Wählen Sie im Fenster **Neues Element hinzufügen** den Eintrag **XML-Datei** unten in der mittleren Liste.

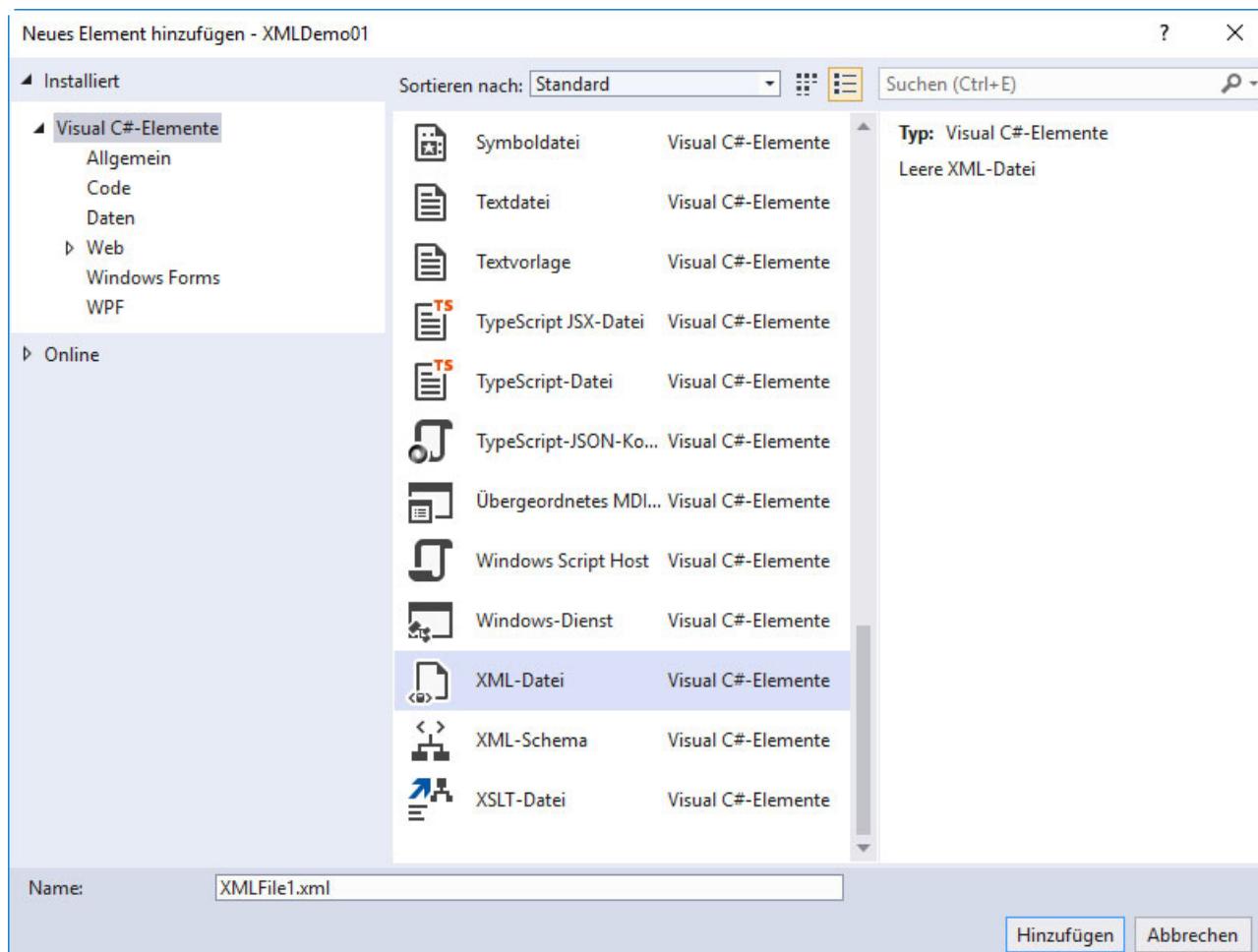


Abb. 2.5: Das Anlegen einer XML-Datei mit Visual Studio

Geben Sie anschließend einen Namen für die Datei in das Feld **Name:** ein, und klicken Sie auf **Hinzufügen**. Wir verwenden in unserem Beispiel den Namen **test.xml**.

In der Datei, die Visual Studio anlegt, ist die XML-Deklaration bereits vorhanden. Ergänzen Sie bitte noch die folgenden Zeilen und speichern Sie die Datei dann in einem Ordner \test. Benutzen Sie dazu die Funktion **test.xml speichern unter ...** im Menü Datei.

Hinweis:

Sie können auch einen beliebigen anderen Ordner benutzen. Wichtig ist vor allem, dass Sie die Datei ohne Schwierigkeiten wiederfinden.

```
<eintraege>
  <eintrag>Ich bin der erste Eintrag</eintrag>
  <eintrag>Ich bin der zweite Eintrag</eintrag>
  <eintrag>Ich bin der dritte Eintrag</eintrag>
  <eintrag>Ich bin der vierte Eintrag</eintrag>
</eintraege>
```

Hinweis:

Der Editor von Visual Studio ergänzt automatisch End-Tags für ein geöffnetes Tag und überprüft auch, ob geöffnete Tags wieder geschlossen werden. Wenn das nicht der Fall ist, wird das geöffnete Tag mit einer roten Wellenlinie markiert und es erscheint eine Fehlermeldung.

Sie können auch die Datei **test.xml** verwenden. Sie finden diese Datei bei den Beispielen für den Lehrgang.

Sie können eine XML-Datei in Visual Studio auch ohne ein Projekt anlegen. Benutzen Sie dazu die Funktion **Datei/Neu/Datei ...**

Tipp:

Da es sich bei XML-Dateien um reinen Text handelt, können Sie auch einen beliebigen anderen Editor zum Anlegen der Datei verwenden – zum Beispiel den Standard-Editor von Windows. Sie müssen lediglich darauf achten, dass der Editor in der Lage ist, Dateien im UTF-8-Format – einer Variante des Unicode-Zeichensatzes – zu speichern. Beim Editor von Windows ändern Sie dazu beim Speichern den Eintrag im Feld **Codierung** entsprechend.

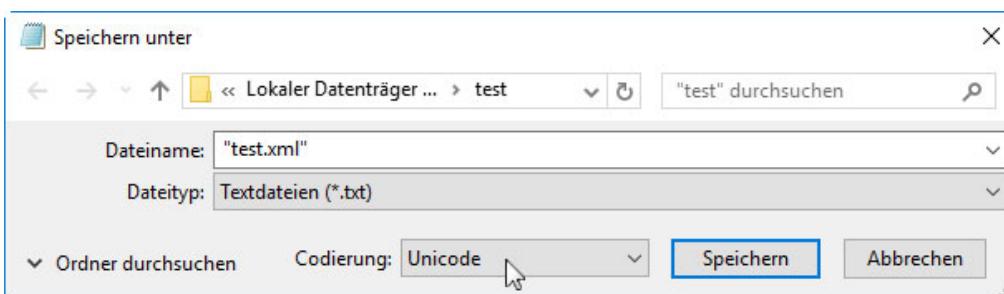


Abb. 2.6: Das Speichern im Format UTF-8 mit dem Windows Editor

Denken Sie daran, den Dateinamen gegebenenfalls in Anführungszeichen zu setzen. Andernfalls ergänzt der Editor von Windows unter Umständen beim Speichern automatisch die Erweiterung **.txt**.

Ob Ihnen beim Eingeben einer XML-Datei ein Fehler unterlaufen ist, können Sie übrigens sehr einfach überprüfen, indem Sie die Datei im Browser anzeigen lassen. Die Anzeige sollte dann so aussehen:



Abb. 2.7: Die Datei **test.xml** im Browser Internet Explorer

Kommen wir nun nach der ganzen Theorie zum praktischen Einsatz. Wir werden einige kleine Programme erstellen, die Daten aus einer XML-Datei lesen und auch Daten in eine XML-Datei schreiben.

2.3 Lesen aus einer XML-Datei

Beginnen wir mit einem ganz einfachen Beispiel. Wir wollen die Daten aus der Datei **test.xml** einlesen und in einem Listenfeld anzeigen. Dazu verwenden wir die Klasse `XmlReader` aus dem Namensraum `System.Xml`.

Legen Sie bitte ein Formular mit einem Listenfeld und einer Schaltfläche an. Beim Anklicken der Schaltfläche lassen Sie dann die Anweisungen aus dem folgenden Code ausführen. Was sich hinter diesen Anweisungen verbirgt, erklären wir Ihnen im Anschluss.

```
//eine neue Instanz für XmlReader über Create() erzeugen
XmlReader xmlLesen = XmlReader.Create("c:\\test\\test.xml");
//solange Daten gelesen werden können
while (xmlLesen.Read())
    //den Wert in das Listenfeld schreiben
    listBox1.Items.Add(xmlLesen.Value);
//die Datei wieder schließen
xmlLesen.Close();
```

Code 2.1: Das erste Lesen aus einer XML-Datei

Mit der Anweisung

```
XmlReader xmlLesen = XmlReader.Create("c:\\test\\test.xml");
```

erzeugen wir über die Methode `XmlReader.Create()` eine neue Instanz der Klasse `XmlReader`. Als Argument übergeben wir dabei den Namen der Datei, die geöffnet werden soll. In unserem Beispiel verwenden wir die Datei **test.xml** im Ordner **c:\test**. Falls sich Ihre Testdatei in einem anderen Ordner befindet, müssen Sie den Pfad entsprechend anpassen.

Zur Erinnerung:

Denken Sie unbedingt an die doppelten Trennzeichen \\ in der Pfadangabe beziehungsweise an das Zeichen @ vor der Zeichenkette. Andernfalls wird die Datei nicht gefunden und das Programm löst nach dem Anklicken der Schaltfläche eine Ausnahme aus.



In der Schleife

```
while (xmlLesen.Read())
    listBox1.Items.Add(xmlLesen.Value);
```

lesen wir dann über die Methode `Read()` den nächsten Knoten aus der Datei ein und schreiben den Wert in das Listenfeld. Den Wert erhalten wir dabei über die Eigenschaft `Value`.

Nach dem Einlesen eines Knotens über `Read()` wird automatisch der nächste Knoten in der Datei angesteuert. Da die Methode `Read()` so lange den Wert `true` liefert, bis keine weiteren Knoten mehr vorhanden sind, wird durch die `while`-Schleife der gesamte Inhalt der Datei eingelesen.



Ergänzen Sie jetzt bitte noch die Anweisung

```
using System.Xml;
```

für den Namensraum `System.Xml` unterhalb der anderen Anweisungen für das Ver einbaren der Namensräume. Speichern Sie dann alle Änderungen und testen Sie das Programm.

Das Ergebnis könnte ungefähr so aussehen:

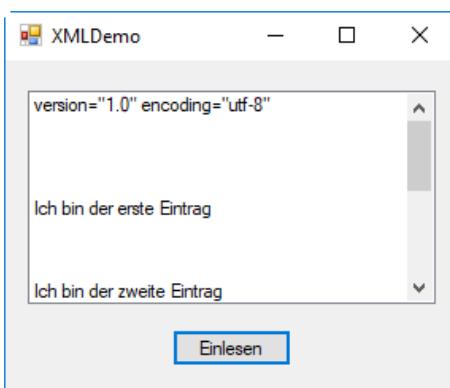


Abb. 2.8: Das erste Lesen aus der XML-Datei

Besonders brauchbar ist die Anzeige allerdings noch nicht. Zum einen wird auch die XML-Deklaration mitgelesen, zum anderen befindet sich eine ganze Reihe Leerzeilen in der Liste, die wir so nicht eingegeben haben. Das liegt daran, dass zum Beispiel auch Zeilenumbrüche in der Datei als Knoten betrachtet und mitverarbeitet werden.

Sie können aber über die Eigenschaft `NodeType`⁹ der Klasse `XmlReader` gezielt überprüfen, um welchen Knotentyp es sich handelt, beziehungsweise gezielt auf den Textinhalt eines Knotens zugreifen. Der Wert `XmlNodeType.Text` steht zum Beispiel für den Inhalt eines Knotens, der Wert `XmlNodeType.XmlDeclaration` für die XML-Deklaration und der Wert `XmlNodeType.Element` für ein Tag.

Um nun nur die reinen Textinhalte der Knoten anzuzeigen, müssen Sie nichts weiter machen, als mit einer `if`-Abfrage zu überprüfen, ob es sich beim aktuellen Knoten um den Typ `XmlNodeType.Text` handelt. Der erweiterte Code würde dann so aussehen:

```
//eine neue Instanz für XMLReader über Create() erzeugen
XmlReader xmlLesen = XmlReader.Create("c:\\test\\test.xml");
//solange Daten gelesen werden können
while (xmlLesen.Read())
    //den Wert in das Listenfeld schreiben, wenn es
    //sich um Text handelt
    if (xmlLesen.NodeType == XmlNodeType.Text)
        listBox1.Items.Add(xmlLesen.Value);
//die Datei wieder schließen
xmlLesen.Close();
```

Code 2.2: Das Lesen der Textinhalte der Knoten

Übernehmen Sie die Erweiterung und testen Sie das Programm noch einmal. Nun werden nur noch die eigentlichen Daten angezeigt.

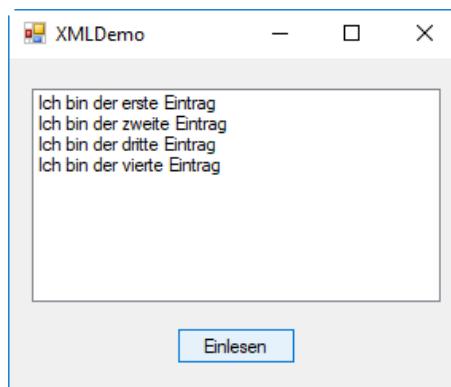


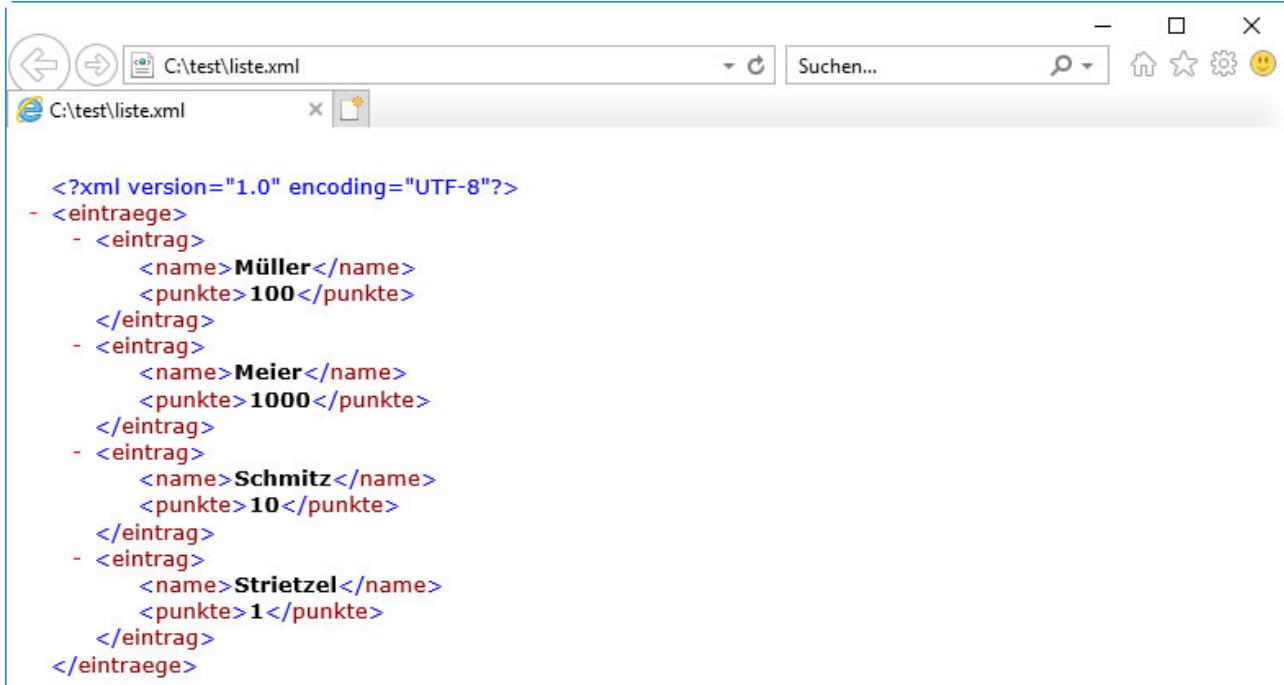
Abb. 2.9: Die Anzeige der Daten aus der XML-Datei

Über die Eigenschaft `Name` der Klasse `XmlReader` können Sie auch den Namen eines Knotens auswerten – also das Tag des Knotens. Das ist vor allem dann interessant, wenn Sie nur bestimmte Teile der Daten auswerten möchten.

Schauen wir uns auch dazu ein Beispiel an: In einer XML-Datei werden Namen und Punkte auf derselben Ebene gespeichert. Die Namen werden dabei durch das Tag `name` gekennzeichnet und die Punkte durch das Tag `punkte`.

9. `NodeType` bedeutet übersetzt so viel wie „Knotentyp“.

Die vollständige XML-Datei sieht im Browser so aus:



```

<?xml version="1.0" encoding="UTF-8"?>
- <eintraege>
  - <eintrag>
    <name>Müller</name>
    <punkte>100</punkte>
  </eintrag>
  - <eintrag>
    <name>Meier</name>
    <punkte>1000</punkte>
  </eintrag>
  - <eintrag>
    <name>Schmitz</name>
    <punkte>10</punkte>
  </eintrag>
  - <eintrag>
    <name>Strietzel</name>
    <punkte>1</punkte>
  </eintrag>
</eintraege>

```

Abb. 2.10: Eine Punktliste als XML-Datei

Im Code 2.3 werden über die Schleife sämtliche Daten aus der Datei gelesen. Die `if`-Abfrage überprüft dabei, ob es sich um ein Tag handelt und ob dieses Tag den Namen `name` hat. Wenn das der Fall ist, wird über die Methode `ReadElementString()`¹⁰ der Textinhalt des aktuellen Knotens eingelesen und in das Listenfeld geschrieben.

```

while (xmlLesen.Read())
    //den Wert in das Listenfeld schreiben, wenn es
    //sich um ein Tag mit dem Namen name handelt
    //bitte in einer Zeile eingeben
    if ((xmlLesen.NodeType == XmlNodeType.Element) &&
        (xmlLesen.Name == "name"))
        listBox1.Items.Add(xmlLesen.ReadElementString());

```

Code 2.3: Gezielter Zugriff auf einen Knoten

Etwas kompakter wird der gezielte Zugriff auf einen bestimmten Knoten, wenn Sie die Methode `ReadToFollowing()`¹¹ der Klasse `XmlReader` benutzen. Diese Methode sucht automatisch den nächsten passenden Knoten mit dem angegebenen Namen. Wenn kein passender Knoten gefunden wird, liefert die Methode `false` zurück.

10. `ReadElementString` lässt sich mit „Lese die Zeichenkette des Elements“ übersetzen.

11. `ReadToFollowing` bedeutet übersetzt so viel wie „Lese bis zum Nächsten“.

Die folgende Schleife erzielt also genau dieselbe Wirkung wie die Schleife im vorigen Code:

```
while (xmlLesen.ReadToFollowing("name"))
    listBox1.Items.Add(xmlLesen.ReadElementString());
```

Code 2.4: Kompakter Zugriff auf einen bestimmten Knoten über ReadToFollowing()

Über die Methode `ReadToNextSibling()`¹² können Sie auch einen Knoten ansteuern, der sich auf derselben Ebene wie der zuvor gelesene Knoten befindet – also quasi einen nebengeordneten Knoten. Als Argument übergeben Sie dabei wieder den Namen des Knotens.

Die folgende Konstruktion würde zum Beispiel zuerst den nächsten Knoten mit dem Namen **name** ansteuern und danach den Knoten **punkte** aus derselben Ebene der Struktur.

```
//den nächsten Knoten name ansteuern
while (xmlLesen.ReadToFollowing("name"))
{
    listBox1.Items.Add(xmlLesen.ReadElementString());
    //den nebengeordneten Knoten punkte ansteuern
    if (xmlLesen.ReadToNextSibling("punkte"))
        listBox1.Items.Add(xmlLesen.ReadElementString());
}
```

Code 2.5: Zugriff auf einen nebengeordneten Knoten über die Methode `ReadToNextSibling()`



Im praktischen Einsatz finden Sie die drei Varianten im Projekt **XmlDemo02**. Sie finden dieses Projekt im heftbezogenen Download-Bereich auf Ihrer Online-Lernplattform. Die Datei mit der Punktliste befindet sich bei den Beispielen für den Lehrgang.

So viel zum Lesen von Daten aus einer XML-Datei.

2.4 Schreiben in eine XML-Datei

Kommen wir nun zum Schreiben einer XML-Datei. Dazu verwenden Sie die Klasse `XmlWriter` aus dem Namensraum `System.Xml`. Diese Klasse verfügt über zahlreiche Methoden, mit denen Sie unter anderem die XML-Deklaration und auch Elemente recht komfortabel erstellen können.

Beginnen wir mit einem einfachen Beispiel. Aus einer Anwendung soll eine XML-Datei `probe.xml` mit einigen Einträgen erzeugt werden. Der entsprechende Quelltext sieht so aus:

```
//eine Instanz von XmlWriter erzeugen und die Datei anlegen
//bitte in einer Zeile eingeben
XmlWriter xmlSchreiben = XmlWriter.Create
("c:\\test\\probe.xml");
```

12. `ReadToNextSibling` bedeutet übersetzt „Lese bis zum nächsten Geschwister“. Mit Geschwister sind dabei Knoten auf derselben Ebene gemeint.

```
//die Deklaration schreiben  
xmlSchreiben.WriteStartDocument();  
//den Wurzelknoten erstellen  
xmlSchreiben.WriteStartElement("eintraege");  
//einen Eintrag erstellen  
xmlSchreiben.WriteStartElement("eintrag");  
//einen Namen mit Wert schreiben  
xmlSchreiben.WriteLineString("name", "Müller");  
//Punkte mit Wert schreiben  
xmlSchreiben.WriteLineString("punkte", "100");  
//den Eintrag abschließen  
xmlSchreiben.WriteEndElement();  
//und noch einen Eintrag erstellen  
xmlSchreiben.WriteStartElement("eintrag");  
//einen Namen mit Wert schreiben  
xmlSchreiben.WriteLineString("name", "Meier");  
//Punkte mit Wert schreiben  
xmlSchreiben.WriteLineString("punkte", "1000");  
//den Eintrag abschließen  
xmlSchreiben.WriteEndElement();  
//den Wurzelknoten abschließen  
xmlSchreiben.WriteEndElement();  
//das gesamte Dokument schließen  
xmlSchreiben.WriteEndDocument();  
//die Datei schließen  
//erst jetzt werden die Daten auch tatsächlich geschrieben  
xmlSchreiben.Close();
```

Code 2.6: Das Schreiben in eine XML-Datei

Zunächst einmal erzeugen wir mit der Anweisung

```
XmlWriter xmlSchreiben = XmlWriter.Create  
("c:\\test\\probe.xml");
```

über die Methode `XmlWriter.Create()` eine neue Instanz für die Klasse `XmlWriter`. Als Argument übergeben wir dabei den Namen der Datei. Falls diese Datei nicht vorhanden ist, wird sie automatisch erzeugt. Andernfalls wird sie überschrieben.

Mit der Anweisung

```
xmlSchreiben.WriteStartDocument();
```

erzeugen wir dann die XML-Deklaration. Als Version wird dabei automatisch 1.0 verwendet und für die Codierung UTF-8.

Anschließend erstellen wir mit den Anweisungen

```
xmlSchreiben.WriteStartElement("eintraege");  
xmlSchreiben.WriteStartElement("eintrag");
```

den Wurzelknoten mit dem Namen `eintraege` und einen untergeordneten Knoten `eintrag`.

Mit den beiden Anweisungen

```
xmlSchreiben.WriteElementString("name", "Müller");
xmlSchreiben.WriteElementString("punkte", "100");
```

schreiben wir dann die Elemente **name** und **punkte** in die Datei. Hinter dem Namen müssen Sie dabei jeweils den Wert als Zeichenkette angeben. Das End-Tag für ein Element wird über die Methode `WriteElementString()` automatisch erzeugt.

Die Anweisung

```
xmlSchreiben.WriteElementString("name", "Müller");
```

erzeugt also den folgenden Eintrag in der Datei:

```
<name>Müller</name>
```



Bitte beachten Sie:

Sie müssen den Wert des Elements bei der Methode `WriteElementString()` immer als Zeichenkette übergeben – auch dann, wenn es sich um eine Zahl handelt.

Anschließend schreiben wir das End-Tag für den Knoten **eintrag** über die Anweisung

```
xmlSchreiben.WriteEndElement();
```

Ein Argument müssen Sie an die Methode `WriteEndElement()` nicht übergeben. Sie wirkt immer auf den aktuellen noch nicht geschlossenen Knoten.

Mit den folgenden Anweisungen schreiben wir dann mit derselben Technik einen weiteren Eintrag mit dem Namen Meier und der Punktzahl 1 000 in die Datei. Dabei gibt es keine Besonderheiten.

Danach wird dann auch das End-Tag für den Wurzelknoten geschrieben und das Schreiben in das Dokument beendet. Das übernehmen die beiden Anweisungen

```
xmlSchreiben.WriteEndElement();
xmlSchreiben.WriteEndDocument();
```

Hinweis:

Sie können die letzten beiden Aufrufe von `WriteEndElement()` auch weglassen. Denn beim Aufruf der Methode `WriteEndDocument()` werden alle noch geöffneten Tags automatisch geschlossen.

Mit der Anweisung

```
xmlSchreiben.Close();
```

schließen wir die Datei wieder. Erst jetzt werden auch tatsächlich alle Daten vollständig geschrieben.



Bitte beachten Sie:

Achten Sie beim Schreiben sehr sorgfältig darauf, dass Sie die Datei wieder schließen. Andernfalls erzeugen Sie unter Umständen lediglich eine leere Datei.

Die XML-Datei, die wir erzeugt haben, sieht bei der Anzeige im Browser so aus:



```
<?xml version="1.0" encoding="UTF-8"?>
- <eintraege>
  - <eintrag>
    <name>Müller</name>
    <punkte>100</punkte>
  </eintrag>
  - <eintrag>
    <name>Meier</name>
    <punkte>1000</punkte>
  </eintrag>
</eintraege>
```

Abb. 2.11: Die Datei **probe.xml** im Browser

Bei der Anzeige in einem Editor werden Sie allerdings eine Überraschung erleben. Denn hier steht alles in einer einzigen Zeile.



```
<?xml version="1.0" encoding="utf-8"?><eintraege><eintrag><name>Müller</name><punkte>100</p
```

Abb. 2.12: Die Datei **probe.xml** in einem Editor

Dadurch werden die Daten nicht nur für einen Menschen sehr schwierig zu lesen, sondern unter Umständen auch für ein Programm. Denn bei der Darstellung in einer Zeile kann die Methode `ReadToNextSibling()` einen nebengeordneten Knoten nicht immer korrekt erkennen und auch Methoden wie `ReadToFollowing()` arbeiten nicht immer richtig.

Sie können aber über die Klasse `XmlWriterSettings` gezielt die Einstellungen für das Erzeugen der Datei verändern und so zum Beispiel dafür sorgen, dass Einrückungen verwendet werden.

Dazu erzeugen Sie im ersten Schritt eine neue Instanz der Klasse `XmlWriterSettings` und setzen die gewünschten Einstellungen. Danach geben Sie die Instanz der Klasse `XmlWriterSettings` beim Erzeugen der `XmlWriter`-Instanz über die Methode `Create()` als zweites Argument an.

Für unser Beispiel von oben könnte das so aussehen:

```
//die Einstellungen für XmlWriter setzen
XmlWriterSettings einstellungen = new XmlWriterSettings();
//Eintrückungen aktivieren
einstellungen.Indent = true;
//eine Instanz von XmlWriter erzeugen und die Datei anlegen
//bitte in einer Zeile eingeben
XmlWriter xmlSchreiben = XmlWriter.Create
("c:\\test\\probe.xml", einstellungen);
//die Deklaration schreiben
xmlSchreiben.WriteStartDocument();
...
```

Code 2.7: Das Setzen der Einstellungen

Danach werden auch bei der Anzeige im Editor die Zeilen ordentlich getrennt und eingekürtzt.

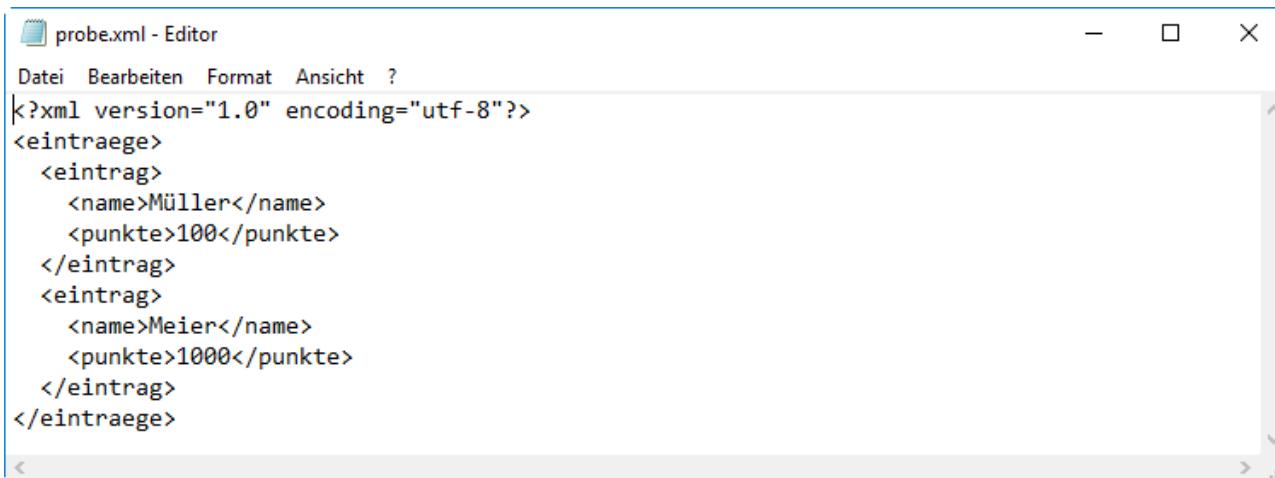


Abb. 2.13: Die geänderte Darstellung im Editor

Hinweis:

Denken Sie beim Test des Codes bitte an das Bekanntmachen des Namensraums `System.Xml`.

Die Klasse `XmlWriter` kennt noch zahlreiche weitere Methoden zum Schreiben von Daten, die wir Ihnen hier nicht einzeln vorstellen wollen. Auch bei den Einstellungen können Sie noch weitere Vorgaben setzen. Sehen Sie bei Interesse bitte in der Hilfe nach.

Beim Ändern der Voreinstellungen sollten Sie allerdings mit etwas Vorsicht ans Werk gehen. Denn unter Umständen wird eine XML-Datei mit falschen Voreinstellungen unleserlich beziehungsweise kann nicht mehr korrekt verarbeitet werden.



Das komplette Beispiel für den vorigen Code finden Sie im Projekt **XmlDemo03** im heftbezogenen Download-Bereich auf Ihrer Online-Lernplattform.

2.5 Ein etwas anspruchsvolleres Beispiel

Schauen wir uns nun ein etwas anspruchsvolleres Beispiel für die Arbeit mit XML-Dateien an. Das folgende Programm soll mehrere Werte in unterschiedlichen Formaten über eine XML-Datei verwalten und zusätzlich auch die Positionsdaten des Formulars in der XML-Datei speichern.

Legen Sie bitte eine neue Windows Forms-Anwendung an. Setzen Sie dann in das Formular drei Eingabefelder mit Beschriftung, eine GroupBox mit zwei Optionsfeldern, vier Labels und insgesamt drei Schaltflächen. Das Formular könnte zum Beispiel so aussehen:

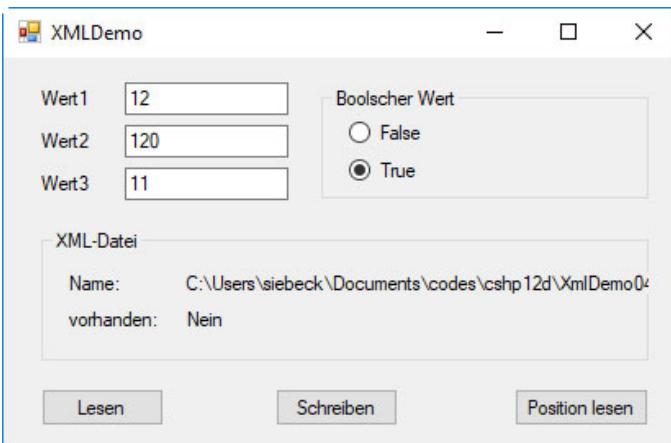


Abb. 2.14: Das Formular für die Anwendung

Hinweis:

Bei sehr langen Pfadnamen wird die Anzeige in der Anwendung abgeschnitten.
Wenn Sie das stört, können Sie das Label ja entsprechend verbreitern.

Sie finden das Projekt im heftbezogenen Download-Bereich unter dem Namen **XmlDemo04**.



In die drei Eingabefelder sollen beliebige Daten eingegeben werden können, die in der XML-Datei gespeichert werden. Über die Optionsfelder kann der Anwender einen booleschen Wert einstellen. Dieser Wert wird ebenfalls in der Datei gespeichert.

Im mittleren Bereich des Formulars sollen der vollständige Pfad und der Name der XML-Datei angezeigt werden. Außerdem soll hier ausgegeben werden, ob die Datei bereits angelegt ist oder nicht.

Den Pfad und den Namen der XML-Datei machen wir dabei vom Pfad und Namen der Anwendung abhängig. Dazu fragen wir im Konstruktor des Formulars über die Eigenschaft `Application.ExecutablePath`¹³ den Pfad und Namen der Anwendung ab und ändern dann über die Methode `System.IO.Path.ChangeExtension()`¹⁴ die Dateierweiterung in `.xml`. Das Ergebnis dieser Operation weisen wir anschließend einem Feld `xmlName` zu.

13. `ExecutablePath` steht für „Pfad der ausführbaren Datei“.

Die vollständige Anweisung sieht so aus:

```
xmlName = System.IO.Path.ChangeExtension
(Application.ExecutablePath, ".xml");
```

Hinweis:

Die Methode `System.IO.Path.ChangeExtension()` nimmt keine Änderungen an einer vorhandenen Datei vor und erzeugt auch keine neue Datei. Sie ändert lediglich in einer Zeichenkette vom Typ `string` eine eventuell vorhandene Dateierweiterung und liefert die geänderte Zeichenkette als Ergebnis zurück.

Danach überprüfen wir im Konstruktor, ob es die Datei bereits gibt. Dazu verwenden wir die Methode `System.IO.File.Exists()`. Diese Methode liefert `true`, wenn die Datei mit dem als Argument übergebenen Namen vorhanden ist. Andernfalls wird `false` geliefert. Das Ergebnis der Abfrage weisen wir einem Feld `xmlVorhanden` zu.

In unserem Beispiel sieht der komplette Konstruktor so aus:

```
public Form1()
{
    InitializeComponent();
    //den Namen aus der ausführbaren Datei ableiten
    //bitte in einer Zeile eingeben
    xmlName = System.IO.Path.ChangeExtension
    (Application.ExecutablePath, ".xml");
    //prüfen, ob es die Datei bereits gibt
    xmlVorhanden = System.IO.File.Exists(xmlName);
}
```

Code 2.8: Der Konstruktor

Im Ereignis `Shown` für das Formular setzen wir dann die Eigenschaft `Text` der beiden Labels für die Anzeige. Das ist nicht weiter schwierig.

```
private void Form1_Shown(object sender, EventArgs e)
{
    //die Anzeige in den Labels setzen
    labelName.Text = xmlName;
    if (xmlVorhanden == true)
        labelVorhanden.Text = "Ja";
    else
        labelVorhanden.Text = "Nein";
}
```

Code 2.9: Die Methode `Form1_Shown()`

14. *Change Extension* bedeutet übersetzt „Ändere Erweiterung“.

Die Methoden zum Schreiben und Lesen der Daten können dann so aussehen:

```
private void ButtonSchreiben_Click(object sender, EventArgs e)
{
    //die Einstellungen setzen
    XmlWriterSettings einstellungen = new XmlWriterSettings();
    einstellungen.Indent = true;
    //die Datei erzeugen
    //bitte in einer Zeile eingeben
    XmlWriter xmlSchreiben = XmlWriter.Create(xmlName,
    einstellungen);
    //die Deklaration schreiben
    xmlSchreiben.WriteStartDocument();
    //die Wurzel schreiben
    xmlSchreiben.WriteStartElement("daten");
    //das Unterelement werte schreiben
    xmlSchreiben.WriteStartElement("werte");
    //die Einträge schreiben
    xmlSchreiben.WriteLineString("wert1", textBoxWert1.Text);
    xmlSchreiben.WriteLineString("wert2", textBoxWert2.Text);
    xmlSchreiben.WriteLineString("wert3", textBoxWert3.Text);
    if (radioButtonTrue.Checked == true)
        xmlSchreiben.WriteLineString("bool", "true");
    else
        xmlSchreiben.WriteLineString("bool", "false");
    //das Element werte abschließen
    xmlSchreiben.WriteEndElement();
    //und die Positionsdaten
    xmlSchreiben.WriteStartElement("position");
    //bitte jeweils in einer Zeile eingeben
    xmlSchreiben.WriteLineString("top",
    Convert.ToString(this.Top));
    xmlSchreiben.WriteLineString("left",
    Convert.ToString(this.Left));
    xmlSchreiben.WriteLineString("height",
    Convert.ToString(this.Height));
    xmlSchreiben.WriteLineString("width",
    Convert.ToString(this.Width));
    //das Schreiben abschließen
    //offene Tags werden automatisch geschlossen
    xmlSchreiben.WriteEndDocument();
    //die Datei schließen
    xmlSchreiben.Close();
    //jetzt ist die Datei vorhanden
    if (xmlVorhanden == false)
    {
        xmlVorhanden = true;
        labelVorhanden.Text = "Ja";
    }
}

private void ButtonLesen_Click(object sender, EventArgs e)
{
    //für den booleschen Wert
    bool tempBool;
```

```

//wenn die Datei nicht da ist, verlassen wir die
//Methode direkt wieder
if (xmlVorhanden == false)
{
    MessageBox.Show("Die Datei ist nicht vorhanden");
    return;
}
//die Datei öffnen
XmlReader xmlLesen = XmlReader.Create(xmlName);
//die Werte für die drei Eingabefelder einlesen
xmlLesen.ReadToFollowing("wert1");
textBoxWert1.Text = xmlLesen.ReadElementString();
xmlLesen.ReadToFollowing("wert2");
textBoxWert2.Text = xmlLesen.ReadElementString();
xmlLesen.ReadToFollowing("wert3");
textBoxWert3.Text = xmlLesen.ReadElementString();
//den booleschen Wert lesen
xmlLesen.ReadToFollowing("bool");
tempBool = Convert.ToBoolean(xmlLesen.ReadElementString());
//die Datei schließen
xmlLesen.Close();
//den RadioButton setzen
if (tempBool == true)
    radioButtonTrue.Checked = true;
else
    radioButtonFalse.Checked = true;
}

private void ButtonPosLesen_Click(object sender, EventArgs e)
{
    //wenn die Datei nicht da ist, verlassen wir die
    //Methode direkt wieder
    if (xmlVorhanden == false)
    {
        MessageBox.Show("Die Datei ist nicht vorhanden");
        return;
    }
    //die Datei öffnen
    XmlReader xmlLesen = XmlReader.Create(xmlName);
    //die Werte einlesen und die Eigenschaften setzen
    xmlLesen.ReadToFollowing("top");
    this.Top = Convert.ToInt32(xmlLesen.ReadElementString());
    xmlLesen.ReadToFollowing("left");
    this.Left = Convert.ToInt32(xmlLesen.ReadElementString());
    xmlLesen.ReadToFollowing("height");
    this.Height = Convert.ToInt32(xmlLesen.ReadElementString());
    xmlLesen.ReadToFollowing("width");
    this.Width = Convert.ToInt32(xmlLesen.ReadElementString());
    //die Datei schließen
    xmlLesen.Close();
}

```

Code 2.10: Die Methoden zum Schreiben und Lesen

Beim Schreiben der Daten setzen wir zunächst die Einstellungen für die Klasse `XmlWriter` so, dass die Zeilen eingerückt werden. Danach schreiben wir dann der Reihe nach die verschiedenen Informationen in die Datei. Dabei gibt es keine Besonderheiten.

In der Methode `ButtonLesen_Click()` überprüfen wir zuerst über das Feld `xmlVorhanden`, ob die Datei angelegt ist. Falls nicht, wird eine Meldung ausgegeben und die Methode wieder verlassen.

Wenn die Datei vorhanden ist, lesen wir die Daten aus und weisen sie den entsprechenden Steuerelementen im Formular zu. Beim Lesen des booleschen Wertes konvertieren wir die Zeichenkette vorher noch in den passenden Typ.

Beim Lesen der Positionsdaten schließlich steuern wir gezielt die Knoten mit den Daten für die obere linke Ecke sowie die Höhe und Breite an. Damit die Daten korrekt zugewiesen werden können, wandeln wir sie noch in den Typ `Int32` um.

Übernehmen Sie jetzt die Anweisungen aus den verschiedenen Codes und testen Sie das Programm dann.

Hinweis:

Denken Sie bitte daran, dass Sie den Namensraum `System.Xml` bekannt machen müssen. Andernfalls erhalten Sie zahlreiche Fehlermeldungen.

Damit ist auch diese Anwendung fertig programmiert. Im nächsten Kapitel werden wir uns um das Speichern und Abrufen der Einstellungen für das Pong-Spiel aus dem letzten Studienheft kümmern.

Bitte beachten Sie:

Die Methoden der Klassen `XmlReader` und `XmlWriter` können Daten nur sequenziell – also der Reihe nach – verarbeiten. Sie können zum Beispiel nicht mit der Methode `ReadToFollowing()` Daten am Ende einer Datei einlesen und danach Daten zu Beginn der Datei. Sie müssen daher selbst sorgfältig darauf achten, dass Sie die Informationen exakt in derselben Reihenfolge einlesen, wie sie in die Datei geschrieben wurden. Andernfalls scheitert das Einlesen der Daten unter Umständen.

Das .NET Framework kennt aber auch eine Klasse `XmlDocument`, die sehr viel flexibler ist. So können Sie mit den Methoden dieser Klasse beliebig in einem XML-Dokument navigieren und auch Daten ändern beziehungsweise an einer beliebigen Stelle einfügen. Mehr zu dieser Klasse finden Sie in der Hilfe unter dem Eintrag `XmlDocument`.



Zusammenfassung

In XML-Dateien können Sie Daten in strukturierter Form als Text speichern. XML-Dateien haben einen festen Aufbau.

Der Zugriff auf XML-Dateien erfolgt über die Klassen `XmlWriter` und `XmlReader`.

Instanzen der Klassen `XmlWriter` und `XmlReader` erzeugen Sie über die Methode `Create()`. Dabei übergeben Sie den Namen der Datei als Argument.

Die beiden Klassen kennen verschiedene Methoden zum Lesen und Schreiben von Einträgen.

Beim Schreiben einer XML-Datei müssen Sie selbst dafür sorgen, dass zum Beispiel auch die XML-Deklaration erstellt wird.

Wenn Sie die Daten verarbeitet haben, müssen Sie die XML-Datei wieder schließen. Das ist besonders beim Speichern von Daten sehr wichtig.

Aufgaben zur Selbstüberprüfung

- 2.1 Beschreiben Sie, wie XML-Dateien grundsätzlich aufgebaut sind.

- 2.2 Sie finden in einer XML-Datei den Eintrag `<Name>`. Was bezeichnet dieser Eintrag?

- 2.3 In welchem Namensraum befinden sich die Klassen `XmlReader` und `XmlWriter`?

- 2.4 Über welche Klasse setzen Sie die Einstellungen für das Schreiben einer XML-Datei? Wie sorgen Sie dafür, dass die Einstellungen wirksam werden?

- 2.5 Mit welcher Methode schreiben Sie die XML-Deklaration in eine Datei?

- 2.6 Sie wollen einen festen Pfad für eine XML-Datei verwenden. Was müssen Sie dabei beachten?

3 Speichern der Einstellungen für das Pong-Spiel

In diesem Kapitel sorgen wir dafür, dass die Einstellungen aus dem Pong-Spiel des letzten Studienhefts in XML-Dateien gespeichert werden. Außerdem sollen auch die Einträge in der Bestenliste dauerhaft auf der Festplatte abgelegt werden.

Bevor wir uns an die praktische Umsetzung machen, zunächst wieder ein paar Vorüberlegungen.

3.1 Vorüberlegungen

Wir verwenden zwei getrennte Dateien: Eine Datei enthält die Größe des Spielfelds und den Schwierigkeitsgrad, die andere Datei die Bestenliste. Die Datei für die Größe und den Schwierigkeitsgrad verarbeiten wir direkt in der eigentlichen Anwendung – also in der Klasse `Form1`. Die Datei für die Bestenliste dagegen lassen wir durch die Klasse `Score` verwalten. Damit halten wir die Bestenliste sauber getrennt vom Rest des Spiels und können sie ohne Weiteres auch in anderen Anwendungen benutzen.

Hinweis:

Um die Datei für die Bestenliste kümmern wir uns gleich. Jetzt konzentrieren wir uns erst einmal auf die Datei mit den Einstellungen.

Die Größe des Spielfelds und den Schwierigkeitsgrad speichern wir in einer XML-Datei, die denselben Namen wie die ausführbare Datei unserer Anwendung haben soll. Lediglich die Dateierweiterung ändern wir in `.xml`.

Die Datei besteht aus zwei Knoten und drei Einträgen. Im Knoten `groesse` speichern wir die Breite und die Höhe des Formulars in den Einträgen `breite` und `hoehe`. Im Knoten `schwierigkeitsgrad` legen wir den aktuell eingestellten Schwierigkeitsgrad in numerischer Form ab. Für die Einstellung `Sehr einfach` verwenden wir den Wert 1, für die Einstellung `Einfach` den Wert 2 und so weiter. Die Einträge in der XML-Datei sollen später so aussehen:

```
<pong>
  <groesse>
    <breite>640</breite>
    <hoehe>480</hoehe>
  </groesse>
  <schwierigkeitsgrad>
    <wert>3</wert>
  </schwierigkeitsgrad>
</pong>
```

Für das Lesen und Schreiben der Daten erstellen wir zwei neue Methoden in der Klasse `Form1`. Das Lesen übernimmt die Methode `LeseEinstellungen()` und das Schreiben die Methode `SchreibeEinstellungen()`.

Die Methode `SchreibeEinstellungen()` lassen wir beim Beenden des Spiels ausführen und die Methode `LeseEinstellungen()` direkt nach dem Start des Spiels.

3.2 Spielfeldgröße und Schwierigkeitsgrad

Beginnen wir jetzt mit der Umsetzung der beiden Methoden. Legen Sie im ersten Schritt bitte die folgenden neuen Felder für die Klasse `Form1` des Pong-Spiels an:

- ein Feld `xmlDateiname` vom Typ `string` für den Namen der XML-Datei,
- ein Feld `xmlBreite` vom Typ `int` für die Breite des Formulars,
- ein Feld `xmlHoehe` vom Typ `int` für die Höhe des Formulars und
- ein Feld `xmlSchwierigkeit` vom Typ `int` für den Schwierigkeitsgrad.

Ergänzen Sie dann noch den Namensraum `System.Xml` bei den anderen Namensräumen und sichern Sie anschließend die Änderungen.

Das Speichern der Einstellungen ist nicht weiter schwierig. Erweitern Sie bitte die Methoden beim Anklicken der Menüeinträge für den Schwierigkeitsgrad so, dass das Feld `xmlSchwierigkeit` passend gesetzt wird. Die Methode `SehrEinfachToolStripMenuItem_Click()` könnte dann zum Beispiel so aussehen:

```
private void SehrEinfachToolStripMenuItem_Click(object sender,
EventArgs e)
{
    //das Intervall für den Ball setzen
    timerBall.Interval = 200;
    //die Einstellungen setzen
    SetzeEinstellungen(100, 1, -20, 2);
    //und die Markierungen
    schwerToolStripMenuItem.Checked = false;
    sehrEinfachToolStripMenuItem.Checked = true;
    einfacheToolStripMenuItem.Checked = false;
    mittelToolStripMenuItem.Checked = false;
    sehrSchwerToolStripMenuItem.Checked = false;
    //für das Speichern des Schwierigkeitsgrades
    xmlSchwierigkeit = 1;
}
```

Code 3.1: Die erweiterte Methode `SehrEinfachToolStripMenuItem_Click()` (die neue Anweisung ist fett markiert)

Hinweis:

Denken Sie bitte daran, dass Sie das Feld `xmlSchwierigkeit` in insgesamt fünf Methoden setzen müssen – und zwar passend für den Menüeintrag.

Damit das Feld `xmlSchwierigkeit` auch dann einen definierten Wert hat, wenn der Anwender die Standardeinstellung nicht verändert, setzen Sie das Feld bitte noch im Konstruktor der Klasse `Form1` auf den Wert 2. Dieser Wert steht für die Einstellung **Einfach**.

Setzen Sie außerdem im Konstruktor von `Form1` das Feld `xmlDateiname` so, dass der Name der Anwendung und die Erweiterung `.xml` benutzt werden. Dazu können Sie die folgende Anweisung verwenden:

```
xmlDateiname = System.IO.Path.ChangeExtension
(Application.ExecutablePath, ".xml");
```

Hinweis:

Damit die Reihenfolge beim Setzen der Standardeinstellungen später stimmt, nehmen Sie die Erweiterungen im Konstruktor bitte nach dem Setzen der Einstellungen für die Ballrichtung und den Ballwinkel vor.

In der Methode `SchreibeEinstellungen()` setzen wir zunächst die Einstellungen für die Datei, erzeugen wir dann eine neue Instanz der Klasse `XmlWriter` und schreiben anschließend die Daten in die Datei. Dabei gibt es keine Besonderheiten.

Die vollständige Methode `SchreibeEinstellungen()` sieht so aus:

```
//speichert die Einstellungen
void SchreibeEinstellungen()
{
    //die Einstellungen setzen
    XmlWriterSettings einstellungen = new XmlWriterSettings();
    einstellungen.Indent = true;
    //eine Instanz für XmlWriter erzeugen
    //bitte in einer Zeile eingeben
    XmlWriter xmlSchreiben =
        XmlWriter.Create(xmlDateiname, einstellungen);
    //die Deklaration schreiben
    xmlSchreiben.WriteStartDocument();
    //den Wurzelknoten pong erzeugen
    xmlSchreiben.WriteStartElement("pong");
    //den Knoten groesse erzeugen
    xmlSchreiben.WriteStartElement("groesse");
    //die Einträge schreiben
    //bitte jeweils in einer Zeile eingeben
    xmlSchreiben.WriteString("breite",
        Convert.ToString(this.Width));
    xmlSchreiben.WriteString("hoehe",
        Convert.ToString(this.Height));
    //den Knoten abschließen
    xmlSchreiben.WriteEndElement();
    //den Knoten schwierigkeitsgrad erzeugen
    xmlSchreiben.WriteStartElement("schwierigkeitsgrad");
    //den Eintrag schreiben
    //bitte in einer Zeile eingeben
    xmlSchreiben.WriteString("wert",
        Convert.ToString(xmlSchwierigkeit));
    //alle abschließen
    xmlSchreiben.WriteEndDocument();
    //Datei schließen
    xmlSchreiben.Close();
}
```

Code 3.2: Die Methode `SchreibeEinstellungen()`

Die Größe des Spielfelds ermitteln wir über die aktuelle Größe des Formulars. Das ist in unserem Beispiel ohne Probleme möglich, da der Anwender das Formular ja nicht beliebig verändern kann. Änderungen sind nur über die Spieldelinstellungen möglich.

Der Aufruf der Methode `SchreibeEinstellungen()` erfolgt dann im Ereignis **Form-Closed** des Formulars. Dabei gibt es ebenfalls keine Besonderheiten.

Speichern Sie jetzt die Änderungen und führen Sie einen ersten Test durch. Nach dem Beenden des Spiels sollte die XML-Datei mit den Daten für das Spielfeld und den Schwierigkeitsgrad in dem Ordner vorhanden sein, in dem sich auch die ausführbare Datei für das Spiel befindet.

Kommen wir nun zum Lesen der Einstellungen. Dabei lesen wir zunächst die Einstellungen aus der XML-Datei, setzen dann die Größe des Formulars und stellen abschließend den Schwierigkeitsgrad über den Aufruf der entsprechenden Methode für das Anklicken des jeweiligen Menüeintrags ein.

Die Methode `LeseEinstellungen()` sieht dann so aus:

```
//liest die Einstellungen
void LeseEinstellungen()
{
    //gibt es die Datei?
    if (System.IO.File.Exists(xmlDateiname) == false)
        return;
    //eine Instanz von XmlReader erzeugen
    XmlReader xmlLesen = XmlReader.Create(xmlDateiname);
    //die Daten lesen und zuweisen
    xmlLesen.ReadToFollowing("breite");
    xmlBreite = Convert.ToInt32(xmlLesen.ReadElementString());
    xmlLesen.ReadToFollowing("hoehe");
    xmlHoehe = Convert.ToInt32(xmlLesen.ReadElementString());
    xmlLesen.ReadToFollowing("wert");
    //bitte in einer Zeile eingeben
    xmlSchwierigkeit = Convert.ToInt32
    (xmlLesen.ReadElementString());
    //die Datei wieder schließen
    xmlLesen.Close();
    //nach Schwierigkeitsgrad die Einstellungen setzen
    //als Sender wird das Formular übergeben, das
    //zweite Argument ist EventArgs.Empty
    switch (xmlSchwierigkeit)
    {
        //bitte jeweils in einer Zeile eingeben
        case 1:
            SehrEinfachToolStripMenuItem_Click(this,
                EventArgs.Empty);
            break;
        case 2:
            EinfachToolStripMenuItem_Click(this, EventArgs.Empty);
            break;
        case 3:
            MittelToolStripMenuItem_Click(this, EventArgs.Empty);
            break;
        case 4:
            SchwerToolStripMenuItem_Click(this, EventArgs.Empty);
            break;
    }
}
```

```

        case 5:
            SehrSchwerToolStripMenuItem_Click(this,
                EventArgs.Empty);
            break;
    }
}

```

Code 3.3: Die Methode LeseEinstellungen()

Die einzige Besonderheit liegt im Aufruf der Methoden für das Anklicken der Menüeinträge. Diese Methoden erwarten als Argumente einen Sender vom Typ `object` und zusätzliche Informationen vom Typ `EventArgs`. In unserem Fall benutzen wir hier als Sender das Formular – also `this` – und `EventArgs.Empty` für ein Ereignis ohne Daten.

Nun müssen wir noch dafür sorgen, dass beim Spielstart die Einstellungen gelesen und über die entsprechenden Methoden auch gesetzt werden. Dazu sind einige Änderungen am Konstruktor der Klasse `Form1` erforderlich.

Ergänzen Sie bitte nach den Anweisungen zum Setzen des Dateinamens für die XML-Datei die Anweisungen aus dem folgenden Code. Sie setzen einige zusätzliche Werte, lesen die Daten aus der XML-Datei ein und setzen dann die Größe des Formulars auf die gespeicherten Einstellungen.

```

...
//Standardwert für die Größe
xmlBreite = 640;
xmlHoehe = 480;
//die Daten lesen
LeseEinstellungen();
//die Größe des Formulars setzen
this.Height = xmlHoehe;
this.Width = xmlBreite;
...

```

Code 3.4: Das Lesen der Einstellungen und das Setzen des Spielfelds

Hinweis:

Die ersten beiden Anweisungen sind erforderlich, da Sie nicht sicher sein können, dass das Einlesen gelingt. Für diesen Fall müssen Sie selbst dafür sorgen, dass die Höhe und die Breite einen definierten Wert haben.

Sie können das Setzen der Formulargröße auch in der Methode `LeseEinstellungen()` vornehmen.

Der vollständige Konstruktor sollte nun so aussehen:

```

public Form1()
{
    InitializeComponent();
    //die Breite der Linien
    spielfeldLinienbreite = 10;
    //die Größe des Schlägers
    schlaegerGroesse = 50;
}

```

```
//die Standardeinstellungen für den
//Schwierigkeitsgrad Einfach
punkteMehr = 1;
punkteWeniger = -5;
winkelZufall = 5;
//die Standardschwierigkeit
xmlSchwierigkeit = 2;
//den Dateinamen setzen
//bitte in einer Zeile eingeben
xmlDateiname = System.IO.Path.
ChangeExtension(Application.ExecutablePath, ".xml");
//Standardwert für die Größe
xmlBreite = 640;
xmlHoehe = 480;
//die Daten lesen
LeseEinstellungen();
//die Größe des Formulars setzen
this.Height = xmlHoehe;
this.Width = xmlBreite;
//erst einmal geht der Ball nach rechts und oben
//mit dem Winkel 0
ballPosition.richtungX = true;
ballPosition.richtungY = true;
ballPosition.winkel = 0;
//den Pinsel erzeugen
pinsel = new SolidBrush(Color.Black);
//die Zeichenfläche beschaffen
zeichenflaeche = spieldfeld.CreateGraphics();
//das Spielfeld bekommt einen schwarzen Hintergrund
spieldfeld.BackColor = Color.Black;
//die Grenzen für das Spielfeld setzen
setzeSpielfeld();
//einen neuen Ball erstellen
neuerBall();
//erst einmal ist das Spiel angehalten
spielPause = true;
//alle drei Timer sind zunächst angehalten
timerBall.Enabled = false;
timerSpiel.Enabled = false;
timerSekunde.Enabled = false;
//die Schriftauszeichnung setzen
schrift = new Font("Arial", 12, FontStyle.Bold);
//die Pausenfunktion ist erst einmal deaktiviert
pauseToolStripMenuItem.Enabled = false;
//eine neue Instanz der Klasse Score für die Punkte
spielpunkte = new Score();
}
```

Code 3.5: Der geänderte Konstruktor für die Klasse Form1
(die neuen Anweisungen sind fett markiert)



Bitte beachten Sie:

Sie dürfen die Anweisungen zum Setzen der Standardeinstellungen für die Punkte und den Winkel nicht am Ende des Konstruktors stehen lassen. Sonst überschreiben Sie die Daten, die Sie aus der Datei lesen, direkt wieder. Setzen Sie die Anweisungen am besten unmittelbar vor die Anweisungen zum Lesen der Daten.

Damit sind das Speichern und Zurücklesen der Einstellungen nahezu komplett. Sichern Sie die Änderungen und probieren Sie die Erweiterung des Pong-Spiels dann aus.

Im letzten Schritt können Sie noch dafür sorgen, dass im Formular für die Einstellungen der Spielfeldgröße auch das Optionsfeld mit der aktuellen Größe markiert ist. Diese Erweiterung haben Sie ja bereits bei den Einsendeaufgaben des letzten Studienhefts vorgenommen. Wir stellen sie Ihnen daher hier nicht weiter vor.

Jetzt können wir uns um das Speichern und Lesen der Bestenliste kümmern.

3.3 Die Bestenliste

Dazu verwenden wir eine eigene XML-Datei mit dem Namen `score.xml`. In dieser XML-Datei legen wir für jeden Eintrag in der Bestenliste einen eigenen Knoten mit dem Namen `eintrag` an.

Die Einträge in der Datei `score.xml` könnten zum Beispiel so aussehen:

```
<bestenliste>
  <eintrag>
    <name>Basti</name>
    <punkte>200</punkte>
  </eintrag>
  <eintrag>
    <name>Papa</name>
    <punkte>45</punkte>
  </eintrag>
  <eintrag>
    <name>Pilli</name>
    <punkte>29</punkte>
  </eintrag>
  <eintrag>
    <name>Papa mit wenig Punkten</name>
    <punkte>13</punkte>
  </eintrag>
  ...
</bestenliste>
```

Die Methode für das Lesen der Daten rufen wir im Konstruktor der Klasse `Score` auf, die Methode für das Speichern der Daten nach jedem neuen Eintrag in der Bestenliste.

Die beiden neuen Methoden der Klasse `Score` selbst könnten so aussehen:

```
//zum Lesen aus der Datei
void LesePunkte()
{
```

```
//zum Zwischenspeichern der gelesenen Daten
int tempPunkte;
string tempName;
//eine Instanz von XmlReader erzeugen
XmlReader xmlLesen = XmlReader.Create(xmlDateiname);
//die Daten in einer Schleife lesen und zuweisen
for (int i = 0; i < anzahl; i++) {
    xmlLesen.ReadToFollowing("name");
    tempName = xmlLesen.ReadElementString();
    xmlLesen.ReadToFollowing("punkte");
    //bitte in einer Zeile eingeben
    tempPunkte = Convert.ToInt32
        (xmlLesen.ReadElementString());
    bestenliste[i].SetzeEintrag(tempPunkte, tempName);
}
//die Datei wieder schließen
xmlLesen.Close();
}

//zum Schreiben in die Datei
void SchreibePunkte()
{
    //die Einstellungen setzen
    XmlWriterSettings einstellungen = new XmlWriterSettings();
    einstellungen.Indent = true;
    //eine Instanz für XmlWriter erzeugen
    //bitte in einer Zeile eingeben
    XmlWriter xmlSchreiben =
        XmlWriter.Create(xmlDateiname, einstellungen);
    //die Deklaration schreiben
    xmlSchreiben.WriteStartDocument();
    //den Wurzelknoten bestenliste erzeugen
    xmlSchreiben.WriteStartElement("bestenliste");
    //die Daten in einer Schleife wegschreiben
    for (int i = 0; i < anzahl; i++) {
        //den Knoten eintrag erzeugen
        xmlSchreiben.WriteStartElement("eintrag");
        //die Einträge schreiben
        //bitte jeweils in einer Zeile eingeben
        xmlSchreiben.WriteElementString("name",
            bestenliste[i].GetName());
        xmlSchreiben.WriteElementString("punkte",
            Convert.ToString(bestenliste[i].GetPunkte()));
        //den Knoten abschließen
        xmlSchreiben.WriteEndElement();
    }
    //alle abschließen
    xmlSchreiben.WriteEndDocument();
    //Datei schließen
    xmlSchreiben.Close();
}
```

Code 3.6: Die Methoden zum Lesen und Schreiben der Bestenliste

Hinweis:

Wir benötigen die beiden Methoden nur innerhalb der Klasse `Score`. Sie können daher auf die Sichtbarkeit `public` verzichten und die Standardsichtbarkeit verwenden.

Beim Lesen beschaffen wir uns in einer Schleife der Reihe nach die vorhandenen Einträge und weisen sie jeweils über die Methode `SetzeEintrag()` der Klasse `Liste` dem aktuellen Listeneintrag zu.

Beim Schreiben geht es genau in die andere Richtung. Hier verarbeiten wir in einer Schleife die aktuellen Einträge in der Liste und speichern sie in der Datei. Den Namen und die Punktzahl jedes Eintrags erhalten wir dabei über die Methoden `GetName()` beziehungsweise `GetPunkte()` der Klasse `Liste`.

Im Konstruktor weisen wir neben dem Aufruf der Methode `LesePunkte()` einem Feld `xmlDateiname` der Klasse `Score` einen Namen zu. Diesen Namen bauen wir aus dem Pfad der Anwendung und dem festen Text „score.xml“ zusammen. Den Pfad der Anwendung beschaffen wir uns dabei über die Eigenschaft `System.Windows.Forms.Application.StartupPath`¹⁵.

**Bitte beachten Sie:**

Die Eigenschaft `System.Windows.Forms.Application.StartupPath` liefert Ihnen den Pfad ohne ein abschließendes Zeichen \. Wenn Sie den Namen einer Datei aus dem Pfad der Anwendung erstellen wollen, müssen Sie daher selbst das Zeichen \ an den Pfad anhängen. Denken Sie dabei bitte wieder daran, das Zeichen doppelt anzugeben.

Der komplette Konstruktor unserer Klasse `Score` sieht dann so aus:

```
public Score()
{
    punkte = 0;
    //eine neue Instanz der Liste erstellen
    bestenliste = new Liste[anzahl];
    //die Elemente initialisieren
    for (int i = 0; i < anzahl; i++)
        bestenliste[i] = new Liste();
    //den Dateinamen aus dem Pfad zusammenbauen
    //bitte in einer Zeile eingeben
    xmlDateiname = System.Windows.Forms.
        Application.StartupPath + "\\score.xml";
    //wenn es die Datei schon gibt, die Daten lesen
    if (System.IO.File.Exists(xmlDateiname))
        LesePunkte();
}
```

Code 3.7: Der Konstruktor der Klasse `Score`

15. `StartupPath` lässt sich mit „Startpfad“ übersetzen.

Hinweis:

Das Vorbelegen der Liste über die Schleife sollten Sie nicht löschen. Die Daten werden zwar durch das Lesen der Bestenliste aus der Datei überschrieben – allerdings nur dann, wenn die Datei gelesen werden kann. Sie müssen daher sicherstellen, dass die Liste in jedem Fall definierte Werte enthält.

Das Speichern der Daten erfolgt in der Methode `NeuerEintrag()` der Klasse `Score`. Dazu ergänzen Sie nach dem Sortieren der Liste über die Anweisung

```
Array.Sort(bestenliste);  
die Anweisung  
SchreibePunkte();
```

Übernehmen Sie jetzt die beiden vorherigen Codes und die Anweisung zum Speichern der Liste in die Datei `Score.cs` des Pong-Spiels. Denken Sie dabei bitte auch daran, den Namensraum `System.Xml` bekannt zu machen und das Feld `xmlDateiname` für den Dateinamen in der Klasse `Score` festzulegen. Speichern Sie dann sämtliche Änderungen und testen Sie die Bestenliste.

Tipp:

Sie können die Einträge in der Bestenliste mit jedem Texteditor anpassen. Ändern Sie dazu einfach die Werte der einzelnen Knoten.

Das Pong-Spiel mit den Erweiterungen finden Sie im Projekt **Pong** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.



Zusammenfassung

Die Einstellungen und die Bestenliste für das Pong-Spiel haben wir auf zwei XML-Daten verteilt.

Um die Einträge in der Bestenliste unterscheiden zu können, haben wir für jeden Eintrag einen eigenen Knoten angelegt.

Das Lesen der Daten aus den Dateien erfolgt zu Beginn des Spiels, das Speichern beim Beenden des Spiels beziehungsweise beim Erstellen eines neuen Eintrags für die Bestenliste.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie wollen eine XML-Datei im selben Ordner wie die Anwendung speichern. Als Name soll dabei der Name der Anwendung mit der Erweiterung `.xml` benutzt werden. Mit welchem Ausdruck können Sie diese Vorgaben am einfachsten umsetzen?

- 3.2 Welche Eigenschaft liefert Ihnen den Pfad einer Anwendung? Worauf müssen Sie dabei besonders achten?

Schlussbetrachtung

Kompliment! Sie können jetzt XML-Dateien erstellen, lesen und auch Daten in XML-Dateien speichern. Außerdem haben Sie die Klassen `Registry` und `RegistryKey` des .NET Frameworks kennengelernt, die Ihnen den Zugriff auf die Registrierung von Windows ermöglichen.

Ob Sie Daten aus Ihren Anwendungen nun in XML-Dateien speichern oder in der Registrierung ablegen, ist in vielen Teilen auch Geschmackssache. XML-Dateien bieten den großen Vorteil, dass sie sehr gut strukturiert sind und sich so auch zur Ablage von Listen oder ähnlichen Konstruktionen eignen. Allerdings laufen Sie immer Gefahr, dass ein Anwender die Daten in der Datei verändert oder sogar aus Versehen die gesamte Datei löscht. Damit sind dann die gespeicherten Informationen verloren.

Die Registrierung von Windows bietet den großen Vorteil, dass alle Daten an einer zentralen Stelle liegen und vergleichsweise gut gegen Zugriffe durch den Anwender geschützt sind. Allerdings ist der Umgang mit der Registrierung nicht ohne Risiken. Sie müssen beim Programmieren sehr sorgfältig vorgehen, um nicht versehentlich Daten zu löschen oder zu verändern. Deshalb an dieser Stelle noch einmal der dringende Rat: Legen Sie vor Experimenten mit der Registrierung in jedem Fall Sicherungskopien an.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Daten für den aktuell angemeldeten Anwender werden im Hauptschlüssel **HKEY_CURRENT_USER** gespeichert.
- 1.2 Ein Eintrag in der Registrierung besteht aus dem Namen, dem Datentyp und dem eigentlichen Wert.
- 1.3 Dem Hauptschlüssel **HKEY_LOCAL_MACHINE** entspricht das Feld `LocalMachine` der Klasse `RegistryKey`.
- 1.4 Damit die Zeichen in einem Eingabefeld nicht im Klartext angezeigt werden, setzen Sie entweder die Eigenschaft **UseSystemPasswordChar** oder die Eigenschaft **PasswordChar**.
- 1.5 Die Methode heißt `Application.Exit()`.
- 1.6 Sie vereinbaren zunächst die Instanz `registrierung` der Klasse `RegistryKey`. Anschließend öffnen Sie über die Methode `OpenSubKey()` den gewünschten Schlüssel. Der Hauptschlüssel wird dabei über das Feld `CurrentUser` angegeben.

Die entsprechenden Anweisungen könnten so aussehen:

```
RegistryKey registrierung;
registrierung = Registry.CurrentUser.
OpenSubKey("Software");
```

- 1.7 Die Methode zum Schreiben von Werten heißt `SetValue()` und die Methode zum Lesen `GetValue()`. Die Methode `GetValue()` liefert den Typ `object` zurück. Sie müssen daher selbst dafür sorgen, dass die Daten in den passenden Typ umgewandelt werden.

Kapitel 2

- 2.1 Eine XML-Datei besteht aus einem Wurzelement und nahezu beliebigen Unterelementen. Für die Beschreibung und die Strukturierung werden Tags benutzt, die in der Regel aus einem Start-Tag und einem End-Tag bestehen.
Damit ein XML-Dokument korrekt erkannt wird, sollte es durch die XML-Deklaration eingeleitet werden.
- 2.2 Der Eintrag `<Name>` steht für ein Start-Tag.
- 2.3 Die Klassen `XmlReader` und `XmlWriter` befinden sich im Namensraum `System.Xml`.

- 2.4 Die Einstellungen für das Schreiben in eine XML-Datei setzen Sie über die Klasse `XmlWriterSettings`. Damit die Einstellungen wirksam werden, müssen Sie die Instanz der Klasse `XmlWriterSettings` als zweites Argument an die Methode `Create()` der Klasse `XmlWriter` übergeben.
- 2.5 Die XML-Deklaration schreiben Sie mit der Methode `WriteStartDocument()` der Klasse `XmlWriter`.
- 2.6 Sie müssen die Trennzeichen `\` doppelt im Pfad angeben – also zum Beispiel `C:\\temp\\datei.xml` – oder das Zeichen `@` vor die Zeichenkette mit dem Pfad setzen.

Kapitel 3

- 3.1 Der Ausdruck lautet:

```
System.IO.Path.ChangeExtension(Application.  
ExecutablePath, ".xml");
```

`Application.ExecutablePath` liefert dabei den Pfad und den Namen der Anwendung. Die Methode `System.IO.Path.ChangeExtension()` ändert die Erweiterung.

- 3.2 Den Pfad einer Anwendung erhalten Sie über die Eigenschaft `Application.StartupPath`. Es wird allerdings kein Trennzeichen `\` am Ende angehängt. Dieses Zeichen müssen Sie gegebenenfalls selbst hinzufügen.

B. Glossar

Extensible Markup Language	Siehe XML
HKEY_CLASSES_ROOT	HKEY_CLASSES_ROOT ist einer der Hauptschlüssel der Registrierung. Er enthält Daten über installierte Programme und Dateierweiterungen.
HKEY_CURRENT_CONFIG	HKEY_CURRENT_CONFIG ist einer der Hauptschlüssel der Registrierung. Er speichert die Hardwarekonfiguration des aktuell angemeldeten Anwenders.
HKEY_CURRENT_USER	HKEY_CURRENT_USER ist einer der Hauptschlüssel der Registrierung. In dem Schlüssel werden Einstellungen für den aktuell angemeldeten Anwender gespeichert.
HKEY_LOCAL_MACHINE	HKEY_LOCAL_MACHINE ist einer der Hauptschlüssel der Registrierung. Der Schlüssel speichert vor allem Daten zur verwendeten Hardware.
HKEY_USERS	HKEY_USERS ist einer der Hauptschlüssel der Registrierung. In ihm werden die Daten aller angelegten Benutzer gespeichert.
HTML	HTML steht für <i>Hypertext Markup Language</i> . HTML ist eine Auszeichnungssprache, die im Internet eingesetzt wird.
Hypertext Markup Language	Siehe HTML
Knoten	Als Knoten wird ein Unterelement in einer XML-Datei bezeichnet. Die Bezeichnung gilt aber auch generell für Elemente in einer Struktur.
Registrierung	Die Registrierung – auch <i>Registry</i> genannt – ist eine Art Datenbank, in der alle wichtigen Einstellungen von Windows zentral zusammenlaufen. Sie wurde mit Windows 95 eingeführt. Die Daten in der Registrierung werden im binären Format gespeichert und hierarchisch strukturiert.
Registrierungs-Editor	Der Registrierungs-Editor ist ein Programm zum Bearbeiten der Registrierung.
Registry	<i>Registry</i> ist die englische Bezeichnung für die Registrierung.

Tag (XML)

Ein XML-*Tag* dient zur Beschreibung und Strukturierung von Informationen in einer XML-Datei. Jedes Tag besteht in der Regel aus dem Start-Tag und dem End-Tag.

XML

XML (*eXtensible Markup Language*) ist ein textbasiertes, strukturiertes Format für die Beschreibung von Daten.

XML-Deklaration

Die XML-Deklaration leitet ein XML-Dokument ein. Sie beschreibt unter anderem die verwendete XML-Version und den eingesetzten Zeichensatz.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in Visual C# mit Visual Studio 2019*. Ideal für Programmieranfänger geeignet. 6. Aufl., Bonn: Rheinwerk.

Speziell mit dem Thema XML beschäftigt sich das folgende Buch:

Hauser, T. (2010). *XML Standards*. Schnell + kompakt. Frankfurt: entwickler.press.

D. Abbildungsverzeichnis

Abb. 1.1	Der Registrierungs-Editor	4
Abb. 1.2	Der Registrierungs-Editor in der Liste der Suchergebnisse	5
Abb. 1.3	Das Exportieren der Registrierungsdatei	6
Abb. 1.4	Die neuen Schlüssel in der Registrierung	12
Abb. 1.5	Das Formular zur Kennworteingabe (rechts) und das Formular zum Festlegen des Kennworts (links)	13
Abb. 1.6	Eine Listenanzeige für die Registrierung	19
Abb. 2.1	Eine typische Zeile in einer XML-Datei	24
Abb. 2.2	Die Deklaration eines XML-Dokuments	25
Abb. 2.3	Ein vollständiges XML-Dokument	25
Abb. 2.4	Die hierarchische Struktur der XML-Datei	25
Abb. 2.5	Das Anlegen einer XML-Datei mit Visual Studio	26
Abb. 2.6	Das Speichern im Format UTF-8 mit dem Windows Editor	27
Abb. 2.7	Die Datei test.xml im Browser Internet Explorer	28
Abb. 2.8	Das erste Lesen aus der XML-Datei	29
Abb. 2.9	Die Anzeige der Daten aus der XML-Datei	30
Abb. 2.10	Eine Punktliste als XML-Datei	31
Abb. 2.11	Die Datei probe.xml im Browser	35
Abb. 2.12	Die Datei probe.xml in einem Editor	35
Abb. 2.13	Die geänderte Darstellung im Editor	36
Abb. 2.14	Das Formular für die Anwendung	37
Abb. I.1	Das Formular der Anwendung	68

E. Tabellenverzeichnis

Tab. 1.1	Hauptschlüssel der Registrierung	3
Tab. 1.2	Felder der Klasse Registry	7

F. Codeverzeichnis

Code 1.1	Zugriff auf einen Schlüssel in der Registrierung	7
Code 1.2	Die Methoden ButtonLesen_Click() und ButtonSchreiben_Click()	10
Code 1.3	Das Speichern und Lesen der Positionsdaten für das Formular	11
Code 1.4	Die Methode Form1_Shown().....	14
Code 1.5	Die Anweisung für das Ereignis Click der Schaltfläche Übernehmen	15
Code 1.6	Die Methoden für das Formular zur Kennworteingabe	17
Code 1.7	Das Löschen des kompletten Zweigs	17
Code 1.8	Das Beschaffen der Einträge für die Listenanzeige	20
Code 2.1	Das erste Lesen aus einer XML-Datei.....	28
Code 2.2	Das Lesen der Textinhalte der Knoten	30
Code 2.3	Gezielter Zugriff auf einen Knoten	31
Code 2.4	Kompakter Zugriff auf einen bestimmten Knoten über ReadToFollowing()	32
Code 2.5	Zugriff auf einen nebengeordneten Knoten über die Methode ReadToNextSibling()	32
Code 2.6	Das Schreiben in eine XML-Datei	33
Code 2.7	Das Setzen der Einstellungen	36
Code 2.8	Der Konstruktor	38
Code 2.9	Die Methode Form1_Shown()	38
Code 2.10	Die Methoden zum Schreiben und Lesen	40
Code 3.1	Die erweiterte Methode SehrEinfachToolStripMenuItem_Click()	45
Code 3.2	Die Methode SchreibeEinstellungen()	46
Code 3.3	Die Methode LeseEinstellungen()	48
Code 3.4	Das Lesen der Einstellungen und das Setzen des Spielfelds	48
Code 3.5	Der geänderte Konstruktor für die Klasse Form1 (die neuen Anweisungen sind fett markiert)	49
Code 3.6	Die Methoden zum Lesen und Schreiben der Bestenliste	51
Code 3.7	Der Konstruktor der Klasse Score	52

G. Medienverzeichnis

Video 1.1 Sicherungskopie der Registrierung erstellen	6
--	---

H. Sachwortverzeichnis

A

Anweisung
Application.Exit() 16

E

Eigenschaft
Application.ExecutablePath 37
NodeType 30
PasswordChar 14
System.Windows.Forms.Application.
 StartupPath 52
UseSystemPasswordChar 14
End-Tag 24

H

HKEY_CLASSES_ROOT 3
HKEY_CURRENT_CONFIG 3
HKEY_CURRENT_USER 3
HKEY_LOCAL_MACHINE 3
HKEY_USERS 3

K

Kennwortabfrage 13
Klasse
 Registry 7
 RegistryKey 7
 XmlReader 28
 XmlWriter 32
 XmlWriterSettings 35
Knoten 24
Kommentar 25

L

Listenanzeige
 für die Registrierung 18

M

Methode
 CreateSubKey() 8
 DeleteSubKeyTree() 17
 GetSubKeyNames() 19
 GetValue() 7
 GetValueNames() 19

OpenSubKey() 7
Read() 29
ReadElementString() 31
ReadToFollowing() 31
ReadToNextSibling() 32
SetValue() 8
System.IO.Path.
 ChangeExtension() 37
WriteElementString() 34
WriteEndDocument() 34
WriteEndElement() 34
XmlReader.Create() 28
XmlWriter.Create() 33

P

Prolog 25

R

Registrierung 3
 Aufbau 3
 Hauptschlüssel 3
Registrierungs-Editor 3
Registry 3

S

Schlüssel
 Zugriff auf einen 7
Start-Tag 24

T

Tag 24

U

Unterelement 24

W

Wurzelement 24

X

XML 24

XML-Datei	24
Aufbau von	24
Lesen aus einer.....	28
mit dem Editor erstellen	26
Schreiben in eine	32
XML-Deklaration	25

I. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH12D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:
Datum:
Note:
Unterschrift Fernlehrer/in:

Schicken Sie für die Lösungen bitte jeweils das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

Beschreiben Sie bei den Aufgaben zusätzlich, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Steuerelemente Sie in dem Formular verwenden und welche Besonderheiten im Spiel berücksichtigt werden müssen.

- Ändern Sie die Listenanzeige für die Registrierung aus dem Studienheft (Abschnitt 1.5) so, dass der Anwender selbst auswählen kann, welche Schlüssel beziehungsweise Einträge im zweiten und dritten Listenfeld angezeigt werden.

Im ersten Listenfeld sollen immer fest die Einträge aus dem Schlüssel **HKEY_CURRENT_USER** angezeigt werden.

Achten Sie bitte darauf, dass die Einträge im zweiten beziehungsweise dritten Listenfeld nur dann angezeigt werden sollen, wenn im ersten beziehungsweise zweiten Listenfeld ein Eintrag markiert ist.

Damit die Lösung nicht zu aufwendig wird, sollen im zweiten Listenfeld immer nur die untergeordneten Schlüssel angezeigt werden. Im dritten Listenfeld dagegen sollen sowohl eventuell vorhandene Unterschlüssel als auch die Einträge angezeigt werden.

Bitte beachten Sie:

Für einige Unterschlüssel im Schlüssel **HKEY_CURRENT_USER** – wie zum Beispiel **Console** – gibt es nicht in jedem Fall weitere untergeordnete Schlüssel. Hier befinden sich auf der zweiten Ebene bereits Einträge. Es kann also durchaus sein, dass bei dem einen oder anderen Schlüssel das zweite Listenfeld leer bleibt.

40 Pkt.

2. Ändern Sie die Methoden zum Schreiben und Lesen der Spieleinstellungen für das Pong-Spiel so, dass die Daten über die Registrierung verarbeitet werden. Welche Schlüssel Sie für das Speichern verwenden, ist Ihnen freigestellt.

Stellen Sie bitte sicher, dass der Zugriff auf die Daten in der Registrierung nur dann erfolgt, wenn die Werte für das Pong-Spiel auch tatsächlich vorhanden sind.

Geben Sie für die Lösung dieser Aufgabe bitte die geänderten Quelltexte der beiden Methoden `SchreibeEinstellungen()` und `LeseEinstellungen()` der Klasse `Form1` sowie alle weiteren erforderlichen Änderungen an.

Hinweis:

Die Bestenliste soll nicht in der Registrierung gespeichert werden. Es sollen lediglich die Einstellungen zur Schwierigkeit sowie zur Größe des Spielfelds abgelegt werden.

30 Pkt.

3. Erstellen Sie eine Anwendung, die Einträge in einem Listenfeld in einer XML-Datei `liste.xml` speichert. Die Datei soll dabei im aktuellen Pfad der Anwendung gespeichert werden.

Die Einträge in der Liste sollen vom Anwender selbst über ein Eingabefeld erfasst werden können.

Wie Sie die Knoten und Einträge benennen, ist Ihnen selbst überlassen. Achten Sie aber bitte darauf, dass Sie die Vorgaben für die Struktur einer XML-Datei einhalten.

Das Formular der Anwendung könnte zum Beispiel so aussehen:

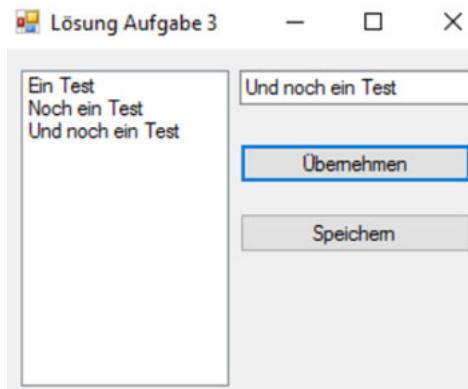


Abb. I.1: Das Formular der Anwendung

Hinweise:

Für eine einfache Verarbeitung der Daten im Listenfeld können Sie die Markierung schrittweise über die Eigenschaft `SelectedIndex` vom ersten bis zum letzten Element verschieben. Über die Eigenschaft `SelectedItem.ToString()` erhalten Sie dann den Wert des aktuell markierten Elements.

Ob die Datei korrekt angelegt wurde, können Sie überprüfen, indem Sie sie mit einem Doppelklick im Browser anzeigen lassen.

30 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit Dateien

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0919N01

CSHP13D

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit Dateien

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit Dateien

Inhaltsverzeichnis

Einleitung	1
1 Standardmethoden zum Laden und Speichern	3
1.1 Das Steuerelement RichTextBox	3
1.2 Ein einfacher Text-Editor	4
Zusammenfassung	9
2 Eine Textverarbeitung	11
2.1 Vorüberlegungen	11
2.2 Das Formular	12
2.3 Die Dateifunktionen	13
2.4 Die Formatierungsfunktionen	18
2.5 Funktionen für die Zwischenablage und das Rückgängigmachen	20
2.6 Direkter Zugriff auf das letzte Dokument	22
Zusammenfassung	28
3 Direkter Zugriff auf Dateien	30
3.1 Die Klassen StreamWriter und StreamReader	30
3.2 Die Klassen FileStream, BinaryWriter und BinaryReader	33
3.2.1 Die Klasse FileStream	33
3.2.2 Die Klassen BinaryWriter und BinaryReader	36
3.3 Die Bestenliste für das Pong-Spiel	42
Zusammenfassung	45
4 Weitere Datei- und Ordnerfunktionen	47
4.1 Suchen und Positionieren in einer Datei	47
4.2 Datei- und Ordnerverwaltung	52
4.3 Drucken	54
Zusammenfassung	60
Schlussbetrachtung	63

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	64
B.	Glossar	66
C.	Literaturverzeichnis	67
D.	Abbildungsverzeichnis	68
E.	Tabellenverzeichnis	69
F.	Codeverzeichnis	70
G.	Sachwortverzeichnis	71
H.	Einsendeaufgabe	73

Einleitung

Sie haben bereits gelernt, wie Sie Daten in der Registrierung von Windows und in XML-Dateien speichern und wieder zurücklesen.

In diesem Studienheft werden wir uns noch mit weiteren Techniken zur Verarbeitung von Dateien beschäftigen – zum Beispiel dem Laden und Speichern von Texten im Rich Text Format (RTF). Dabei handelt sich um ein weitverbreitetes Format für die Speicherung von Textdokumenten mit Formatierungen. Außerdem erfahren Sie, wie Sie über eigene Methoden beliebige Daten aus einer Anwendung in einer Datei speichern und wieder laden.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie eine Symbolleiste in ein Formular einfügen,
- wie Sie mit den Standardmethoden `LoadFile()` und `SaveFile()` Texte laden und speichern,
- wie Sie einen einfachen Texteditor mit einem Steuerelement **RichTextBox** programmieren,
- wie Sie das Schließen eines Formulars über das Ereignis **FormClosing** verhindern können,
- wie Sie Text in einer **RichTextBox** formatieren,
- wie Sie auf die Zwischenablage von Windows zugreifen,
- wie Sie Aktionen rückgängig machen und wiederherstellen,
- wie Sie zur Laufzeit neue Einträge in ein Menü einfügen,
- wie Sie einem Steuerelement zur Laufzeit eine Methode für ein bestimmtes Ereignis zuweisen,
- wie Sie Textdateien über die Klassen `StreamReader` und `StreamWriter` verarbeiten,
- wie Sie über die Klassen `FileStream`, `BinaryReader` und `BinaryWriter` Daten in binärer Form speichern und lesen,
- wie Sie nacheinander auf sämtliche Steuerelemente eines bestimmten Typs in einem Formular zugreifen,
- wie Sie in Dateien suchen,
- wie Sie auf eine beliebige Position innerhalb einer Datei direkt zugreifen,
- wie Sie das Ende einer Datei ermitteln,
- welche Methoden das .NET Framework für die Verwaltung von Ordnern und Dateien anbietet,
- wie Sie Druckaufträge erzeugen und ausgeben,
- wie Sie eine Druckvorschau anzeigen lassen und
- wie Sie ein Dokument mit mehreren Seiten ausdrucken.

Beginnen wir mit den Standardmethoden zum Laden und Speichern von Dateien.

Christoph Siebeck

1 Standardmethoden zum Laden und Speichern

In diesem Kapitel stellen wir Ihnen Standardmethoden zum Laden und Speichern vor.

1.1 Das Steuerelement RichTextBox

Bei einigen Steuerelementen wie zum Beispiel der **RichTextBox** des .NET Frameworks müssen Sie sich um die Dateihandhabung keine weiteren Gedanken machen. Sie können hier vorgefertigte Standardmethoden wie `LoadFile()` und `SaveFile()` verwenden, die Daten aus einer Datei laden und Daten in eine Datei speichern.

Schauen wir uns das an einem sehr einfachen Beispiel an. In einer Anwendung soll der Inhalt aus einer Datei in einer **RichTextBox** angezeigt werden und nach Änderungen auch wieder gespeichert werden.

Legen Sie bitte mit Visual Studio eine neue leere Windows Forms-Anwendung an. Fügen Sie dann in das Formular eine **RichTextBox** und zwei Schaltflächen ein. Vergrößern Sie anschließend die **RichTextBox** ein wenig.

Das Steuerelement **RichTextBox** finden Sie in der Toolbox in der Gruppe **Allgemeine Steuerelemente**.



Beim Anklicken der ersten Schaltfläche lassen wir einen Text laden und beim Anklicken der zweiten Schaltfläche soll der aktuelle Text in der **RichTextBox** gespeichert werden. Die entsprechenden Methoden sehen so aus:

```
private void ButtonLaden_Click(object sender, EventArgs e)
{
    //wenn die Datei existiert, in die RichTextBox
    //laden
    if (System.IO.File.Exists("C:\\test\\demo.rtf"))
    {
        //den alten Inhalt löschen
        richTextBox1.Clear();
        richTextBox1.LoadFile("C:\\test\\demo.rtf");
    }
    //sonst eine Meldung anzeigen
    else
        MessageBox.Show("Bitte speichern Sie zuerst einen Text.");
}

private void buttonSpeichern_Click(object sender,
EventArgs e)
{
    //den Text aus der RichTextBox speichern
    richTextBox1.SaveFile("C:\\test\\demo.rtf");
}
```

Code 1.1: Das Laden und Speichern über das Steuerelement **RichTextBox**

Hinweis:

Die genauen Namen der Methoden hängen von den Namen der Steuerelemente in Ihrem Projekt ab.

In der Methode `ButtonLaden_Click()` überprüfen wir zuerst mit der Methode `System.IO.File.Exists()`, ob die Datei **demo.rtf** im Ordner **C:\test** vorhanden ist. Wenn das der Fall ist, löschen wir über die Methode `Clear()` den Inhalt der **RichTextBox** und laden die Datei über die Methode `LoadFile()`.

In der Methode `ButtonSpeichern_Click()` schreiben wir den gesamten Inhalt aus der **RichTextBox** über die Methode `SaveFile()` in die Datei **demo.rtf** im Ordner **C:\test**.

**Bitte beachten Sie:**

Die Methode `SaveFile()` kann nur eine Datei anlegen, aber keinen Ordner. Falls der Ordner **C:\test** bei Ihnen nicht vorhanden ist, legen Sie ihn bitte selbst an beziehungsweise benutzen Sie einen anderen Pfad für den Test.

Neben dem Standardformat RTF kann das Steuerelement **RichTextBox** auch einfache Textdateien verarbeiten. So können Sie zum Beispiel mit minimalem Aufwand einen einfachen Text-Editor programmieren.

1.2 Ein einfacher Text-Editor

Schauen wir uns das in der Praxis an. Wir werden eine Anwendung erstellen, die Textdateien laden und speichern kann. Der Aufruf der Funktionen soll dabei über eine Symbolleiste erfolgen.

Legen Sie eine neue leere Windows Forms-Anwendung an und setzen Sie den Titel des Formulars auf **Editor**.

Damit wir die Symbolleiste frei positionieren können, fügen Sie im ersten Schritt das Steuerelement **ToolStripContainer**¹ ein. Sie finden es in der Toolbox unten in der Gruppe **Menüs & Symbolleisten**.

1. Wörtlich übersetzt bedeutet **ToolStrip** „Werkzeugleiste“.

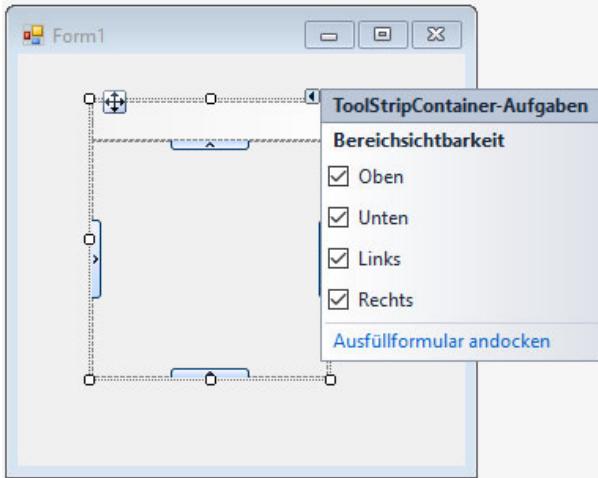


Abb. 1.1: Ein **ToolStripContainer** im Formular

Schalten Sie dann bei den ToolStripContainer-Aufgaben alle Bereichssichtbarkeiten bis auf den Eintrag **Oben** ab und klicken Sie auf **Ausfüllformular andocken** unten bei den Aufgaben.

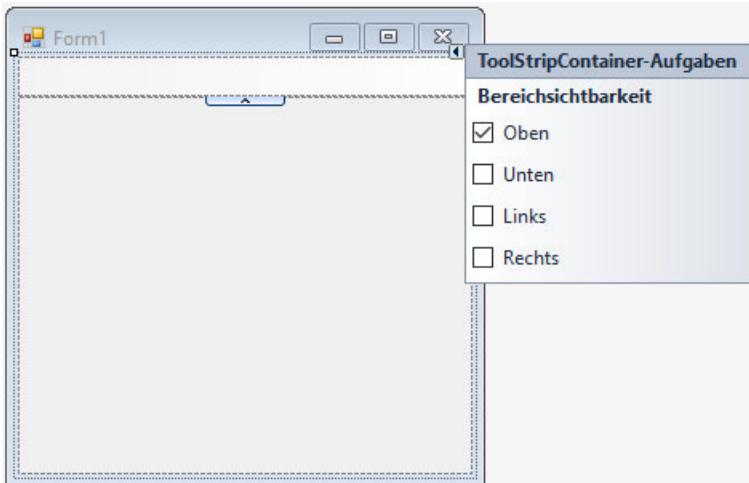


Abb. 1.2: Der angedockte **ToolStripContainer**

Der Container wird jetzt oben im Formular angedockt.

Hinweis:

Unter Umständen wird der Container verkleinert dargestellt. Um ihn zu vergrößern, stellen Sie den Mauszeiger auf das kleine Register mit dem Pfeil in der Mitte des Containers. Wenn der Mauszeiger als Hand mit einem Finger dargestellt wird, klicken Sie einmal mit der linken Maustaste. Dabei müssen Sie unter Umständen sehr genau zielen.

Im nächsten Schritt können Sie jetzt die eigentliche Symbolleiste im Container ablegen. Dazu verwenden Sie das Steuerelement **ToolStrip** aus der Gruppe **Menüs & Symbolleisten** der Toolbox.

Fügen Sie jetzt eine Symbolleiste in den Container ein. Das Formular sollte nun ungefähr so aussehen:

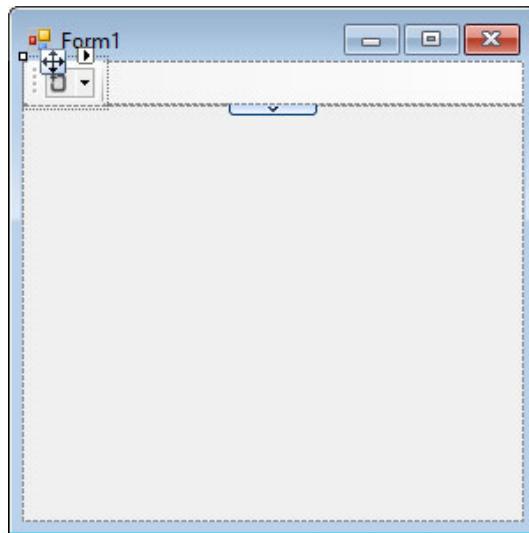


Abb. 1.3: Das Formular mit der Symbolleiste

Über das Kombinationsfeld in der Symbolleiste können Sie sehr einfach verschiedene Steuerelemente einfügen – zum Beispiel Schaltflächen, Labels und auch Eingabefelder. Anschließend setzen Sie wie gewohnt die Eigenschaften für diese Steuerelemente über das Eigenschaftenfenster. Für das Symbol verwenden Sie zum Beispiel die Eigenschaft **Image** in der Gruppe **Darstellung**. Den Text setzen Sie über die Eigenschaft **Text** in der Gruppe **Darstellung**. Dieser Text wird auch automatisch als Tooltipp beziehungsweise QuickInfo angezeigt, wenn Sie den Mauszeiger in der Anwendung einen Moment auf dem Symbol stehen lassen. Sie können den Tooltipp aber auch getrennt über die Eigenschaft **ToolTipText** in der Gruppe **Verhalten** festlegen.

In unserem Beispiel werden wir uns aber ein wenig Arbeit ersparen und mit Standardsymbolen arbeiten. Diese Symbole können Sie sehr einfach mit der Funktion **Standardelemente einfügen** erstellen. Sie finden diese Funktion im Kontext-Menü der Symbolleiste oder bei den ToolStrip-Aufgaben. Rufen Sie die Funktion jetzt bitte auf.

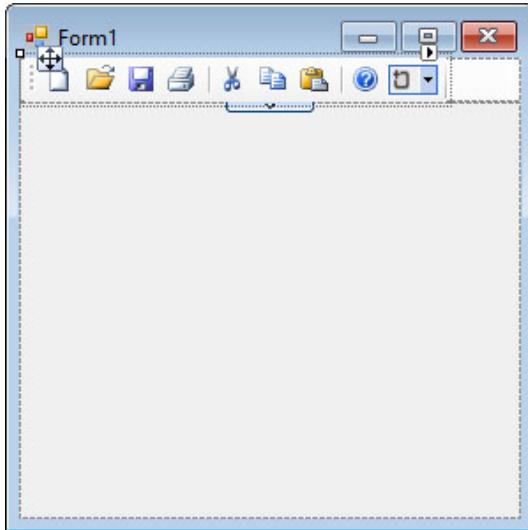


Abb. 1.4: Die eingefügten Standardsymbole

Die Symbole, die Sie nicht benötigen, können Sie ganz einfach wieder löschen. In unserem Beispiel benötigen wir nur das Symbol zum Öffnen und das Symbol zum Speichern. Die anderen Symbole können Sie also wieder entfernen.

Ergänzen Sie anschließend noch einen Standarddialog **Öffnen** und einen Standarddialog **Speichern unter**. Sie finden beide in der Toolbox in der Gruppe **Dialogfelder**. Das Steuerelement für den Öffnen-Dialog heißt **OpenFileDialog** und das Steuerelement für den Speichern-Dialog **SaveFileDialog**.

Fügen Sie dann das Steuerelement **RichTextBox** über den entsprechenden Eintrag in der Toolbox ein und setzen Sie die Eigenschaft **Dock** auf **Fill**. Das Formular im Editor sollte nun so aussehen:

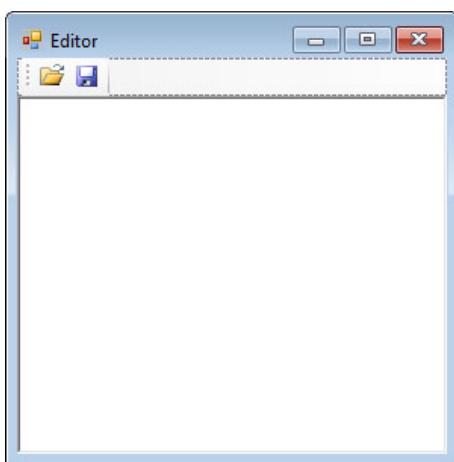


Abb. 1.5: Das Formular

Abschließend müssen wir jetzt noch die Eigenschaften für die Dialoge **Öffnen** und **Speichern unter** festlegen und dafür sorgen, dass eine Datei geladen beziehungsweise gespeichert wird. Legen Sie bitte für die beiden Dialoge einen Filter für Textdateien an und löschen Sie im Öffnendialog außerdem noch den Eintrag im Feld der Eigenschaft **FileName**.



Zur Auffrischung:

Bei einem Ausdruck für einen Filter geben Sie zuerst den Namen für den Filter an und dann den eigentlichen Filter. Der Name und der eigentliche Filter müssen dabei durch das Zeichen | getrennt werden.

Der Filter für Textdateien könnte zum Beispiel so aussehen: Textdateien|*.txt

Das eigentliche Laden beziehungsweise Speichern erfolgt dann im Ereignis **FileOK** des jeweiligen Dialogs. Dieses Ereignis tritt ein, wenn ein Dialog mit der Standardschaltfläche geschlossen wird. Wir rufen dort zuerst den Dateinamen ab, den der Anwender im Dialog ausgewählt beziehungsweise eingegeben hat, und übergeben den Namen als Argument an die Methoden `LoadFile()` beziehungsweise `SaveFile()` des RichTextBox-Steuerelements. Damit die Daten im Textformat gespeichert werden, übergeben wir als zweites Argument noch den Wert `RichTextBoxStreamType.PlainText`².



Ohne weitere Angaben werden Daten über das Steuerelement **RichTextBox** im Format RTF geladen und auch gespeichert. Wenn Sie ein anderes Format verwenden wollen, müssen Sie als zweites Argument einen Wert aus dem Aufzählungstyp `RichTextBoxStreamType` übergeben.

Beim Anklicken der beiden Symbole lassen wir dann den dazugehörigen Dialog modal anzeigen. Die Methoden sehen so aus:

```
private void ÖffnenToolStripButton_Click(object sender,
EventArgs e)
{
    //den Öffnendialog anzeigen
    openFileDialog1.ShowDialog();
}

private void OpenFileDialog1_FileOk(object sender,
CancelEventArgs e)
{
    //den Dateinamen beschaffen und die Datei dann laden
    string dateiname;
    dateiname = openFileDialog1.FileName;
    //bitte in einer Zeile eingeben
    richTextBox1.LoadFile(dateiname,
    RichTextBoxStreamType.PlainText);
}
```

2. *Plain* bedeutet übersetzt so viel wie „einfach, schlicht“.

```
private void SpeichernToolStripButton_Click(object sender,
EventArgs e)
{
    //den Speichern unter-Dialog anzeigen
    saveFileDialog1.ShowDialog();
}

private void SaveFileDialog1_FileOk(object sender,
CancelEventArgs e)
{
    //den Dateinamen beschaffen und die Datei dann speichern
    string dateiname;
    dateiname = saveFileDialog1.FileName;
    //bitte in einer Zeile eingeben
    richTextBox1.SaveFile(dateiname,
    RichTextBoxStreamType.PlainText);
}
```

Code 1.2: Die Methoden zum Laden und Speichern der Textdateien

Übernehmen Sie die Anweisungen aus dem vorigen Code in Ihre Anwendung. Probieren Sie den Text-Editor dann aus. Er kann problemlos beliebig lange Textdateien speichern und auch wieder laden.

Hinweis:

Wir haben den Editor hier bewusst sehr einfach gehalten. Im nächsten Kapitel werden wir ihn zu einer Minitextverarbeitung ausbauen, die unter anderem Textformatierungen unterstützt.

Sie finden den Text-Editor im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **Editor**.



So viel zunächst einmal zu den Standardmethoden zum Laden und Speichern.

Zusammenfassung

Standardmethoden zum Laden und Speichern von Daten gibt es unter anderem für das Steuerelement **RichTextBox**.

Das Steuerelement **RichTextBox** kann auch einfache Textdateien verarbeiten.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Mit welcher Methode laden Sie den Inhalt einer Datei in ein Steuerelement vom Typ **RichTextBox**? Mit welcher Methode speichern Sie den Inhalt einer **RichTextBox**? Was müssen Sie mindestens als Argument an die beiden Methoden übergeben?

- 1.2 In welchem Format werden die Daten einer **RichTextBox** in den Standardeinstellungen geladen und auch gespeichert?

- 1.3 Sie wollen in eine **RichTextBox** eine Textdatei **test.txt** laden. Wie lautet die entsprechende Anweisung?

2 Eine Textverarbeitung

In diesem Kapitel werden wir eine einfache Textverarbeitung programmieren. Diese Textverarbeitung soll unter anderem verschiedene Zeichen- und Absatzformate wie fett, kursiv, rechtsbündig oder Blocksatz unterstützen.

Da dieses Projekt etwas umfangreicher ist, stellen wir zunächst erst einmal wieder ein paar Vorüberlegungen an.

2.1 Vorüberlegungen

Unsere Anwendung soll sowohl Funktionen zur Dateihandhabung wie Öffnen, Speichern und Schließen als auch Funktionen zur Bearbeitung und Formatierung der Texte zur Verfügung stellen. Der Aufruf dieser Funktionen erfolgt über eine Menüleiste und eine Symbolleiste.

Für die Eingabe der Texte benutzen wir wie schon beim Editor im letzten Kapitel ein Steuerelement **RichTextBox**.

Damit sich das Programm windowskonform verhält und auch etwas betriebssicherer wird, werden wir bei der Dateihandhabung folgende Aspekte umsetzen:

Nur beim allerersten Speichern soll der Dialog **Speichern unter** erscheinen. Bei allen weiteren Speichervorgängen wird das Dokument direkt mit den aktuellen Einstellungen gesichert. Wir benötigen also zwei Methoden für das Speichern: eine Methode, die den Dialog **Speichern unter** anzeigt, und eine Methode, die das Dokument direkt speichert.

Diese beiden Methoden bieten wir zum einen im Menü **Datei** an, zum anderen aber auch über das Symbol **Speichern** unserer Anwendung. Beim Anklicken dieses Symbols müssen wir also unterscheiden können, ob das Dokument bereits einmal gespeichert wurde oder nicht. Dazu verwenden wir ein Feld `dateiname`, das zunächst eine leere Zeichenkette enthält. Beim Speichern weisen wir dem Feld dann den Dateinamen zu, den der Anwender im Dialog **Speichern unter** eingegeben hat. Anhand des vergebenen beziehungsweise nicht vergebenen Dateinamens können wir ohne Probleme zwischen den Zuständen „nicht gespeichert“ und „gespeichert“ unterscheiden.

Damit die Anwendung etwas komfortabler wird, zeigen wir den Dateinamen zusätzlich noch in der Titelleiste des Fensters an.

Beim Laden eines Dokuments soll zunächst geprüft werden, ob es in dem alten Dokument ungesicherte Änderungen gibt. Wenn das der Fall ist, soll eine Sicherheitsabfrage erscheinen, die es dem Anwender ermöglicht, die Daten zu speichern.

Um diese nicht gesicherten Änderungen feststellen zu können, fragen wir die Eigenschaft `Modified` der **RichTextBox** ab. Sie erhält automatisch den Wert `true`, wenn Änderungen am Text vorgenommen werden.

Zusätzlich lassen wir das zuletzt bearbeitete Dokument unten im Menü **Datei** anzeigen. Beim Anklicken dieses Eintrags soll direkt die entsprechende Datei geladen werden.



Die fertige Textverarbeitung finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **Minitext**.

Wie gewohnt werden wir das Projekt schrittweise immer weiter ausbauen. Beginnen wir mit dem Anlegen des Formulars.

2.2 Das Formular

Erstellen Sie bitte eine neue leere Windows Forms-Anwendung. Setzen Sie die Eigenschaft **Text** für das Formular auf `MiniText`, und lassen Sie es über die Eigenschaft **WindowState** maximiert darstellen.

Fügen Sie dann eine Menüleiste und eine Symbolleiste über die Steuerelemente **MenuStrip** und **ToolStrip** ein. Lassen Sie für beide Leisten die Standardelemente erstellen. In der Menüleiste löschen Sie anschließend bitte die Einträge **Extras** und **Hilfe** und die beiden Einträge **Drucken** und **Seitenansicht** im Menü **Datei**. Beim Eintrag **Beenden** im Menü **Datei** lassen Sie die Anwendung wie gewohnt über die Methode `Close()` schließen.

In der Symbolleiste entfernen Sie das Hilfesymbol ganz rechts und das Symbol mit dem Drucker.

Hinweis:

Die weiteren Einträge in den Menüs **Datei** und **Bearbeiten** werden wir später entsprechend anpassen.

Auf einen ToolStrip-Container zum Bewegen der Menü- und Symbolleisten verzichten wir. Wenn Sie möchten, können Sie diesen Container ja in Ihrem Projekt verwenden.

Fügen Sie anschließend ein RichTextBox-Steuerelement ein und lassen Sie es in der Größe des Formulars darstellen. Setzen Sie dazu die Eigenschaft **Dock** des RichTextBox-Steuerelements auf `Fill`.

Nehmen Sie abschließend noch einen Standarddialog **Öffnen** und einen Standarddialog **Speichern unter** in das Formular auf.

Das Formular sollte danach ungefähr so aussehen:



Abb. 2.1: Das Formular mit Menü- und Symbolleiste

2.3 Die Dateifunktionen

Im nächsten Schritt können wir jetzt die Funktionen zum Anlegen, Öffnen, Speichern beziehungsweise Speichern unter erstellen. Dazu verwenden wir auf der einen Seite Standardmethoden, auf der anderen Seite aber auch einige selbst erstellte Methoden.

Im ersten Schritt sorgen wir dafür, dass jedes neue Dokument einen „leeren“ Dateinamen erhält und eine Anzeige in der Titelleiste erscheint. Dabei soll neben dem Text „ohnename“ noch eine laufende Nummer angezeigt werden. Beim ersten neuen Dokument soll also der Text „ohnename1“ in der Titelleiste erscheinen, beim zweiten neuen Dokument der Text „ohnename2“ und so weiter. Die laufende Nummer ermitteln wir über einen Zähler, der bei jedem Anlegen eines neuen Dokuments um 1 erhöht wird.

Für den Zähler vereinbaren Sie bitte ein Feld `zaehler` vom Typ `int` in der Klasse `Form1`. Setzen Sie den Wert dieses Feldes im Konstruktor auf den Wert 1. Legen Sie außerdem ein Feld `dateiname` vom Typ `string` an.

Erstellen Sie dann eine Methode `NeuesDokument()`. In dieser Methode löschen Sie den Inhalt der `RichTextBox`, setzen den Dateinamen auf eine leere Zeichenkette und bauen die Anzeige in der Titelleiste aus dem Text „ohnename“ und dem Zähler zusammen. Die Methode könnte zum Beispiel so aussehen:

```
void NeuesDokument()
{
    //den Text "löschen"
    richTextBox1.Clear();
    //die Anzeige in der Titelleiste setzen
    this.Text = "ohnename" + zaehler;
    //den Zähler um 1 erhöhen
    zaehler++;
    //der Dateiname ist leer
    dateiname = string.Empty;
}
```

Code 2.1: Die Methode `NeuesDokument()`

Rufen Sie die Methode `NeuesDokument()` dann im Ereignis **Shown** des Formulars sowie beim Anklicken des Menüeintrags **Datei/Neu** und beim Anklicken des Symbols **Neu**  auf.

Sichern Sie anschließend die Änderungen und testen Sie das Programm. Sie sollten jetzt nahezu beliebig viele neue Dokumente mit fortlaufenden Nummern im Namen erzeugen können.

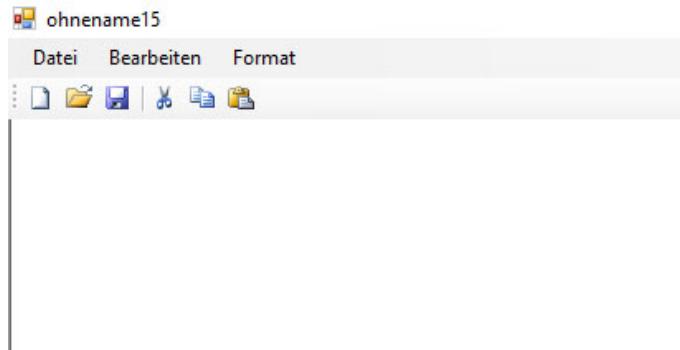


Abb. 2.2: Der Zähler beim Anlegen neuer Dokumente (oben in der Titelleiste)

Hinweis:

Wir legen keine echten neuen Dokumente an, sondern löschen lediglich den Inhalt der **RichTextBox**.

Kommen wir nun zum Speichern. Beim Anklicken der Funktion **Speichern** im Menü **Datei** oder beim Anklicken des Symbols **Speichern** überprüfen wir zuerst, ob der Dateiname für das Dokument noch leer ist, und zeigen dann entweder den Dialog **Speichern unter** an oder schreiben den Inhalt der **RichTextBox** direkt in eine Datei. Dazu verwenden wir die Standardmethode `SaveFile()`.

Beim Anklicken der Funktion **Speichern unter** im Menü **Datei** dagegen zeigen wir in jedem Fall den Dialog **Speichern unter** an.

Erstellen Sie jetzt bitte für das eigentliche Speichern eine eigene Methode, die den Wert des Feldes `dateiname` auf den Wert eines Arguments `name` setzt, den Namen dann in der Titelleiste anzeigt und anschließend den Inhalt des **RichTextBox**-Steuerelements unter diesem Namen speichert.

Die komplette Methode finden Sie im Code 2.2:

```
void Speichern(string name)
{
    dateiname = name;
    //den Text in der Titelleiste setzen
    this.Text = dateiname;
    //die Datei speichern
    richTextBox1.SaveFile(dateiname);
    //es sind keine Änderungen mehr vorhanden
    richTextBox1.Modified = false;
}
```

Code 2.2: Die Methode `Speichern()`

Mit der letzten Anweisung wird die Eigenschaft `Modified`³ der **RichTextBox** auf den Wert `false` gesetzt. Diese Eigenschaft erhält – wie Sie ja bereits wissen – automatisch den Wert `true`, sobald Änderungen am Inhalt des Feldes vorgenommen werden.

3. *Modified* bedeutet übersetzt so viel wie „geändert“.

Legen Sie jetzt bitte für den Dialog **Speichern unter** einen Filter für RTF-Dateien an. Geben Sie dann für das Ereignis **FileOK** des Dialogs `saveFileDialog1` die folgenden Anweisungen ein:

```
string name;
//den Namen beschaffen
name = saveFileDialog1.FileName;
//und die Datei über die eigene Methode speichern
Speichern(name);
```

Code 2.3: Die Anweisungen für die Methode `SaveFileDialog1_FileOk()`

Hier gibt es eigentlich keine Besonderheiten. Wir beschaffen uns den Namen, den der Anwender im Dialog eingetragen hat, und rufen unsere eigene Methode `Speichern()` mit diesem Namen als Argument auf.

Legen Sie nun für den Eintrag **Speichern** im Menü **Datei** fest, dass beim Anklicken eine Methode `DateiSpeichern()` ausgeführt werden soll. Dazu geben Sie den Namen der Methode ohne Klammern in das Feld hinter dem Ereignis **Click** ein. Visual Studio legt die Methode dann automatisch an.

Übernehmen Sie anschließend für die Methode `DateiSpeichern()` die Anweisungen aus dem Code 2.4:

```
//wenn der Dateiname noch leer ist, den Dialog Speichern
//unter aufrufen
if (dateiname == string.Empty)
    saveFileDialog1.ShowDialog();
//sonst unter demselben Namen wieder speichern
else
    Speichern(dateiname);
```

Code 2.4: Das Speichern

Auch hier gibt es keine Besonderheiten. Wenn der Dateiname leer ist, rufen wir den Dialog **Speichern unter** auf, andernfalls speichern wir die Datei direkt über unsere Methode `Speichern()`.

Sorgen Sie dann noch dafür, dass die Methode `DateiSpeichern()` auch beim Anklicken des Symbols **Speichern**  ausgeführt wird, und lassen Sie beim Anklicken des Menüeintrags **Datei/Speichern unter** den Dialog `saveFileDialog1` modal anzeigen.

Tipp:

Der Anwender muss im Dialog **Speichern unter** selbst eine Dateierweiterung angeben. Sie können aber auch eine Standarderweiterung über die Eigenschaft **DefaultExt** des Dialogs setzen. Diese Erweiterung wird immer dann benutzt, wenn der Anwender selbst keine eigene Erweiterung eingibt. Der trennende Punkt zwischen dem Namen und der Erweiterung wird dabei automatisch gesetzt.

Sichern Sie anschließend alle Änderungen und testen Sie das Programm.

Jetzt fehlt noch die Funktion zum Öffnen einer Datei.

Legen Sie bitte im ersten Schritt einen Filter für RTF-Dateien für den Öffnendialog an und löschen Sie den Eintrag bei der Eigenschaft **FileName**. Legen Sie dann für das Symbol **Öffnen** eine neue Methode `DateiLaden()` an. In dieser Methode überprüfen wir zunächst einmal, ob am aktuellen Text Änderungen vorliegen, und fragen nach, ob diese Änderungen gespeichert werden sollen. Danach lassen wir dann den Dialog **Öffnen** anzeigen.

Die vollständige Methode `DateiLaden()` finden Sie im Code 2.5:

```
private void DateiLaden(object sender, EventArgs e)
{
    //wenn nicht gespeicherte Änderungen vorliegen
    if (richTextBox1.Modified == true)
    {
        //Meldung mit Ja und Nein erzeugen und
        //überprüfen, ob Ja angeklickt wurde
        //bitte in einer Zeile eingeben
        if (MessageBox.Show("Wollen Sie die Änderungen
                           speichern?", "Abfrage",
                           MessageBoxButtons.YesNo,
                           MessageBoxIcon.Question) == DialogResult.Yes)
            DateiSpeichern(sender, e);
    }
    //den Öffnendialog anzeigen
    openFileDialog1.ShowDialog();
}
```

Code 2.5: Die Methode `DateiLaden()`

Falls der Anwender in der Sicherheitsabfrage die Schaltfläche **Ja** anklickt und `MessageBox.Show()` damit das Ergebnis `DialogResult.Yes` liefert, wird vor dem Anzeigen des Dialogs **Öffnen** noch die Methode `DateiSpeichern()` aufgerufen. Andernfalls erscheint der Dialog **Öffnen** sofort.

Das eigentliche Laden der Datei erfolgt dann im Ereignis **FileOK** des Öffnendialogs. Die Methode `OpenFileDialog1_FileOk()` muss so aussehen wie im Code 2.6.

```
private void OpenFileDialog1_FileOk(object sender,
CancelEventArgs e)
{
    //den Namen setzen
    dateiname = openFileDialog1.FileName;
    this.Text = dateiname;
    //die Datei laden
    richTextBox1.LoadFile(dateiname);
    //die Eigenschaft Modified zur Sicherheit auf false setzen
    richTextBox1.Modified = false;
}
```

Code 2.6: Die Methode `openFileDialog1_FileOk()`

Hier setzen wir den Wert des Feldes `dateiname` und zeigen den Namen in der Titelleiste an. Danach laden wir die Datei und setzen zur Sicherheit den Wert der Eigenschaft `Modified` selbst auf `false`.

Jetzt müssen wir noch die Öffnenfunktion über das Menü zur Verfügung stellen. Dazu lassen Sie beim Anklicken des entsprechenden Eintrags im Menü **Datei** die Methode `DateiLaden()` ausführen. Testen Sie dann die Änderungen. Nun sollte auch das Laden funktionieren.

Im letzten Schritt der Dateifunktionen stellen wir jetzt noch sicher, dass auch beim Schließen der Anwendung eine Abfrage erfolgt, ob Änderungen an dem Dokument gespeichert werden sollen. Dazu programmieren wir Anweisungen für das Ereignis **FormClosing**. Dieses Ereignis tritt ein, wenn das Formular geschlossen werden soll – allerdings noch vor dem eigentlichen Schließen. In dem Ereignis können Sie dann auch das Schließen des Formulars über das Setzen der Eigenschaft `e.Cancel` auf `true` abbrechen.

Die folgenden Anweisungen für das Ereignis **FormClosing** erzeugen zum Beispiel eine Meldung mit den drei Schaltflächen **Ja**, **Nein** und **Abbrechen**, falls Änderungen in dem Dokument nicht gespeichert wurden.

```
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    //wenn nicht gespeicherte Änderungen vorliegen
    if (richTextBox1.Modified == true)
    {
        //Meldung mit Ja, Nein und Abbrechen erzeugen
        //und auswerten
        //bitte in einer Zeile eingeben
        switch (MessageBox.Show("Wollen Sie die
Änderungen speichern?", "Abfrage",
MessageBoxButtons.YesNoCancel,
MessageBoxIcon.Question))
        {
            case DialogResult.Yes:
                //das Dokument speichern
                dateiSpeichern(sender, e);
                break;
            case DialogResult.Cancel:
                //Abbrechen
                e.Cancel = true;
                break;
        }
    }
}
```

Code 2.7: Die Methode `FormClosing()`

Klickt der Anwender auf die Schaltfläche **Ja**, werden die Änderungen gesichert und das Formular geschlossen. Klickt er dagegen auf **Nein**, wird das Formular sofort geschlossen. Interessant wird es beim Anklicken von **Abbrechen**. Hier wird `e.Cancel` auf `true` gesetzt. Das führt dazu, dass das Schließen abgebrochen wird. Das Formular bleibt also weiter geöffnet.

Übernehmen Sie auch diese Erweiterung und testen Sie das Programm noch einmal. Jetzt erscheint auch beim Schließen eine Abfrage, ob Änderungen gespeichert werden sollen.

Damit sind die Dateifunktionen für unsere Textverarbeitung zunächst einmal komplett. Im nächsten Schritt kümmern wir uns um die Formatierungsfunktionen für den Text.

2.4 Die Formatierungsfunktionen

Beginnen wir mit den Zeichenformatierungen.

Dazu weisen Sie der Eigenschaft `SelectionFont`⁴ der **RichTextBox** einen neuen Wert zu. Dieser Wert muss vom Typ `Font` sein und über `new` erzeugt werden. Um den Text in der **RichTextBox** fett zu formatieren, können Sie zum Beispiel die folgende Anweisung verwenden:

```
richTextBox1.SelectionFont = new
Font(richTextBox1.SelectionFont,
richTextBox1.SelectionFont.Style ^ FontStyle.Bold);
```

An den Konstruktor von `Font` wird dabei als erstes Argument die aktuelle Schrift von `richTextBox1` übergeben. Das zweite Argument verknüpft den aktuellen Schriftstil aus der Eigenschaft `richTextBox1.SelectionFont.Style` bitweise Exklusiv-Oder über den Operator `^` mit dem Wert `FontStyle.Bold`⁵.



Bei einer logischen Exklusiv-Oder-Verknüpfung ist das Ergebnis der Verknüpfung ausschließlich dann wahr, wenn nur einer der beiden Werte wahr ist. Wenn beide Werte wahr oder falsch sind, ist auch das Ergebnis falsch.

Über die bitweise Exklusiv-Oder-Verknüpfung können wir zum einen den Aufwand zum getrennten Ein- beziehungsweise Abschalten der Formatierungen sparen. Denn wenn eine Formatierung eingeschaltet ist, wird sie durch die bitweise Exklusiv-Oder-Verknüpfung wieder abgeschaltet. Außerdem können wir so auch mehrere Formatierungen gleichzeitig anwenden, da beim einzelnen Setzen der Formatierungen eventuell bereits vorhandene Formatierungen überschrieben würden.

Die Auswirkung der Formatierung hängt davon ab, ob gerade Text markiert ist oder nicht. Bei einem markierten Text wird die Formatierung nur auf die Markierung angewendet. Ist dagegen kein Text markiert, gilt die Formatierung so lange für alle neu eingegebenen Zeichen, bis sie wieder abgeschaltet wird. Dieses Verhalten kennen Sie wahrscheinlich auch von Ihrer Textverarbeitung.

4. `SelectionFont` bedeutet übersetzt „Schriftart der Auswahl“.

5. `Bold` bedeutet übersetzt „fett“.

Setzen wir die Zeichenformatierungen jetzt praktisch um. Fügen Sie in die Symbolleiste drei Symbole für die Formatierungen fett, kursiv und unterstrichen ein und legen Sie beim Anklicken des Symbols die gewünschte Formatierung fest. Die Methode für das Anklicken des Symbols **Fett** könnte zum Beispiel so aussehen:

```
private void FettToolStripButton_Click(object sender,
EventArgs e)
{
    //bitte in einer Zeile eingeben
    richTextBox1.SelectionFont = new
    Font(richTextBox1.SelectionFont,
    richTextBox1.SelectionFont.Style ^ FontStyle.Bold);
}
```

Code 2.8: Die Methode `FettToolStripButton_Click()`

Hinweis:

Der Name der Methode hängt vom Namen des Steuerelements ab.

Damit ein Symbol bei eingeschalteter Formatierung markiert wird, können Sie die Eigenschaft **CheckOnClick** in der Gruppe **Verhalten** des Eigenschaftenfensters auf **True** setzen.

Bilder für die Symbole finden Sie bei den Beispielen im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. Sie können sie als lokale Ressource importieren. Die entsprechenden Funktionen finden Sie im Dialog **Ressource auswählen**, der bei der Auswahl eines Symbols über die Eigenschaft **Image** erscheint.

Die beiden anderen Methoden stellen wir Ihnen hier nicht noch einmal vor, da sie sich kaum von der Methode `FettToolStripButton_Click()` unterscheiden. Sie müssen ja lediglich einen anderen Wert für die Zeichenformatierung benutzen. Für die kursive Formatierung benutzen Sie den Wert `FontStyle.Italic` und für das Unterstreichen den Wert `FontStyle.Underline`. Ausprogrammiert finden Sie die Methoden im Projekt **Minitext**. Sie finden es im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Weitere Zeichenformatierungen können Sie über den Dialog **Zeichen** durchführen. Sie finden ihn in der Toolbox in der Gruppe **Dialogfelder** unter der Bezeichnung **FontDialog**. Die Einstellungen aus dem Dialog erhalten Sie über die Eigenschaft `Font` des Dialogs.

Der folgende Code zeigt den Dialog **Zeichen** an und setzt beim Anklicken der Schaltfläche **OK** die Eigenschaften für den Text.

```
//wenn der Dialog Zeichen über OK geschlossen wurde, die
//Zeichenformatierungen setzen
if (fontDialog1.ShowDialog() == DialogResult.OK)
    richTextBox1.SelectionFont = fontDialog1.Font;
```

Code 2.9: Der Dialog **Zeichen** im Einsatz

Die beiden Anweisungen aus dem vorigen Code können Sie zum Beispiel beim Anklicken eines Eintrags **Zeichen** in einem Menü **Format** ausführen lassen.

So viel zu den Zeichenformatierungen.

Die Funktionen für die Absatzausrichtungen können Sie über sehr ähnliche Techniken erstellen. Im ersten Schritt erstellen Sie entsprechende Symbole in der Symbolleiste. Beim Anklicken setzen Sie dann den Wert der Eigenschaft `SelectionAlignment`⁶ für die RichTextBox auf einen Wert aus der Aufzählung `HorizontalAlignment`. Möglich sind dabei die Werte `Center` für zentriert, `Left` für linksbündig und `Right` für rechtsbündig.

Eine Methode für die zentrierte Absatzausrichtung könnte zum Beispiel so aussehen:

```
private void ZentriertToolStripButton_Click(object
    sender, EventArgs e)
{
    //zentriert formatieren
    //bitte in einer Zeile eingeben
    richTextBox1.SelectionAlignment =
        HorizontalAlignment.Center;
}
```

Code 2.10: Die Methode `ZentriertToolStripButton_Click()` für die zentrierte Ausrichtung

Hinweis:

Der Name der Methode hängt vom Namen des Steuerelements ab.

Die Methoden für die links- und rechtsbündige Ausrichtung haben wir hier nicht abgedruckt, da sie sich ebenfalls nur durch einen anderen Wert bei der Zuweisung unterscheiden. Ausprogrammiert finden Sie die Methoden im Quelltext des Projekts im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Das Steuerelement **RichTextBox** kennt noch zahlreiche weitere Formatierungsmöglichkeiten. So können Sie Texte zum Beispiel auch hoch- oder tiefstellen oder einen Absatz mit Einzügen versehen. Da sich die dabei eingesetzten Techniken aber kaum von den hier vorgestellten Techniken unterscheiden, stellen wir Ihnen das Vorgehen nicht weiter vor. Bitte sehen Sie bei Interesse in der Hilfe zum RichTextBox-Steuerelement nach.

2.5 Funktionen für die Zwischenablage und das Rückgängigmachen

Die noch fehlenden Funktionen aus dem Menü **Bearbeiten** lassen sich mit recht wenig Aufwand umsetzen, da Sie fertige Methoden des Steuerelements **RichTextBox** benutzen können.

Für das Ausschneiden und Kopieren in die Zwischenablage benutzen Sie zum Beispiel die Methoden `Cut()` und `Copy()`. Das Einfügen aus der Zwischenablage erfolgt mit der Methode `Paste()`.

Um Aktionen rückgängig zu machen und rückgängig gemachte Aktionen wiederzuholen, stehen Ihnen die Methoden `Undo()` und `Redo()` des Steuerelements **RichTextBox** zur Verfügung. Zur Sicherheit sollten Sie bei beiden Methoden vorher überprüfen,

6. `SelectionAlignment` bedeutet übersetzt so viel wie „Ausrichtung der Auswahl“.

ob überhaupt Aktionen zum Rückgängigmachen beziehungsweise zum Wiederholen verfügbar sind. Dazu verwenden Sie die Eigenschaften `CanUndo` beziehungsweise `CanRedo`.

Die Methoden für die Funktionen **Rückgängig**, **Wiederholen**, **Ausschneiden**, **Kopieren** und **Einfügen** im Menü **Bearbeiten** könnten dann so aussehen:

```
private void RückgängigToolStripMenuItem_Click(object sender, EventArgs e)
{
    //kann etwas rückgängig gemacht werden?
    if (richTextBox1.CanUndo == true)
        richTextBox1.Undo();
}

private void WiederholenToolStripMenuItem_Click(object sender, EventArgs e)
{
    //kann etwas wiedergeholt werden?
    if (richTextBox1.CanRedo == true)
        richTextBox1.Redo();
}

private void AusschneidenToolStripMenuItem_Click(object sender, EventArgs e)
{
    //Ausschneiden
    richTextBox1.Cut();
}

private void KopierenToolStripMenuItem_Click(object sender, EventArgs e)
{
    //Kopieren
    richTextBox1.Copy();
}

private void EinfügenToolStripMenuItem_Click(object sender, EventArgs e)
{
    richTextBox1.Paste();
}
```

Code 2.11: Die Methoden für die Funktionen im Menü **Bearbeiten**

Hinweis:

Auch hier hängen die genauen Namen der Methoden wieder von den Namen der Steuerelemente ab.

Jetzt bleibt nur noch die letzte Funktion im Menü **Bearbeiten** – nämlich **Alle auswählen**. Hier können Sie die Methode `SelectAll()` des Steuerelements **RichTextBox** benutzen. Sie markiert den gesamten Text im Feld.

Da das Umsetzen der verschiedenen Funktionen eigentlich nur Fleißarbeit ist, stellen wir Ihnen die einzelnen Schritte hier nicht im Detail vor. Denken Sie aber bitte daran, die Funktionen zum Ausschneiden, Kopieren und Einfügen auch noch den entsprechenden Symbolen zuzuweisen.

2.6 Direkter Zugriff auf das letzte Dokument

Abschließend wollen wir noch den direkten Zugriff auf das zuletzt bearbeitete Dokument programmieren.

Dazu legen wir beim Schließen der Anwendung einen Eintrag in der Registrierung an, der den Namen des zuletzt angezeigten Dokuments enthält. Beim Starten der Anwendung lesen wir den Namen des Dokuments wieder aus und erzeugen dynamisch einen neuen Eintrag im Menü **Datei**. Beim Anklicken dieses Eintrags soll das Dokument angezeigt werden.

Das Speichern des Eintrags ist nicht weiter schwierig. Wir kontrollieren, ob das aktuelle Dokument einen Dateinamen hat, und schreiben dann diesen Namen in die Registrierung. Diese Anweisungen sollen beim Schließen der Anwendung ausgeführt werden – also im Ereignis **FormClosed**. Die entsprechende Methode sieht so aus:

```
private void Form1_FormClosed(object sender,
    FormClosedEventArgs e)
{
    //hat das aktuelle Dokument einen Namen?
    if (dateiname != string.Empty)
    {
        //den Schlüssel HKEY_CURRENT_USER anlegen bzw. öffnen
        //bitte in einer Zeile eingeben
        using (RegistryKey regSchluessel =
            Registry.CurrentUser.CreateSubKey
            ("Software\\Minitext"))
        {
            //den Dateinamen in die Registrierung schreiben
            regSchluessel.SetValue("Datei", dateiname);
        }
    }
}
```

Code 2.12: Das Speichern des Dateinamens in der Registrierung

Hinweis:

Bitte denken Sie an das Bekanntgeben des Namensraums `Microsoft.Win32`. Andernfalls erhalten Sie Fehlermeldungen, da die Klassen `RegistryKey` und `Registry` nicht bekannt sind.

Das Zurücklesen des Dateinamens erfolgt dann beim Anzeigen des Formulars – also im Ereignis **Load**.

Wir überprüfen zunächst, ob es überhaupt einen passenden Eintrag in der Registrierung gibt. Wenn das der Fall ist, lesen wir den Wert aus und erzeugen über die Klasse `ToolStripMenuItem` einen neuen Menüeintrag. Den neuen Eintrag fügen wir anschließend im Menü **Datei** direkt oberhalb der Funktion **Beenden** ein. Beim Anklicken des Menüeintrags rufen wir eine Methode auf, die versucht, das Dokument zu öffnen.

Der kniffligste Teil ist dabei das Anlegen des neuen Menüeintrags und die Auswertung des Namens, der sich hinter diesem Menüeintrag verbirgt. Denn dabei gibt es einige Besonderheiten. Schauen wir uns zuerst an, wie ein neuer Menüeintrag angelegt wird.

Dazu erstellen Sie im ersten Schritt eine Variable für die Klasse `ToolStripMenuItem`. Anschließend erzeugen Sie über den Konstruktor der Klasse `ToolStripMenuItem` einen neuen Menüeintrag und weisen diesen Eintrag der Variablen zu. Der Konstruktor erwartet dabei folgende Argumente:

- den Text für den Eintrag als Typ `string`,
- ein Bild, das eventuell vor dem Eintrag angezeigt werden soll, und
- den Eventhandler für das Ereignis, das beim Anklicken des Eintrags ausgeführt werden soll.

Die ersten beiden Argumente sind schnell gefunden. Den Text des Eintrags setzen wir aus einer laufenden Nummer und dem Namen der Datei zusammen. Ein Bild verwenden wir nicht. Deshalb übergeben wir hier den Wert `null`.

Nicht ganz so einfach ist das dritte Argument – also der Eventhandler. Hier müssen Sie nämlich die Verbindung zwischen dem Ereignis – also dem Klicken – und der Methode herstellen, die beim Klicken ausgeführt wird.

Ein Eventhandler stellt die Verbindung zwischen einem Ereignis und der Methode für dieses Ereignis her.



Was sich vielleicht etwas kompliziert liest, ist eigentlich nicht weiter schwierig – denn Sie können die Vereinbarung eines Eventhandlers quasi „abschreiben“. Sehen Sie sich dazu einmal die Anweisungen in der Klasse `Form1` an, die Visual Studio automatisch für die verschiedenen Steuerelemente erstellt. Sie finden diese Anweisungen in der Datei `Form1.Designer.cs`. Beim Steuerelement `speichernToolStripMenuItem` finden Sie ganz unten zum Beispiel die Zeile

```
this.speichernToolStripMenuItem.Click += new  
System.EventHandler(this.DateiSpeichern);
```

Der Eventhandler steht dabei hinter dem Zuweisungsoperator. Hier wird über den Konstruktor des Delegaten `System.EventHandler` eine Verbindung zwischen dem Ereignis `Click` für den Menüeintrag und der Methode `DateiSpeichern()` in der Klasse `Form1` hergestellt.

Ein Delegat stellt einen Verweis auf eine Methode dar.





Bitte beachten Sie:

Beim Erstellen eines Eventhandlers teilen Sie dem Compiler lediglich mit, welche Methode aufgerufen werden soll. Sie dürfen die Methode hier nicht aufrufen. Daher dürfen im Konstruktor des Delegaten `EventHandler` hinter dem Namen der Methode auch keine Klammern stehen.

Die Anweisungen für die verschiedenen Steuerelemente werden in der Datei `Form1.Designer.cs` unter Umständen nicht sofort angezeigt. Klicken Sie dann einmal auf das Symbol vor der Zeile **Vom Windows Form-Designer generierter Code**.

Nehmen Sie bitte keine Änderungen an den Anweisungen in der Datei `Form1.Designer.cs` vor. Möglicherweise funktioniert danach das gesamte Projekt nicht mehr richtig.

Das Erzeugen des neuen Menüeintrags könnte in unserem Beispiel also so aussehen:

```
menuEintrag = new ToolStripMenuItem("&1 - " + name,
null, new EventHandler(this.LetzteDatei_Click));
```

Hinweis:

Über das Zeichen `&` vor der 1 ermöglichen wir die Auswahl des Eintrags über die Tastatur.

Die eigentliche Methode, die beim Anklicken des Menüeintrags ausgeführt wird, erstellen wir gleich.

Wenn Ihnen das Erstellen des Eventhandlers über den Konstruktor zu kompliziert erscheint, können Sie die Verbindung auch über die Eigenschaft `Click` des neuen Eintrags herstellen. Löschen Sie dazu das dritte Argument im Konstruktor und ergänzen Sie dann direkt nach dem Erstellen des neuen Menüeintrags die folgende Anweisung:

```
menuEintrag.Click += new
EventHandler(this.LetzteDatei_Click);
```

Die Wirkung ist identisch.

Das eigentliche Einfügen des Menüeintrags erfolgt dann entweder mit der Methode `DropDownItems.Add()` oder `DropDownItems.Insert()` für das Menü. Die Methode `Add()` fügt den Eintrag dabei immer am Ende an. Bei der Methode `Insert()` können Sie die gewünschte Position als Typ `int` selbst angeben.

In unserem Fall soll der neue Eintrag an der vorletzten Position eingefügt werden. Wir verwenden also die Methode `Insert()` und ermitteln über die Eigenschaft `DropDownItems.Count` des Menüs die Anzahl der bereits vorhandenen Einträge. Daraus ziehen wir den Wert 1 ab und landen so an der gewünschten Position im Menü.

Damit alles seine Ordnung hat, fügen wir noch über die Methode `Insert()` eine Trennlinie hinter dem neuen Eintrag ein. Als zweites Argument übergeben wir dabei an die Methode eine neue Instanz der Klasse `ToolStripSeparator`. Die gesamte Konstruktion sieht so aus:

```
//für den Menüeintrag
ToolStripMenuItem menuEintrag;
//den neuen Menüeintrag erzeugen
//bitte jeweils in einer Zeile eingeben
menuEintrag = new ToolStripMenuItem("&1 - " + name,
null, new EventHandler(this.LetzteDatei_Click));
//und einfügen
dateiToolStripMenuItem.DropDownItems.Insert
(dateiToolStripMenuItem.DropDownItems.Count - 1, menuEintrag);
//und noch eine Trennlinie einfügen
dateiToolStripMenuItem.DropDownItems.Insert
(dateiToolStripMenuItem.DropDownItems.Count - 1, new
ToolStripSeparator());
```

Code 2.13: Das dynamische Einfügen eines Eintrags im Menü **Datei**

Bitte beachten Sie:

Beim Erzeugen des neuen Eintrags wird der Eintrag nicht automatisch einem Menü zugewiesen. Das erfolgt durch den Aufruf der Methode `DropDownItems.Insert()` für das jeweilige Menü. Dieses Menü sprechen Sie über den internen Namen an und nicht über den Text des Menüs. In unserem Beispiel lautet der interne Name `dateiToolStripMenuItem`.



Da der Compiler bei der Anweisung zum Anlegen des neuen Menüeintrags überprüft, ob die angegebene Methode vorhanden ist, wollen wir jetzt zunächst einmal die Methode `LetzteDatei_Click()` programmieren. Dabei gibt es eigentlich nur eine Besonderheit: Sie müssen für die Methode zwingend die Parameter `object sender` und `EventArgs e` vereinbaren. Diese Vereinbarung ist auch dann erforderlich, wenn Sie die Parameter gar nicht verarbeiten wollen.

In der Methode selbst holen wir uns den Namen der Datei aus der Eigenschaft `Text` des angeklickten Menüeintrags zurück. Dazu wandeln wir den Wert von `sender` über einen Cast in ein Objekt vom Typ `ToolStripMenuItem` um und rufen dann für dieses Objekt den Wert der Eigenschaft `Text` ab.

Der Text enthält nun aber nicht nur den eigentlichen Dateinamen, sondern auch noch das Zeichen `&` als Markierung für die Taste und die Zeichen „`1 -`“. Dabei handelt es sich um vier Zeichen, da vor und hinter dem Strich jeweils ein Leerzeichen steht. Diese Zeichen müssen wir über die Methode `Remove()` wieder löschen. Danach können wir dann den Inhalt der Datei laden.

Die vollständige Methode `LetzteDatei_Click()` sieht dann so aus:

```
private void LetzteDatei_Click(object sender, EventArgs e)
{
    string name;
    //den Namen zurücklesen
    name = ((ToolStripMenuItem)(sender)).Text;
    //die ersten fünf(!!) Zeichen wieder löschen
    name = name.Remove(0, 5);
    //gibt es die Datei noch?
    if (System.IO.File.Exists(name))
    {
        //den Dateinamen setzen
        dateiname = name;
        //in der Titelleiste anzeigen
        this.Text = dateiname;
        //die Datei laden
        richTextBox1.LoadFile(dateiname);
        //die Eigenschaft Modified zur Sicherheit auf
        //false setzen
        richTextBox1.Modified = false;
    }
}
```

Code 2.14: Die Methode `LetzteDatei_Click()`



Zur Auffrischung:

Die Methode `Remove()` für den Typ `string` erwartet als erstes Argument die Position, ab der gelöscht werden soll, und als zweites Argument die Anzahl der Zeichen, die gelöscht werden sollen. Das Zählen der Position beginnt dabei mit 0.

Hinweis:

Die Überprüfung, ob die Datei vorhanden ist, mag auf den ersten Blick überflüssig erscheinen, ist aber trotzdem wichtig. Möglicherweise hat der Anwender die Datei ja in der Zwischenzeit gelöscht, umbenannt oder verschoben. Und dann würde das Programm ohne die Überprüfung eine Ausnahme auslösen.

Sie können sich das aufwendige Umbauen der Eigenschaft `Text` zum Dateinamen auch sparen, wenn Sie in der Methode `LetzteDatei_Click()` kurzerhand noch einmal den Dateinamen aus der Registrierung lesen und dann diesen Namen für das Öffnen der Datei verwenden. Das funktioniert bei einem einzigen Eintrag auch sehr gut, führt allerdings bei mehreren Einträgen wieder zu neuen Problemen. Denn dann müssen Sie herausfinden, auf welchen der Einträge denn nun geklickt wurde, um auch den passenden Eintrag in der Registrierung zu lesen. Sie sehen selbst, manch scheinbarer Ausweg ist mit etwas Voraussicht dann doch keiner.

Sie können das Laden auch etwas kompakter gestalten, indem Sie eine eigene Methode erstellen. Dann müssen Sie zum Beispiel die Anweisungen für das Setzen des Dateinamens und zur Anzeige in der Titelleiste nicht mehrfach in identischer Form verwenden.

Um unsere Textverarbeitung zu vollenden, fehlen jetzt noch die Anweisungen für die Methode `Form1_Load()`. Hier lesen wir den Eintrag der zuletzt bearbeiteten Datei aus der Registrierung aus und erzeugen den neuen Menüeintrag. Die vollständige Methode sieht so aus:

```
private void Form1_Load(object sender, EventArgs e)
{
    //für den Menüeintrag
    ToolStripMenuItem menuEintrag;
    //für den Namen des Eintrags
    string name;
    //den Namen zur Sicherheit leer initialisieren
    name = string.Empty;
    //den Schlüssel HKEY_CURRENT_USER öffnen
    //bitte in einer Zeile eingeben
    using (RegistryKey regSchluessel =
        Registry.CurrentUser.OpenSubKey("Software\\Minitext"))
    {
        //ist der Schlüssel vorhanden?
        if (regSchluessel != null)
        {
            //bitte jeweils in einer Zeile eingeben
            //lesen und zuweisen
            name = Convert.ToString
                (regSchluessel.GetValue("Datei"));
            //den neuen Menüeintrag erzeugen
            menuEintrag = new ToolStripMenuItem("&1 - " +
                name, null, new EventHandler
                (this.LetzteDatei_Click));
            //und einfügen
            dateiToolStripMenuItem.DropDownItems.Insert
                (dateiToolStripMenuItem.DropDownItems.Count - 1,
                menuEintrag);
            //und noch eine Trennlinie einfügen
            dateiToolStripMenuItem.DropDownItems.Insert
                (dateiToolStripMenuItem.DropDownItems.Count - 1,
                new ToolStripSeparator());
        }
    }
}
```

Code 2.15: Die Methode `Form1_Load()`

Übernehmen Sie jetzt die Erweiterungen für das Abrufen der zuletzt bearbeiteten Datei in das Projekt und testen Sie das Programm dann.

Damit beenden wir die Programmierung der Textverarbeitung – auch wenn an der einen oder anderen Stelle sicherlich noch ein wenig Feinschliff erforderlich wäre. Sie können das Programm ja auch selbst weiter verfeinern und zusätzliche Funktionen hinzufügen. Versuchen Sie zum Beispiel einmal, im Menü **Datei** nicht nur die letzte bearbeitete Datei anzubieten, sondern mehrere. Außerdem könnten Sie auch beim Neuanlegen von Dokumenten überprüfen, ob möglicherweise nicht gesicherte Änderungen vorliegen. Denn jetzt wird ja das alte Dokument einfach gelöscht – auch dann, wenn Änderungen nicht gespeichert wurden.

Zusammenfassung

Das Steuerelement **RichTextBox** stellt Ihnen zahlreiche Formatierungsmöglichkeiten zur Verfügung.

Sie können neue Menüeinträge dynamisch zur Laufzeit eines Programms erstellen lassen. Dazu erzeugen Sie eine neue Instanz der Klasse `ToolStripMenuItem`.

Ein Eventhandler stellt die Verbindung zwischen einem Ereignis und der Methode für das Ereignis her.

Aufgaben zur Selbstüberprüfung

- 2.1 Sie wollen in einer **RichTextBox** Zeichen fett formatieren. Formulieren Sie bitte eine Anweisung, mit der Sie die Formatierung sowohl ein- als auch ausschalten können.

- 2.2 Mit welchen Methoden der Klasse **RichTextBox** können Sie Text in die Zwischenablage kopieren beziehungsweise ausschneiden?

- 2.3 Sie wollen zur Laufzeit eines Programms einen neuen Eintrag in einem Menü einfügen. Beim Anklicken des Eintrags soll eine Methode `Neu_Click()` ausgeführt werden. Formulieren Sie bitte eine entsprechende Anweisung. Der Name des Eintrags ist dabei beliebig. Ein Bild soll der Eintrag nicht erhalten.

- 2.4 Wie können Sie die Anzahl der Einträge in einem Menü **Datei** ermitteln? Das Steuerelement für das Menü trägt den Namen **dateiToolStrip**.

3 Direkter Zugriff auf Dateien

In diesem Kapitel lernen Sie, wie Sie direkt Daten in eine Datei schreiben und wieder zurücklesen. Das ist zum Beispiel dann interessant, wenn Sie Werte aus einem Formular sichern möchten und das entsprechende Steuerelement keine Methoden zum Speichern und Laden der Daten kennt.

3.1 Die Klassen StreamWriter und StreamReader

Für das Verarbeiten von Textdateien verwenden Sie am einfachsten die Klassen `StreamWriter` und `StreamReader`. Die Klasse `StreamWriter` ist dabei für das Erstellen der Datei und das Schreiben der Daten zuständig, die Klasse `StreamReader` übernimmt das Lesen der Daten. Beide Klassen befinden sich im Namensraum `System.IO`.



Ein Stream (übersetzt „Strom, Bach“) ist ein unformatierter Datenstrom. Die Interpretation erfolgt durch das Zielgerät, das den Stream verarbeitet.

Schauen wir uns die beiden Klassen im praktischen Einsatz an. Beginnen wir mit dem Speichern. Im folgenden Code wird der Inhalt eines Listenfelds in eine Datei `daten.txt` im Ordner C:\test geschrieben.

```
private void ButtonSpeichern_Click(object sender,
EventArgs e)
{
    //eine neue Instanz von StreamWriter erzeugen
    //bitte in einer Zeile eingeben
    StreamWriter textDatei = new
    StreamWriter("c:\\test\\\\daten.txt");
    //alle Daten in der ListBox in die Datei schreiben
    for (int i = 0; i < listBox1.Items.Count; i++)
        textDatei.WriteLine(listBox1.Items[i].ToString());
    //die Datei wieder schließen
    textDatei.Close();
}
```

Code 3.1: Das Speichern von Texten aus einem Listenfeld

Zunächst einmal erzeugen wir mit der Anweisung

```
StreamWriter textDatei = new
StreamWriter("c:\\test\\\\daten.txt");
```

eine Instanz `textDatei` der Klasse `StreamWriter`. An den Konstruktor von `StreamWriter` übergeben wir dabei den Namen der Datei, die verarbeitet werden soll.

Dann schreiben wir alle Zeilen in dem Listenfeld über die Methode `WriteLine()` in die Datei. Das übernimmt die Schleife

```
for (int i = 0; i < listBox1.Items.Count; i++)
    textDatei.WriteLine(listBox1.Items[i].ToString());
```

Danach wird die Datei über die Methode `Close()` wieder geschlossen. Bitte beachten Sie auch hier, dass vor allem beim Schreiben von Daten das Schließen sehr wichtig ist. Denn erst dann können Sie sicher sein, dass die Daten wirklich in die Datei geschrieben werden.

Hinweis:

Neben der Methode `WriteLine()` können Sie Daten auch über die Methode `Write()` der Klasse `StreamWriter` in die Datei schreiben. Dabei wird aber kein Zeilenumbruch angehängt.

Wenn die Datei, die Sie über `StreamWriter` verarbeiten wollen, nicht vorhanden ist, wird sie automatisch angelegt.

In der Standardeinstellung wird der Inhalt der Datei immer überschrieben. Um neue Zeilen an den bereits vorhandenen Inhalt anzuhängen, geben Sie beim Konstruktor zusätzlich noch den Wert `true` an.

Die Klasse `StreamWriter` verarbeitet Texte im Format UTF-8 – einer Variante des Unicode-Zeichensatzes. Wenn Sie ein anderes Format benutzen möchten, müssen Sie es explizit im Konstruktor angeben. Die folgende Anweisung würde zum Beispiel die Datei `C:\test\daten.txt` öffnen beziehungsweise anlegen, neue Zeilen an das Ende anhängen und die Zeichen als ASCII interpretieren:

```
StreamWriter textDatei = new  
StreamWriter("C:\\test\\daten.txt", true,  
System.Text.Encoding.ASCII);
```

Weitere Hinweise zu den verschiedenen Konstruktoren der Klasse `StreamWriter` und auch zu den Angaben für die Codierung finden Sie in der Hilfe.

Bitte beachten Sie:

Auch bei den Klassen `StreamReader` und `StreamWriter` empfiehlt Microsoft ausdrücklich, alle Ressourcen direkt nach dem Zugriff wieder freizugeben. Dazu können Sie wieder die Methode `Dispose()` in einer `try ... catch ... finally`-Konstruktion aufrufen oder eine `using`-Konstruktion benutzen. Wir benutzen in den folgenden Beispielen der Einfachheit halber immer eine `using`-Konstruktion. Dabei wird nach dem Verlassen des Blocks auch die Datei automatisch geschlossen. Wir müssen also die Methode `Close()` nicht selbst aufrufen.



Das Lesen erfolgt mit einer sehr ähnlichen Technik. Im ersten Schritt legen Sie eine neue Instanz der Klasse `StreamReader` an und übergeben den Namen der gewünschten Datei als Argument an den Konstruktor. Anschließend können Sie Daten zum Beispiel mit der Methode `ReadLine()` zeilenweise lesen. Das Ende der Datei ermitteln Sie dabei über die Eigenschaft `EndOfStream` der Klasse `StreamReader`. Eine Methode zum Einlesen aller Zeilen in einer Textdatei ein Listenfeld könnte dann so aussehen:

```
private void ButtonLaden_Click(object sender, EventArgs e)  
{  
    //existiert die Datei?  
    if (File.Exists("c:\\test\\daten.txt"))
```

```
{
    //eine neue Instanz von StreamReader erzeugen
    //bitte in einer Zeile eingeben
    using (StreamReader textDatei = new
    StreamReader("c:\\test\\daten.txt"))
    {
        //für die Zeilen in der Datei
        string zeile;
        //die ListBox leeren
        listBox1.Items.Clear();
        //solange das Ende der Datei nicht erreicht ist,
        //die Daten lesen und in die ListBox schreiben
        while (textDatei.EndOfStream == false)
        {
            zeile = textDatei.ReadLine();
            listBox1.Items.Add(zeile);
        }
    }
}
```

Code 3.2: Das Einlesen aus einer Textdatei in ein Listenfeld

Mit der ersten Anweisung überprüfen wir, ob die Datei überhaupt vorhanden ist. Wenn das der Fall ist, erzeugen wir im `using`-Block eine Instanz der Klasse `StreamReader` und lesen dann so lange zeilenweise Daten ein, bis das Ende der Datei erreicht ist. Das erledigt die Schleife

```
while (textDatei.EndOfStream == false)
{
    zeile = textDatei.ReadLine();
    listBox1.Items.Add(zeile);
}
```

Hinweis:

Die Klasse `StreamReader` kennt noch einige weitere Anweisungen zum Lesen von Daten. So können Sie zum Beispiel mit der Methode `ReadBlock()` Daten in Blöcken lesen oder über die Methode `ReadToEnd()` alle Daten von der aktuellen Position bis zum Ende der Datei in einem Rutsch einlesen. Details zu diesen Methoden finden Sie in der Hilfe der Klasse `StreamReader`.

Auch die Klasse `StreamReader` geht ohne weitere Angaben davon aus, dass die Daten in der Datei im Format UTF-8 interpretiert werden sollen. Sie können aber über den Konstruktor – genau wie bei der Klasse `StreamWriter` – auch ein anderes Format angeben.

Neben der festen Pfadangabe der Datei im Konstruktor der Klassen `StreamReader` und `StreamWriter` können Sie natürlich auch Variablen benutzen – also zum Beispiel den Namen aus einem **Öffnen-** oder **Speichern unter-**Dialog.

Bitte denken Sie beim Einsatz der Klassen `StreamReader` und `StreamWriter` an die Bekanntgabe des Namensraums `System.IO`.

Ein vollständiges Beispiel für das Speichern und Lesen von Daten über ein Listenfeld finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **ListBoxDemo**.



3.2 Die Klassen FileStream, BinaryWriter und BinaryReader

Die Klassen `StreamReader` und `StreamWriter` sind zwar sehr komfortabel, haben allerdings einen großen Nachteil: Sie sind vor allem für die Verarbeitung von Zeichen ausgelegt. Sie können so zwar auch Zahlen speichern, müssen die Werte dann aber spätestens beim Einlesen mühselig wieder konvertieren.

Sehr viel flexibler sind dagegen die Klassen `BinaryWriter` und `BinaryReader`. Sie speichern beliebige Daten im binären Format – also im ursprünglichen Bitmuster. Dabei wird allerdings nicht jedes Bit einzeln gespeichert, sondern die Bytes. Die Zahl 1 als `int`-Typ würde im binären Format zum Beispiel so aussehen:

00 00 00 01

Dargestellt werden hier die Werte von vier Bytes – also 32 Bits. Und das entspricht – wenn Sie noch einmal etwas weiter zurückdenken – genau der Größe des Datentyps `int`. Als `short`-Typ dagegen würde die Zahl 1 im binären Format so aussehen:

00 01

Hinweis:

Intern werden die Daten byteweise in umgekehrter Reihenfolge verarbeitet. Der Wert 00 00 00 01 wird also als 00 01 00 00 gespeichert. Diese Form der Darstellung wird auch **LittleEndian** genannt. Wenn die Daten in der gewohnten Reihenfolge gespeichert werden, spricht man von **BigEndian**.

Allerdings hat die Flexibilität der Klassen `BinaryWriter` und `BinaryReader` auch ihren Preis. Denn beide Klassen sind nicht ganz so komfortabel wie die anderen Klassen zum Lesen und Schreiben von Daten, die Sie bisher kennengelernt haben. Sie müssen zum einen zunächst immer selbst einen Datei-Stream über die Klasse `FileStream` erzeugen, den Sie dann an die Klasse `BinaryReader` beziehungsweise `BinaryWriter` übergeben. Zum anderen sind Sie beim Einlesen selbst für die Interpretation der Daten verantwortlich. Was das genau bedeutet und welche Tücken dort drohen, erfahren Sie gleich.

Jetzt wollen wir uns aber erst einmal die Klasse `FileStream` ansehen.

3.2.1 Die Klasse FileStream

Über die Klasse `FileStream` können Sie beliebige neue Dateien anlegen, vorhandene Dateien öffnen sowie Daten schreiben und lesen.

Das Anlegen und Öffnen von Dateien erfolgt durch den Konstruktor der Klasse. Dieser Konstruktor erwartet den Namen der Datei als `string` und den Modus, in dem die Datei geöffnet werden soll. Für den Modus können Sie dabei folgende konstante Werte benutzen:

Tab. 3.1: Modi für das Öffnen von Dateien

Modus	Wirkung
<code> FileMode.Append^{a)}</code>	Die Datei wird geöffnet. Danach wird automatisch zum Dateiende gesprungen. Daten, die neu in die Datei geschrieben werden, werden also an eventuell bereits vorhandene Inhalte angehängt. Wenn die Datei nicht vorhanden ist, wird sie neu erzeugt.
<code> FileMode.Create</code>	Es wird eine Datei mit dem angegebenen Namen erzeugt. Wenn diese Datei bereits vorhanden ist, wird sie geöffnet. Wenn neue Daten in die Datei geschrieben werden, gehen eventuell bereits vorhandene Inhalte vollständig verloren.
<code> FileMode.CreateNew</code>	Es wird in jedem Fall eine neue Datei mit dem angegebenen Namen erzeugt. Wenn diese Datei bereits vorhanden ist, wird eine Ausnahme ausgelöst.
<code> FileMode.Open</code>	Die Datei wird geöffnet. Wenn sie nicht vorhanden ist, wird eine Ausnahme ausgelöst. Ob die Datei zum Lesen oder Schreiben geöffnet wird, hängt vom Zugriffsmodus ab (siehe Tab. 3.2).
<code> FileMode.OpenOrCreate</code>	Wenn die Datei vorhanden ist, wird sie geöffnet. Andernfalls wird die Datei mit dem angegebenen Namen erzeugt.
<code> FileMode.Truncate^{b)}</code>	Die Datei wird zum Schreiben von Daten geöffnet. Dabei werden alle eventuell bereits vorhandenen Inhalte gelöscht.

a) *Append* bedeutet übersetzt so viel wie „anhängen“.

b) *Truncate* bedeutet übersetzt so viel wie „kürzen, stutzen“.

Praktische Bedeutung haben vor allem die drei Modi `FileMode.Open` zum Öffnen einer Datei, `FileMode.Create` zum Anlegen einer Datei und `FileMode.Append` zum Anhängen von Daten an eine Datei.

Ob die Datei zum Lesen oder Schreiben geöffnet wird, können Sie über den **Zugriffsmodus** festlegen. Er wird ebenfalls an den Konstruktor der Klasse `FileStream` übergeben und kann folgende konstante Werte haben:

Tab. 3.2: Zugriffsmodi für Dateien

Modus	Wirkung
<code> FileAccess.Read</code>	Die Datei wird zum Lesen geöffnet.
<code> FileAccess.ReadWrite</code>	Die Datei wird zum Lesen und Schreiben geöffnet.
<code> FileAccess.Write</code>	Die Datei wird zum Schreiben geöffnet.

Neben dem Zugriffsmodus können Sie über den Konstruktor auch noch den sogenannten **Share-Modus**⁷ angeben. Er legt fest, ob andere Anwendungen oder Anwender auf die Datei zugreifen dürfen, solange sie geöffnet ist. Die wichtigsten Werte für den Share-Modus finden Sie in der Tab. 3.3:

Tab. 3.3: Share-Modi für Dateien

Modus	Wirkung
FileShare.None	Die gemeinsame Nutzung wird ausgeschlossen. Die Datei steht erst dann wieder zur Verfügung, wenn sie geschlossen wurde.
FileShare.Read	Die Datei kann von anderen Anwendungen oder Anwendern gelesen werden. Schreibzugriffe sind nicht möglich.
FileShare.ReadWrite	Andere Anwendungen oder Anwender können sowohl in die Datei schreiben als auch aus ihr lesen.
FileShare.Write	Andere Anwender oder Anwendungen können in die Datei schreiben, aber nicht aus ihr lesen.

Die folgende Anweisung würde zum Beispiel eine Variable `fStream` für die Klasse `FileStream` vereinbaren und dann über den Konstruktor eine Datei mit dem Namen **test.dat** im Ordner C:\test anlegen beziehungsweise öffnen. Die Daten in der Datei können dabei nur geschrieben werden.

```
FileStream fStream = new
FileStream("C:\\test\\test.dat", FileMode.Create,
 FileAccess.Write);
```

Um die Datei **test.dat** zum Lesen zu öffnen und gleichzeitig anderen Anwendern oder Anwendungen vollen Zugriff auf die Datei zu geben, verwenden Sie die folgende Anweisung:

```
FileStream fStream = new
FileStream("C:\\test\\test.dat", FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);
```

Bitte beachten Sie:

Sie sollten eine Datei, die Sie über die Klasse `FileStream` geöffnet haben, auch selbst wieder schließen. Dazu verwenden Sie die Methode `Close()`.

Allerdings empfiehlt Microsoft auch für die Klasse `FileStream`, die Ressourcen direkt nach Gebrauch freizugeben. Wir verwenden dafür wieder eine `using`-Konstruktion. Dann ist das Schließen über die Methode `Close()` nicht erforderlich.



7. *Share* bedeutet übersetzt so viel wie „Anteil“ oder „teilen“.

3.2.2 Die Klassen BinaryWriter und BinaryReader

Das Schreiben und Lesen von Daten in eine Datei, die Sie über die Klasse `FileStream` erzeugt haben, erfolgt am einfachsten über die Klassen `BinaryWriter` und `BinaryReader`.



Bitte beachten Sie:

Auch für die Klassen `BinaryWriter` und `BinaryReader` empfiehlt Microsoft die unmittelbare Freigabe der Ressourcen nach Gebrauch. Wir benutzen hier wieder die Variante mit dem `using`-Block. Ohne Freigabe der Ressourcen müssen Sie die Datei, die Sie mit den Klassen verarbeiten, selbst über die Methode `Close()` schließen.

Zum Schreiben verwenden Sie dabei die Methode `Write()` der Klasse `BinaryWriter`. Der folgende Code fragt zum Beispiel den Namen einer Datei über einen **Speichern unter**-Dialog ab und schreibt dann die Zahlen 0 bis 19 im binären Format in eine Datei.

```
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
{
    //eine neue Instanz von FileStream erzeugen
    //die Datei soll entweder geöffnet oder neu erzeugt werden
    //bitte in einer Zeile eingeben
    using (FileStream fStream = new
        FileStream(saveFileDialog1.FileName, FileMode.OpenOrCreate))
    {
        //eine neue Instanz von BinaryWriter auf der Basis
        //von fStream erzeugen
        //das Erzeugen erfolgt in einem eigenen using-Block
        //bitte in einer Zeile eingeben
        using (BinaryWriter binaerDatei = new
            BinaryWriter(fStream))
        {
            //die Zahlen von 0 bis 19 in die Datei schreiben
            for (int i = 0; i < 20; i++)
                binaerDatei.Write(i);
        }
    }
}
```

Code 3.3: Das Schreiben von Daten in binärer Form

Mit der Anweisung

```
using FileStream fStream = new
FileStream(saveFileDialog1.FileName,
FileMode.OpenOrCreate)
```

erzeugen wir eine Instanz `fStream` für die Klasse `FileStream`. Dabei soll die Datei, die im Dialog `saveFileDialog1` ausgewählt wurde, entweder geöffnet oder angelegt werden.

Anschließend erzeugen wir eine Instanz `binaerDatei` der Klasse `BinaryWriter`. Dabei übergeben wir die Instanz `fStream` über den Konstruktor. Das erledigt die Anweisung:

```
using BinaryWriter binaerDatei = new
    BinaryWriter(fStream)
```

Wir vereinbaren in dem Code zwei geschachtelte `using`-Blöcke. Der äußere Block gilt für die Instanz der Klasse `FileStream` und der innere für die Instanz der Klasse `BinaryWriter`.



Solch geschachtelte Konstruktionen können zu Problemen führen, da die Freigabe für den inneren Block unter Umständen mehrfach erfolgt. Um das zu vermeiden, müssten Sie sehr umständliche `try ... catch ... finally`-Konstruktionen erstellen. In unserem Fall spielt die drohende mehrfache Freigabe aber auch keine Rolle, da der mehrfache Aufruf intern abgefangen wird.

Danach schreiben wir mit der Schleife

```
for (int i = 0; i < 20; i++)
    binaerDatei.Write(i);
```

die Werte in die Datei. In welcher Form die Daten dabei gespeichert werden, ermittelt die Methode `Write()` automatisch anhand des Datentyps.

Beim Schreiben von binären Daten müssen Sie sich nicht weiter um das Format kümmern.



Bitte beachten Sie:

Sowohl die Klasse `FileStream` als auch die Klassen `BinaryWriter` und `BinaryReader` liegen im Namensraum `System.IO`. Denken Sie daher bitte vor dem Test des vorigen Codes daran, diesen Namensraum bekanntzumachen.

Das Lesen von Daten ist im Prinzip genauso einfach. Sie müssen allerdings peinlich genau darauf achten, dass Sie die Daten im richtigen Format interpretieren. Denn das binäre Muster `00 00 00 01` kann ja zum Beispiel für die Zahl 1 als `int`-Typ stehen, aber genauso gut für die beiden Zahlen 0 und 1 als `short`-Typen oder auch für eine Zeichenkette. Deshalb kennt die Klasse `BinaryReader` auch eine ganze Reihe von Methoden zum Lesen und Interpretieren der Daten. Einen ersten Überblick finden Sie in der Tab. 3.4:

Tab. 3.4: Die Methoden der Klasse `BinaryReader` zum Lesen und Interpretieren von Daten

Methode	Wirkung
<code>ReadBoolean()</code>	Es wird ein Wert vom Typ <code>Boolean</code> gelesen.
<code>ReadChar()</code>	Es wird ein Wert vom Typ <code>Char</code> gelesen – also ein einzelnes Zeichen.
<code>ReadDouble()</code>	Es wird ein Wert vom Typ <code>Double</code> gelesen.

Methode	Wirkung
ReadInt16()	Es wird ein Wert vom Typ <code>Int16</code> gelesen.
ReadInt32()	Es wird ein Wert vom Typ <code>Int32</code> gelesen.
ReadSingle()	Es wird ein Wert vom Typ <code>Single</code> gelesen.
ReadString()	Es wird eine Zeichenfolge gelesen. Die Länge wird dabei automatisch über das Präfix ermittelt, das zu Beginn der binären Folge steht. Die Länge des Präfixes hängt dabei von der Länge der Zeichenkette ab.



Ein Präfix ist – allgemein ausgedrückt – ein Teil einer Information, der selbst eine Bedeutung hat. Es steht vor der eigentlichen Information.

Hinweis:

Der zweite Teil des Namens der Methoden wird vom .NET Framework-Typ bestimmt. Es gibt daher zum Beispiel keine Methode `ReadShort()`, sondern nur eine Methode `ReadInt16()`.

Die Tabelle enthält nur die Methoden für die wichtigsten Datentypen. Auch für die anderen Datentypen aus dem .NET Framework gibt es entsprechende Methoden. Eine vollständige Liste finden Sie in der Hilfe der Klasse `BinaryReader`.

Das Lesen der Daten im Format `int` könnte dann zum Beispiel so aussehen:

```
//eine neue Instanz von FileStream erzeugen
//die Datei soll nur geöffnet erzeugt werden
//bitte in einer Zeile eingeben
using (FileStream fStream = new
FileStream(openFileDialog1.FileName, FileMode.Open))
{
    //eine neue Instanz von BinaryReader auf der Basis
    //von fStream erzeugen
    using (BinaryReader binaerDatei = new BinaryReader(fStream))
    {
        //die Zahlen auslesen und in das Listenfeld schreiben
        for (Int32 i = 0; i < 20; i++)
            listBox1.Items.Add(binaerDatei.ReadInt32());
    }
}
```

Code 3.4: Das Lesen von Daten im Format int

Mit der Anweisung

```
using FileStream fStream = new
FileStream(openFileDialog1.FileName, FileMode.Open)
```

erzeugen wir einen Datei-Stream für die Datei, die im Dialog `openFileDialog1` ausgewählt wurde. Die Datei soll dabei nur geöffnet werden.

Danach erzeugen wir in einem weiteren `using`-Block eine Instanz der Klasse `BinaryReader` und übergeben dabei die Instanz `fStream` an den Konstruktor.

In der Schleife

```
for (Int32 i = 0; i < 20; i++)
    listBox1.Items.Add(binaerDatei.ReadInt32());
```

lesen wir dann die Werte über die Methode `ReadInt32()` ein und schreiben sie in ein Listenfeld `listBox1`.

Da wir die Daten über die Methode `ReadInt32()` wieder in derselben Form zurücklesen, in der sie gespeichert wurden, sieht das Ergebnis im Listenfeld auch exakt so aus wie gewünscht.

Wenn Sie Daten dagegen über die Methode `ReadInt16()` zurücklesen, werden die Daten falsch interpretiert. In der Liste stehen dann zahlreiche Nullen, außerdem endet die Anzeige kurz nach dem Wert 9. Denn beim Lesen über die Methode `ReadInt16()` werden ja nur jeweils zwei Bytes gelesen – und damit auch nur die Hälfte der Datei verarbeitet.

Noch unangenehmer kann es werden, wenn Sie in unserem Beispiel für das Einlesen der Daten die Methode `ReadDouble()` benutzen. Denn hier werden jedes Mal acht Bytes gelesen – und damit lesen Sie über das Ende der Datei hinaus. Die Folge: Das Programm löst eine Ausnahme aus und stellt die Arbeit ein.

Achten Sie beim Verarbeiten von binären Daten in einer Datei deshalb sehr sorgfältig darauf, dass Sie die Daten beim Lesen korrekt interpretieren. Andernfalls erscheint nur Datenmüll und Sie riskieren den Absturz des Programms, wenn Sie über das Ende der Datei hinauslesen.

Hinweis:

Wie Sie das Ende einer Datei über die Klasse `BinaryReader` ermitteln, erfahren Sie im weiteren Verlauf des Studienhefts.

Die unterschiedlichen Auswirkungen der Methoden zum Lesen können Sie selbst im Projekt **BinaryDemo** ausprobieren.



Sie finden das Projekt im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Exkurs: Werte vieler gleicher Steuerelemente verarbeiten

Wenn Sie in einem Formular viele gleiche Steuerelemente verarbeiten wollen, kann es recht mühselig werden, die einzelnen Steuerelemente immer über den vollständigen Namen ansprechen zu müssen. Nehmen wir einmal ein Beispiel:

Sie haben ein Formular mit insgesamt sieben `TextBoxen` erstellt und wollen die Werte aus diesen Steuerelementen jetzt in eine Datei schreiben beziehungsweise wieder aus einer Datei lesen. Wenn Sie die Steuerelemente direkt über den Namen ansprechen, müssen Sie jetzt auch siebenmal die Methoden `Write()` beziehungsweise `ReadString()` verwenden.

Das Ganze lässt sich aber durch einen „Trick“ erheblich vereinfachen. Denn die Steuerelemente in einem Formular werden über eine Liste verwaltet, die Sie über die Eigenschaft `Controls` des Formulars ansprechen können. Sie können diese Liste durchgehen und überprüfen, ob es sich bei dem aktuellen Element um den Typ handelt, den Sie verarbeiten wollen. Dazu können Sie die folgende Konstruktion verwenden:

```
//eine Schleife über alle Steuerelemente im aktuellen
//Formular
foreach (Control element in this.Controls)
{
    //ist element eine TextBox?
    if (element.GetType() == typeof(TextBox))
        //dann den Namen ausgeben
        MessageBox.Show(element.Name);
}
```

Code 3.5: Zugriff auf alle Textboxen in einem Formular

In der Schleife wird zunächst eine Variable `element` vom allgemeinen Typ `Control` für ein Steuerelement vereinbart. In der `if`-Anweisung

```
if (element.GetType() == typeof(TextBox))
```

wird dann überprüft, ob der Typ dieses Elements mit dem Typ `TextBox` übereinstimmt. Wenn das der Fall ist, wird der Name über die Variable `element` ausgegeben.

Da die Schleife über den Ausdruck `this.Controls` sämtliche Steuerelemente im Formular verarbeitet, werden die Namen aller Textboxen hintereinander angezeigt.



Bitte beachten Sie unbedingt:

Sie müssen den Vergleich zwischen den Typen durchführen. Daher ist der Operator `typeof` im zweiten Ausdruck zwingend erforderlich. Eine Konstruktion wie

```
if (element.GetType() == TextBox)
```

führt zu Fehlermeldungen, da der Operator `typeof` fehlt.

Hinweis:

Sie können auch den Operator `is` für die Prüfung verwenden. Der Ausdruck sieht dann so aus:

```
if (element is TextBox)
```

Das Speichern und Lesen der Werte aus allen Textboxen in einem Formular könnte dann so erfolgen:

```
private void ButtonSpeichern_Click(object sender,
EventArgs e)
{
    //die Datei über FileStream erzeugen
    //bitte in einer Zeile eingeben
    using (FileStream fStream = new
    FileStream("C:\\test\\bin2.dat", FileMode.Create))
```

```
{  
    //eine neue Instanz von BinaryWriter auf der Basis  
    //von fStream erzeugen  
    //bitte in einer Zeile eingeben  
    using (BinaryWriter binaerDatei = new  
        BinaryWriter(fStream))  
    {  
        //die Werte aus den TextBoxen speichern  
        foreach (Control element in this.Controls)  
        {  
            //ist element eine TextBox?  
            if (element.GetType() == typeof(TextBox))  
                //oder  
                //if (element is TextBox)  
                binaerDatei.Write(element.Text);  
        }  
    }  
}  
  
private void ButtonLaden_Click(object sender, EventArgs e)  
{  
    //existiert die Datei?  
    if (File.Exists("C:\\test\\bin2.dat"))  
    {  
        //die Datei über FileStream erzeugen  
        //bitte in einer Zeile eingeben  
        using (FileStream fStream = new  
            FileStream("C:\\test\\bin2.dat", FileMode.Open))  
        {  
            //eine neue Instanz von BinaryReader auf der  
            //Basis von fStream erzeugen  
            //bitte in einer Zeile eingeben  
            using (BinaryReader binaerDatei = new  
                BinaryReader(fStream))  
            {  
                //die Werte auslesen und in die TextBoxen schreiben  
                foreach (Control element in this.Controls)  
                {  
                    //ist element eine TextBox?  
                    if (element.GetType() == typeof(TextBox))  
                        element.Text = binaerDatei.ReadString();  
                }  
            }  
        }  
    }  
    else  
        //bitte in einer Zeile  
        MessageBox.Show("Die Datei bin2.dat ist nicht  
        vorhanden!");  
}
```

Code 3.6: Speichern und Lesen für sämtliche TextBoxen in einem Formular



Ein komplettes Beispiel, das diese Technik umsetzt, finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **BinaryDemo2**.

Zum Abschluss dieses Exkurses noch ein Hinweis:

Die Konstruktion

```
foreach (Control element in this.Controls)
```

verarbeitet die Steuerelemente in der Reihenfolge, in der sie in das Formular eingefügt wurden – allerdings in der falschen Richtung. Das Steuerelement, das zuletzt eingefügt wurde, wird zuerst verarbeitet, und das Steuerelement, das zuerst eingefügt wurde, zuletzt.

3.3 Die Bestenliste für das Pong-Spiel

Zum Abschluss dieses Kapitels wollen wir die Bestenliste für das Pong-Spiel so ändern, dass die Daten nicht mehr in einer XML-Datei gespeichert werden, sondern in einer Datei **score.dat**. Diese Datei soll sich im selben Ordner befinden, in dem auch die Anwendung läuft.

Am grundsätzlichen Vorgehen werden wir dabei möglichst wenig ändern. Das heißt, wir schreiben in jedem Fall immer zehn Einträge in die Datei und lesen auch zehn Einträge wieder zurück. Die Punkte speichern wir als Typ `int` und den Namen als Typ `string`. Die Punkte werden dabei zuerst gespeichert. Wir müssen also beim Einlesen erst die Punkte beschaffen und danach den Namen.

Hinweis:

Wenn Ihnen das grundsätzliche Vorgehen beim Speichern und Lesen der Bestenliste nicht mehr geläufig ist, lesen Sie bitte noch einmal im letzten Studienheft im Kapitel 3 nach.

Damit Sie die folgenden Anleitungen praktisch nachvollziehen können, legen Sie bitte eine Kopie des Pong-Spiels an. Erstellen Sie einen neuen Ordner und kopieren Sie zum Beispiel sämtliche Dateien des Pong-Spiels, das Sie im letzten Studienheft erstellt haben, in diesen neuen Ordner.

Da wir die einzelnen Klassen unseres Pong-Spiels sauber getrennt haben, müssen wir nur Änderungen in der Klasse `Score` vornehmen. Diese Klasse finden Sie in der Datei `Score.cs` im Projekt **Pong**.

Damit alles seine Ordnung hat, ändern wir zunächst einmal den Namen des Feldes `xmlDateiname` in `dateiname`.

Tipp:

Wenn Sie den Bezeichner ändern, erscheint links vor der Spalte eine kleine Glühbirne für die Livecodeanalyse. Über diese Glühbirne können Sie den Bezeichner im gesamten Projekt automatisch ändern lassen.

Ersetzen Sie anschließend die Anweisung

```
using System.Xml;
```

in der Datei **Score.cs** durch die Anweisung

```
using System.IO;
```

Ändern Sie dann im Konstruktor der Klasse **Score** die Anweisung

```
xmlDateiname = System.Windows.Forms.Application.StartupPath  
+ "\\score.xml";
```

in

```
dateiname = System.Windows.Forms.Application.StartupPath  
+ "\\score.dat";
```

und ersetzen Sie auch den Bezeichner **xmlDateiname** in der Abfrage, ob die Datei existiert, durch den Bezeichner **dateiname**.

Im nächsten Schritt kümmern wir uns jetzt erst einmal um das Speichern der Daten. Dazu erzeugen wir in der Methode **SchreibePunkte()** der Klasse **Score** Instanzen der Klassen **FileStream** und **BinaryWriter**. Über den Konstruktor der Klasse **FileStream** öffnen wir die Datei **score.dat** im Modus **Create**. Beim ersten Speichern der Bestenliste wird die Datei also neu angelegt, bei jedem weiteren Speichern werden die vorhandenen Daten überschrieben. Da wir die Bestenliste immer vollständig speichern, ist das aber durchaus gewünscht.

Anschließend schreiben wir dann in einer Schleife die Daten in die Datei. Dabei setzen wir die Techniken ein, die Sie bereits in den vorherigen Kapiteln kennengelernt haben.

Die neue Methode **SchreibePunkte()** für die Klasse **Score** sieht dann so aus:

```
void SchreibePunkte()  
{  
    //eine neue Instanz von FileStream erzeugen  
    //die Datei soll entweder geöffnet oder neu erzeugt  
    //werden  
    //bitte in einer Zeile eingeben  
    using (FileStream fStream = new FileStream(dateiname,  
        FileMode.Create))  
    {  
        //eine neue Instanz von BinaryWriter auf der Basis  
        //von fStream erzeugen  
        //bitte in einer Zeile eingeben  
        using (BinaryWriter binaerDatei = new  
            BinaryWriter(fStream))  
        {  
            //die Einträge in die Datei schreiben  
            for (int i = 0; i < anzahl; i++)  
            {  
                //die Punkte  
                binaerDatei.Write(bestenliste[i].GetPunkte());  
            }  
        }  
    }  
}
```

```
//und dann den Namen  
binaerDatei.Write(bestenliste[i].GetName());  
}  
}  
}  
}
```

Code 3.7: Das Schreiben der Bestenliste in eine binäre Datei

Das Lesen der Daten erfolgt mit ähnlichen Techniken. Wir laden die Daten in einer Schleife aus der Datei und weisen sie dem entsprechenden Element der Liste zu. Dabei müssen wir lediglich darauf achten, dass wir die Daten in der richtigen Reihenfolge einlesen – also zuerst die Punkte und dann den Namen.

Die Methode `LesePunkte()` für die Klasse `Score` sieht so aus:

```

void LesePunkte()
{
    //zum Zwischenspeichern der gelesenen Daten
    int tempPunkte;
    string tempName;
    //eine neue Instanz von FileStream erzeugen
    //die Datei soll geöffnet werden
    //bitte in einer Zeile eingeben
    using (FileStream fStream = new FileStream(dateiname,
        FileMode.Open))
    {
        //eine neue Instanz von BinaryReader auf der Basis
        //von fStream erzeugen
        //bitte in einer Zeile eingeben
        using (BinaryReader binaerDatei = new
            BinaryReader(fStream))
        {
            //die Einträge lesen und zuweisen
            for (int i = 0; i < anzahl; i++)
            {
                //die Punkte
                tempPunkte = binaerDatei.ReadInt32();
                //den Namen
                tempName = binaerDatei.ReadString();
                //und jetzt zuweisen
                bestenliste[i].SetzeEintrag(tempPunkte, tempName);
            }
        }
    }
}

```

Code 3.8: Das Lesen der Bestenliste aus der binären Datei

Das war bereits alles. Übernehmen Sie die Änderungen aus den beiden vorigen Codes und testen Sie das Programm dann.

So viel an dieser Stelle zum Umgang mit Dateien. Im nächsten Kapitel erfahren Sie unter anderem, wie Sie in Dateien suchen und wie Sie Dateien löschen und umbenennen.

Zusammenfassung

Über die Klassen `StreamReader` und `StreamWriter` können Sie sehr einfach auf Textdateien zugreifen. Beide Klassen befinden sich im Namensraum `System.IO`.

In der Standardeinstellung verarbeiten beide Klassen Texte im Format UTF-8.

Das zeilenweise Lesen beziehungsweise Schreiben erfolgt mit den Methoden `WriteLine()` beziehungsweise `ReadLine()`.

Nach dem Zugriff auf die Daten sollten Sie die Datei in jedem Fall wieder schließen. Das ist vor allem beim Schreiben von Daten sehr wichtig. Das Schließen können Sie auch über eine `using`-Konstruktion automatisch durchführen lassen.

Mit den Klassen `FileStream`, `BinaryReader` und `BinaryWriter` können Sie auch beliebige Daten in binärer Form verarbeiten.

An den Konstruktor der Klasse `FileStream` übergeben Sie den Namen der Datei, die geöffnet werden soll, und den Modus, in dem die Datei geöffnet werden soll. Außerdem können Sie auch noch den Zugriffsmodus und den Share-Modus angeben.

Das Schreiben von Daten in eine binäre Datei erfolgt mit der Methode `Write()` von `BinaryWriter`. Um das Format der Daten müssen Sie sich dabei nicht weiter kümmern.

Beim Zurücklesen müssen Sie darauf achten, dass Sie die richtige Methode für den entsprechenden Datentyp verwenden. Andernfalls werden die Daten falsch interpretiert.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie wollen eine Textdatei `c:\test\versuch.txt` über eine Instanz `textDatei` der Klasse `StreamWriter` öffnen. Formulieren Sie bitte die entsprechende Anweisung.

- 3.2 Wie können Sie festlegen, dass der Inhalt einer Datei, die Sie über die Klasse `StreamWriter` öffnen, nicht überschrieben wird?

- 3.3 Mit welcher Eigenschaft können Sie ermitteln, ob Sie das Ende einer Datei erreicht haben, die Sie über die Klasse `StreamReader` geöffnet haben?

- 3.4 Was müssen Sie an den Konstruktor der Klassen `BinaryReader` und `BinaryWriter` übergeben?

- 3.5 Beschreiben Sie bitte kurz die Wirkung der folgenden Modi für das Öffnen von Dateien über die Klasse `FileStream`.
- a) `FileMode.Create`
 - b) `FileMode.Open`
 - c) `FileMode.Append`

- 3.6 Über welche Eigenschaft können Sie auf die Steuerelemente in einem Formular zugreifen? Formulieren Sie bitte auch eine Schleife, die sämtliche Steuerelemente in einem Formular durchläuft.

4 Weitere Datei- und Ordnerfunktionen

In diesem Kapitel stellen wir Ihnen noch einige weitere nützliche Funktionen zum Arbeiten mit Dateien und Ordnern vor.

Beginnen wir mit dem Suchen und Positionieren in Dateien.

4.1 Suchen und Positionieren in einer Datei

Bisher haben wir Daten immer der Reihe nach in eine Datei geschrieben und auch der Reihe nach wieder gelesen. Um den korrekten Zugriff auf die Daten mussten wir uns dabei eigentlich nicht weiter kümmern. Wir mussten den verschiedenen Methoden lediglich mitteilen, woher die Daten kommen beziehungsweise wo die Daten abgelegt werden sollen.

Neben diesem **sequenziellen Zugriff** können Sie aber auch auf eine beliebige Position innerhalb einer Datei zugreifen und dann von dort direkt Daten lesen. Das ist zum Beispiel dann sehr interessant, wenn Sie in einer Datei suchen möchten und dabei bestimmte Informationen „überlesen“ wollen.

Schauen wir uns dazu ein Beispiel an:

Nehmen wir einmal an, Sie wollen in der Bestenliste des Pong-Spiels nach einer bestimmten Punktzahl suchen und dann den dazugehörigen Spielernamen anzeigen lassen. Dazu könnten Sie jetzt mit den bekannten Techniken einen Eintrag nach dem anderen einlesen, die Punktzahl mit dem Suchkriterium vergleichen und dann gegebenenfalls den Spielernamen ausgeben.

Dabei verarbeiten Sie aber mehr Daten als nötig. Denn eigentlich müssen Sie den Namen des Spielers ja nur dann lesen, wenn die Punktzahl mit dem Suchkriterium übereinstimmt. Andernfalls könnten Sie das Einlesen des Spielernamens auch überspringen, da Sie die Daten sowieso nicht verarbeiten wollen.

Bevor wir diese Suche programmieren, wollen wir uns aber zunächst einmal kurz ansehen, wie der Zugriff auf die Daten in einer Datei intern erfolgt.

Dazu wird eine Art Zeiger verwendet, der die aktuelle Position innerhalb der Datei markiert. Direkt nach dem Öffnen einer Datei steht dieser Zeiger in der Regel ganz am Anfang der Datei. Bei jedem erfolgreichen Lese- oder Schreibzugriff wird er automatisch um die Anzahl der gelesenen beziehungsweise geschriebenen Bytes verschoben. Jeder weitere Lese- oder Schreibzugriff erfolgt dann an dieser neuen Position.

Über die Eigenschaft `Position` der Klasse `FileStream` können Sie die Position des internen Zeigers in der Datei auch selbst beliebig setzen. Dabei sind Sie allerdings auch selbst dafür verantwortlich, dass sich an der neuen Stelle brauchbare Daten befinden. Außerdem müssen Sie sorgfältig darauf achten, dass Sie den internen Zeiger nicht über das Ende der Datei hinaus verschieben. Dazu können Sie die Eigenschaft `Length` der Klasse `FileStream` verwenden. Sie liefert Ihnen die Größe der Datei in Bytes.

Im folgenden Fragment wird zum Beispiel der interne Zeiger auf den Wert einer Variablen `neuePosition` gesetzt. Vor dem Positionieren wird überprüft, ob sich die neue Position überhaupt noch innerhalb der Datei befindet.

```
//bitte in einer Zeile eingeben
using (FileStream fStream = new
FileStream(Application.StartupPath + "\\\\demo.dat",
 FileMode.Open))
{
    using (BinaryReader binaerDatei = new BinaryReader(fStream) )
    {
        //ist die Position gültig?
        if (neuePosition <= fStream.Length - 4)
        {
            //positionieren
            fStream.Position = neuePosition;
            //dann lesen und anzeigen
            //bitte in einer Zeile eingeben
            labelWert.Text =
            Convert.ToString(binaerDatei.ReadInt32());
        }
    }
}
```

Code 4.1: Direktes Positionieren in einer Datei (es handelt sich um ein Fragment)



Bitte beachten Sie unbedingt:

Sie müssen die Abfrage auf das Ende der Datei beim direkten Zugriff äußerst sorgfältig formulieren. Andernfalls lesen Sie sehr schnell über das Ende hinaus.

Ein Beispiel:

Nehmen wir einmal an, die Datei, die im vorigen Code verarbeitet wird, ist 120 Bytes groß. Wenn Sie jetzt den internen Zeiger auf die Position 120 setzen und danach versuchen, Daten in der Größe eines `int`-Typs zu lesen, wird eine Ausnahme ausgelöst. Denn nach der Position 120 gibt es ja keine Daten mehr in der Datei.

Auch die Positionen 117, 118 und 119 sind nicht gültig, wenn Sie nach dem Positionieren einen `int`-Typ über `ReadInt32()` lesen möchten. Denn der Datentyp `int` benötigt vier Bytes. Sie würden also ebenfalls über das Ende der Datei hinaus lesen.

Daher erfolgt die Abfrage auf eine gültige Position im vorigen Code auf die Größe der Datei abzüglich der Größe des Datentyps `int`. Die letzte gültige Position für das Einlesen eines `int`-Typs bei einer Dateigröße von 120 Bytes wäre dann 116 (120 Bytes – 4 Bytes für die Größe des Typs).

Das Positionieren des internen Zeigers auf einen ungültigen Wert führt nicht direkt zu einem Fehler. Erst Schreib- oder Lesezugriffe an der ungültigen Position lösen eine Ausnahme aus.

Ein komplettes Beispiel mit dem zufälligen Positionieren in einer Datei finden Sie im Projekt **BinaryDemo3** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

So viel zum allgemeinen Vorgehen. Setzen wir jetzt die Suche in der Bestenliste praktisch um. Ein Problem ist dabei die Überprüfung auf das Dateiende. Denn wir wissen vor dem Lesen des letzten Eintrags in der Liste gar nicht, wie viele Bytes überhaupt noch

folgen. Der Name kann ja eine beliebige Länge haben. Trotzdem fragen wir einfach ab, ob die aktuelle Position in der Datei kleiner ist als die Größe der Datei. Das ist ohne Weiteres möglich, da wir den Aufbau der Datei genau kennen – also durch die Lese- beziehungsweise Positionieroperationen exakt am Dateiende landen.

Außerdem müssen wir herausfinden, wie lang die Zeichenketten für den Namen sind. Das ist aber ebenfalls nicht weiter schwierig, da die Länge in dem Byte unmittelbar vor dem eigentlichen Namen gespeichert wird.

Legen Sie jetzt bitte eine neue leere Windows Forms-Anwendung an. Fügen Sie in das Formular eine **TextBox** zur Eingabe des Suchkriteriums, ein Listenfeld zur Anzeige der Namen und eine Schaltfläche zum Starten der Suche ein. Das Listenfeld benutzen wir, damit wir alle Treffer zum Suchkriterium anzeigen können.

Speichern Sie dann das gesamte Projekt und kopieren Sie anschließend die Datei **score.dat** aus dem Ordner des Pong-Spiels in den Ordner der neuen Anwendung.

Bitte beachten Sie:

Sie müssen die Datei **score.dat** in den Ordner kopieren, in dem Visual Studio die ausführbare Datei ablegt – also zum Beispiel in den Unterordner **\bin\Release** des Projektordners. Wenn Ihnen das vorgestellte Verfahren zu umständlich ist, können Sie natürlich auch den Pfad zur Datei in den Codes anpassen.



Die Anweisungen für das Anklicken der Schaltfläche zum Suchen könnten dann so aussehen:

```
private void ButtonSuche_Click(object sender, EventArgs e)
{
    string dateiname;
    int treffer = 0;
    int punkte = 0;
    int neuePosition = 0;
    string name;
    //die Liste leeren
    listBox1.Items.Clear();
    //ist das Suchkriterium leer?
    if (textBox1.Text == String.Empty)
    {
        MessageBox.Show("Bitte geben Sie ein Suchkriterium ein.");
        //den Focus auf das Eingabefeld setzen
        textBox1.Select();
        return;
    }
    //den Namen für die Datei zusammenbauen
    dateiname = Application.StartupPath + "\\score.dat";
    //existiert die Datei?
    if (File.Exists(dateiname) == false)
    {
        MessageBox.Show("Die Datei score.dat ist nicht vorhanden!");
        return;
    }
    else
```

```

{
    //die Datei öffnen
    //bitte in einer Zeile eingeben
    using (FileStream fStream = new FileStream(dateiname,
        FileMode.Open))
    {
        //bitte in einer Zeile eingeben
        using (BinaryReader binaerDatei = new
        BinaryReader(fStream))
        {
            //die gesamte Datei durchsuchen
            while (fStream.Position < fStream.Length)
            {
                //die erste Punktzahl lesen
                punkte = binaerDatei.ReadInt32();
                //stimmt die Punktzahl mit dem Suchkriterium überein?
                if (punkte == Convert.ToInt32(textBox1.Text))
                {
                    //den Wert für Treffer erhöhen
                    treffer++;
                    //den Namen lesen
                    name = binaerDatei.ReadString();
                    //und in der Liste ausgeben
                    listBox1.Items.Add(name);
                }
                //stimmt das Suchkriterium nicht, dann den Namen
                //überlesen
                else
                {
                    //die Länge beschaffen
                    neuePosition = binaerDatei.ReadByte();
                    //und die neue Position ansteuern
                    fStream.Position = fStream.Position + neuePosition;
                }
            }
        }
    }
    //wenn kein Treffer, eine Meldung in der Liste
    //erzeugen
    if (treffer == 0)
        listBox1.Items.Add("Kein Treffer gefunden");
}
}

```

Code 4.2: Das Suchen in der Bestenliste

Hinweis:

Der genaue Name der Methode hängt vom Namen der Schaltfläche ab.

Zunächst einmal vereinbaren wir einige Variablen für den Dateinamen, die Treffer der Suche, die zu lesenden Daten und die neue Position.

Danach löschen wir den Inhalt des Listenfeldes und sorgen so dafür, dass nur die Treffer der aktuellen Suche angezeigt werden.

Mit den beiden folgenden `if`-Anweisungen wird überprüft, ob das Feld für das Suchkriterium leer ist und ob die Datei überhaupt vorhanden ist.

Wenn das der Fall ist, wird die Datei geöffnet und zuerst die Punktzahl eingelesen. Falls das Suchkriterium und die eingelesene Punktzahl übereinstimmen, wird der Name aus der Datei beschafft und im Listenfeld ausgegeben. Dabei gibt es keine Besonderheiten.

Stimmen das Suchkriterium und die eingelesene Punktzahl nicht überein, wird der interne Zeiger in der Datei um die Länge des Namens verschoben. Dazu beschaffen wir uns erst über die Methode `ReadByte()` die Länge des Namens aus der Datei und setzen dann die Position des internen Zeigers in der Datei neu.

Diese Anweisungen werden in der Schleife so lange wiederholt, bis die aktuelle Position in der Datei größer oder gleich der Dateigröße ist.

Bitte beachten Sie:



Unsere Technik zum Überspringen der Zeichenketten funktioniert nur, wenn die Zeichenketten nicht länger sind als 255 Zeichen. Sonst wird nämlich die Länge auf mehrere Bytes verteilt, die vor der Zeichenkette gespeichert werden. Da unser Programm aber nur ein Byte für die Länge liest, kommt es in diesem Fall komplett durcheinander.

Abschließend wird noch überprüft, ob `treffer` den Wert 0 hat – also die Suche kein Ergebnis brachte. Dann wird im Listenfeld ein entsprechender Hinweis eingefügt.

Das komplette Projekt finden unter dem Namen **BinaryDemo4** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.



So viel zum Suchen und Positionieren in einer Datei. Im nächsten Abschnitt werden wir uns einige Methoden zur Verwaltung von Dateien und Ordnern ansehen.

Hinweis:

Neben dem Setzen der Position über die Eigenschaft `Position` können Sie auch die Methode `Seek()`⁸ von `FileStream` verwenden. Über diese Methode können Sie unter anderem auch die Position vom Dateiende aus setzen. Details zum Einsatz der Methode finden Sie in der Hilfe des .NET Frameworks unter dem Stichwort `FileStream.Seek`.

8. `Seek` bedeutet übersetzt so viel wie „suchen“ oder „streben nach“.

4.2 Datei- und Ordnerverwaltung

Bisher haben wir vor allem mit dem Inhalt einer Datei gearbeitet. Das .NET Framework kennt aber auch zahlreiche Methoden, um eine Datei insgesamt zu bearbeiten – zum Beispiel zu löschen oder umzubenennen.

Einige dieser Methoden mit einer kurzen Beschreibung finden Sie in der Tab. 4.1:

Tab. 4.1: Methoden zur Verwaltung von Dateien

Methode	Wirkung
<code>File.Copy()</code>	Mit der Methode <code>Copy()</code> können Sie eine Datei kopieren. Die Methode erwartet als erstes Argument den Namen der Datei, die kopiert werden soll, und als zweites Argument den Zielordner sowie den neuen Namen. Über ein drittes Argument vom Typ <code>bool</code> können Sie festlegen, ob die Datei gegebenenfalls überschrieben werden soll.
<code>File.Delete()</code>	Mit der Methode <code>Delete()</code> löschen Sie eine Datei. Der Name der Datei wird dabei als Argument vom Typ <code>string</code> übergeben.
<code>File.Exists()</code>	Mit der Methode <code>Exists()</code> können Sie überprüfen, ob eine Datei vorhanden ist. Dazu übergeben Sie den Namen der Datei als Argument vom Typ <code>string</code> . Wenn die Datei gefunden wird, liefert <code>Exists()</code> den Wert <code>true</code> , sonst <code>false</code> .
<code>File.Move()</code>	Mit der Methode <code>Move()</code> verschieben Sie eine Datei. Die Methode erwartet als erstes Argument den Namen der Datei, die verschoben werden soll, und als zweites Argument den Zielordner sowie den neuen Namen. Über ein drittes Argument vom Typ <code>bool</code> können Sie festlegen, ob die Datei gegebenenfalls überschrieben werden soll.
<code>FileInfo.DirectoryName()</code>	Die Methode <code>DirectoryName()</code> liefert Ihnen den Pfad einer Datei als Typ <code>string</code> . Geliefert wird nur der eigentliche Pfad ohne den Namen der Datei.
<code>FileInfo.Extension()</code>	Die Methode <code>Extension()</code> liefert Ihnen die Erweiterung einer Datei einschließlich des führenden Punktes als Typ <code>string</code> .
<code>FileInfo.FullName()</code>	Die Methode <code>FullName()</code> liefert Ihnen den vollständigen Pfad einer Datei einschließlich des Namens.
<code>Path.ChangeExtension()</code>	Über die Methode <code>ChangeExtension()</code> können Sie die Erweiterung einer Datei verändern. Dazu übergeben Sie den vollständigen Pfad der Datei und die neue Erweiterung jeweils als Argumente vom Typ <code>string</code> . Bitte beachten Sie, dass die Methode lediglich den neuen Namen liefert, aber die Datei selbst nicht umbenennt.

Hinweis:

C# kennt keine Methode für das Umbenennen einer Datei. Sie können hier aber die Methode `File.Move()` verwenden und die Datei quasi in denselben Ordner mit einem anderen Namen verschieben.

Für die Verwaltung von Ordnern können Sie unter anderem folgende Methoden des .NET Frameworks verwenden:

Tab. 4.2: Methoden für die Verwaltung von Ordnern

Methode	Wirkung
<code>Directory.CreateDirectory()</code>	Mit der Methode <code>CreateDirectory()</code> legen Sie einen neuen Ordner an. Der Name des Ordners wird dabei als Argument vom Typ <code>string</code> übergeben.
<code>Directory.Delete()</code>	Die Methode <code>Delete()</code> löscht einen Ordner. Der Name des Ordners wird dabei als Argument vom Typ <code>string</code> übergeben. Über ein zweites Argument vom Typ <code>bool</code> können Sie festlegen, ob der Ordner auch dann gelöscht werden soll, wenn er nicht leer ist.
<code>Directory.Exists()</code>	Mit der Methode <code>Exists()</code> können Sie überprüfen, ob ein Ordner vorhanden ist. Dazu übergeben Sie den Namen des Ordners als Argument vom Typ <code>string</code> . Wenn der Ordner gefunden wird, liefert <code>Exists()</code> den Wert <code>true</code> zurück, sonst <code>false</code> .
<code>Directory.GetCurrentDirectory()</code>	Die Methode <code>GetCurrentDirectory()</code> liefert Ihnen den aktuellen Ordner als Typ <code>string</code> .
<code>Directory.GetDirectories()</code>	Die Methode <code>GetDirectories()</code> liefert Ihnen alle Unterordner in einem Ordner als Array vom Typ <code>string</code> .
<code>Directory.GetFiles()</code>	Die Methode <code>GetFiles()</code> liefert Ihnen alle Dateien in einem Ordner als Array vom Typ <code>string</code> .
<code>Directory.Move()</code>	Die Methode <code>Move()</code> verschiebt einen Ordner. Der Name des Ordners wird dabei als Argument vom Typ <code>string</code> übergeben.
<code>Directory.SetCurrentDirectory()</code>	Die Methode <code>SetCurrentDirectory()</code> setzt den aktuellen Ordner. Der Name des Ordners wird dabei als Argument vom Typ <code>string</code> angegeben.

Auch für die Arbeit mit Laufwerken kennt das .NET Framework einige Methoden und Eigenschaften:

Tab. 4.3: Methoden und Eigenschaften für die Arbeit mit Laufwerken

Methode/Eigenschaft	Wirkung
DriveInfo.AvailableFreeSpace	Die Eigenschaft AvailableFreeSpace liefert Ihnen den freien Platz auf einem Laufwerk in Bytes.
DriveInfo.DriveType	Die Eigenschaft DriveType liefert Ihnen Informationen zum Typ des Laufwerks.
DriveInfo.GetDrives()	Die Methode GetDrives() liefert Ihnen die Namen aller logischen Laufwerke auf einem Computer als Array vom Typ DriveInfo.
DriveInfo.Totalsize	Die Eigenschaft Totalsize liefert Ihnen die Größe eines Laufwerks in Bytes.

Hinweis:

Für den Einsatz der Eigenschaften und Methoden der Klassen `DriveInfo` und `FileInfo` müssen Sie zuerst eine Instanz der Klasse erzeugen.

Nahezu alle Methoden beziehungsweise Klassen, die wir Ihnen in den vorigen Tabellen vorgestellt haben, liegen im Namensraum `System.IO`.

Einige dieser Methoden finden Sie im Projekt **Verwaltung** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im praktischen Einsatz. Bitte beachten Sie aber, dass das Programm recht einfach gehalten ist und über keinerlei Sicherheitsfunktionen verfügt. So können Sie zum Beispiel Dateien, die Sie mit der Anwendung löschen, nicht wiederherstellen. Testen Sie das Programm daher nur mit Daten, die Sie nicht mehr benötigen.

Neben diesen Methoden und Eigenschaften gibt es noch zahlreiche weitere Methoden und Eigenschaften für den Zugriff auf Dateien, Ordner und Laufwerke, die wir Ihnen hier nicht einzeln vorstellen wollen.

So viel zur Datei- und Ordnerverwaltung. Kommen wir abschließend zum Drucken.

4.3 Drucken

Das Drucken im .NET Framework erfolgt über das manuelle Erzeugen eines Druckauftrags. Dabei können Sie auf der Zeichenfläche des Druckers Texte und auch Grafiken ausgeben.

Das hört sich beim ersten Lesen vielleicht ein wenig kompliziert an, ist aber eigentlich recht einfach – zumindest, wenn Sie nur einen kurzen Text oder eine Grafik ausgeben möchten.

Im ersten Schritt fügen Sie ein Steuerelement `PrintDocument` in Ihre Anwendung ein. Sie finden dieses Steuerelement in der Gruppe **Drucken** der Toolbox.

Tipp:

Über die Eigenschaft **DocumentName** des Steuerelements **PrintDocument** legen Sie einen Namen für das Ausgabedokument fest. Dieser Name wird zum Beispiel in der Druckerwarteschlange angezeigt und in den Anzeigen zum Druckfortschritt.

Im Ereignis **PrintPage** des Steuerelements geben Sie anschließend die Anweisungen ein, die beim Drucken jeder Seite ausgeführt werden sollen. Dazu können Sie nahezu alle Anweisungen verwenden, die Ihnen auch für das Zeichnen auf einer normalen Zeichenfläche zur Verfügung stehen.

Die Druckausgabe einer Grafik aus einer **PictureBox** und eines Textes aus einer **TextBox** könnte dann zum Beispiel so aussehen:

```
private void PrintDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    //ein Pinsel für die Farbe
    SolidBrush pinsel = new SolidBrush(Color.Black);
    //für die Schriftart
    Font schrift = new Font("Arial", 12, FontStyle.Bold);
    //bitte in einer Zeile eingeben
    //den Text auf der Zeichenfläche des Druckers ausgeben
    e.Graphics.DrawString(textBox1.Text, schrift,
    pinsel, 20, 20);
    //eine neue Bitmap-Grafik aus der Bitmap in der
    //PictureBox erzeugen
    e.Graphics.DrawImage(pictureBox1.Image, new Point(20, 200));
}
```

Code 4.3: Die Ausgabe einer Grafik und eines Textes auf dem Drucker

Zuerst wird ein Pinsel für die Textfarbe vereinbart und die Schriftart festgelegt. Das kennen Sie ja bereits vom Zeichnen in einem Formular.

Danach wird über die Methode `DrawString()` der Text in einer Textbox auf der Zeichenfläche des Druckers ausgegeben. Diese Zeichenfläche erhalten Sie wie gewohnt durch den Ausdruck `e.Graphics`.

Anschließend wird mit der Methode `DrawImage()` das Bild aus der **PictureBox** gedruckt. Als erstes Argument erwartet die Methode das Bild, das ausgegeben werden soll. Danach folgen die Koordinaten für die Druckausgabe vom Typ `Point`.

Um die Druckausgabe zu starten, rufen Sie die Methode `Print()` der Instanz für das Steuerelement **PrintDocument** auf – zum Beispiel beim Anklicken einer Schaltfläche.

Bitte beachten Sie:

Die Methode `Print()` startet nur die Druckausgabe. Was gedruckt wird, legen Sie im Ereignis **PrintPage** fest.



Auch eine Druckvorschau lässt sich recht einfach erzeugen. Dazu fügen Sie das Steuerelement **PrintPreviewDialog** aus der Gruppe **Drucken** der Toolbox ein und setzen im Quelltext die Eigenschaft **Document** für das Steuerelement auf das Dokument, das an-

gezeigt werden soll. Das ist in der Regel der Name des Steuerelements **PrintDocument**. Anschließend lassen Sie dann den Dialog mit `ShowDialog()` anzeigen. Die entsprechenden Anweisungen zum Erzeugen einer Druckvorschau könnten so aussehen:

```
private void ButtonVorschau_Click(object sender,
EventArgs e)
{
    //das anzuzeigende Dokument setzen
    printPreviewDialog1.Document = printDocument1;
    //und den Dialog anzeigen
    printPreviewDialog1.ShowDialog();
}
```

Code 4.4: Das Erzeugen einer Druckvorschau

Die Druckvorschau für das Textfeld und die Grafik würde dann so aussehen:

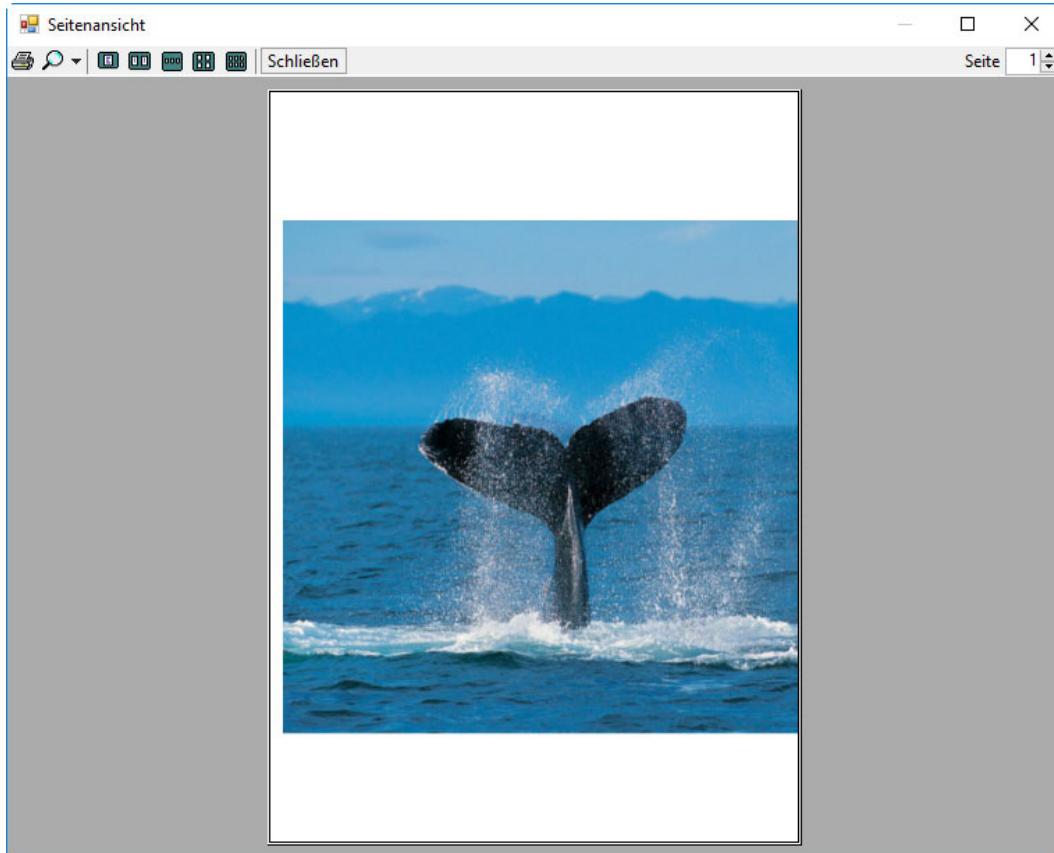


Abb. 4.1: Eine Druckvorschau



Bitte beachten Sie:

In der Regel ist die Auflösung eines Druckers sehr viel höher als die Auflösung eines Bildschirms. Die einzelnen Pixel bei einem Drucker liegen also sehr viel enger zusammen als bei einem Bildschirm. Dadurch kann die Ausgabe auf einem Drucker stark von der Ausgabe auf dem Bildschirm abweichen.

Tipp:

Für die Formatierung der Druckausgabe stehen Ihnen nahezu alle Möglichkeiten zur Verfügung, die Sie auch für normale Zeichnungen einsetzen können. So lassen sich zum Beispiel andere Farben wählen, eine andere Schriftart oder auch eine andere Schriftgröße.

Das .NET Framework kennt noch einige weitere Steuerelemente für das Drucken. So können Sie über das Steuerelement **PrintDialog** sehr einfach auf den Standarddialog **Drucken** von Windows zugreifen.

Ausführliche Beschreibungen der verschiedenen Steuerelemente finden Sie in der Hilfe im Dokument **Druckunterstützung in Windows Forms**.

Das Drucken eines Dokuments mit mehreren Seiten ist nicht ganz so einfach wie die Ausgabe einer einzigen Seite. Denn Sie müssen im Ereignis **PrintPage** selbst ausrechnen, ob die Ausgabe noch auf die aktuelle Seite passt. Dazu können Sie über die Eigenschaft `e.MarginBounds` auf die Ränder der Druckseite zugreifen und dann mitzählen, ob noch weiterer Text auf der aktuellen Seite dargestellt werden kann. Für eine Textdatei könnte das zum Beispiel so aussehen:

```
private void ButtonDrucken_Click(object sender, EventArgs e)
{
    //gibt es die Datei überhaupt?
    if (File.Exists("C:\\test\\druckdatei.txt") == false)
        //bitte in einer Zeile eingeben
        MessageBox.Show("Die Datei druckdatei.txt ist nicht vorhanden!");
    else
    {
        //eine neue Instanz von StreamReader erzeugen
        //textDatei ist als Feld der Klasse vereinbart
        try
        {
            //bitte in einer Zeile eingeben
            textDatei = new
            StreamReader("C:\\test\\druckdatei.txt");
            //die Druckausgabe starten
            printDocument1.Print();
        }
        catch (Exception)
        {
            MessageBox.Show("Es hat ein Problem gegeben.");
        }
        finally
        {
            //die Ressourcen freigeben
            //dabei wird die Datei auch geschlossen
            if (textDatei != null)
                textDatei.Dispose();
        }
    }
}
```

```

private void PrintDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    //ein Pinsel für die Farbe
    SolidBrush pinsel = new SolidBrush(Color.Black);
    //für die Schriftart
    Font schrift = new Font("Arial", 12, FontStyle.Bold);
    //für die Zeilen pro Seite
    int zeilenProSeite = 0;
    //für die Y-Position der Ausgabe
    int yPos = e.MarginBounds.Top;
    //zum Mitzählen der Zeilen
    int zeilenGedruckt = 0;
    //für die Textzeile
    string zeile;
    zeile = String.Empty;
    //die Höhe des Ausgabebereichs wird durch die Höhe
    //der Schrift geteilt
    zeilenProSeite = e.MarginBounds.Height / schrift.Height;
    //bitte in einer Zeile eingeben
    //solange noch mehr Zeilen da sind und der untere
    //Rand nicht erreicht ist, wird gedruckt
    while ((zeilenGedruckt < zeilenProSeite) &&
(textDatei.EndOfStream == false))
    {
        //die nächste Zeile lesen
        zeile = textDatei.ReadLine();
        //die Zeile ausgeben
        //bitte in einer Zeile eingeben
        e.Graphics.DrawString(zeile, schrift, pinsel,
e.MarginBounds.Left, yPos);
        //die Anzahl der gedruckten Zeilen erhöhen
        zeilenGedruckt++;
        //yPos auf die nächste Ausgabeposition setzen
        yPos = yPos + schrift.Height;
    }
    //kommt noch mehr Text
    //dann die nächste Seite anfangen
    if (textDatei.EndOfStream == false)
        e.HasMorePages = true;
    else
        e.HasMorePages = false;
}

```

Code 4.5: Das Drucken einer Textdatei mit mehreren Seiten

In der Methode `ButtonDrucken_Click()` wird zunächst überprüft, ob die Datei `druckdatei.txt` im Ordner `C:\test` vorhanden ist. Wenn das der Fall ist, wird die Datei über ein Feld `textDatei` geöffnet und die Druckausgabe über die Methode `Print()` der Instanz `printDocument1` gestartet. Das Feld `textDatei` vereinbaren wir dabei in der Klasse des Formulars als Variable vom Typ `StreamReader` und weisen ihm zunächst den Wert `null` zu. Dadurch können wir auch in der Methode `printDocument1_PrintPage()` sehr einfach auf die geöffnete Datei zugreifen. Allerdings können wir dann keine `using`-Konstruktion für die Freigabe mehr verwenden.

Wir erstellen daher eine `try ... catch ... finally`-Konstruktion und geben die Ressourcen der Datei im `finally`-Zweig über die Methode `Dispose()` frei, wenn vorher eine Instanz erzeugt wurde. Dabei wird die Datei auch automatisch geschlossen.

Hinweis:

Die Datei `druckdatei.txt` finden Sie bei den Beispielen auf der Online-Lernplattform. Sie können aber auch eine beliebige andere Textdatei benutzen. Wichtig ist lediglich, dass die Datei deutlich mehr als 60 Zeilen enthält. Andernfalls reicht nämlich eine Seite für die Ausgabe aus.

In der Methode `PrintDocument1_PrintPage()` werden dann wie gewohnt ein Pinsel und die Schrift festgelegt. Anschließend vereinbaren wir einige Variablen für die Steuerung der Druckausgabe.

Die Variable `zeilenProSeite` soll die maximale Anzahl der Druckzeilen pro Seite speichern. Den Wert berechnen wir über die Anweisung:

```
zeilenProSeite = e.MarginBounds.Height / schrift.Height;
```

Hier teilen wir die Höhe des Ausgabebereichs durch die Höhe der Schrift. Die Höhe des Ausgabebereichs erhalten wir dabei über die Eigenschaft `e.MarginBounds.Height` und die Höhe der Schrift über die Eigenschaft `Height` der Instanz `schrift`.

Die Variable `yPos` soll die Ausgabeposition der aktuellen Zeile speichern. Da die erste Zeile oben auf der Seite steht, setzen wir den Wert der Variablen zunächst auf `e.MarginBounds.Top`.

Die beiden Variablen `zeilenGedruckt` und `zeile` schließlich zählen mit, wie viele Zeilen bereits gedruckt worden sind, beziehungsweise speichern die gelesene Zeile aus der Datei.

Mit der Schleife

```
while ((zeilenGedruckt < zeilenProSeite) &&
       (textDatei.EndOfStream == false))
```

lassen wir dann so lange Zeilen ausgeben, bis die Anzahl der gedruckten Zeilen größer wird als die Anzahl der möglichen Zeilen oder bis das Ende der Datei erreicht wurde.

Die Anweisung

```
zeile = textDatei.ReadLine();
```

liest dabei die nächste Zeile aus der Datei ein. Ausgegeben wird die Zeile über die Anweisung:

```
e.Graphics.DrawString(zeile, schrift, pinsel,
e.MarginBounds.Left, yPos);
```

Die Ausgabe erfolgt dabei jeweils am linken Rand (`e.MarginBounds.Left`) und am aktuellen Wert von `yPos`.

Nach der Ausgabe erhöhen wir dann den Wert der gedruckten Zeilen und setzen die Y-Position für die nächste Ausgabe neu. Dazu addieren wir den Wert von `schrift.Height` auf den aktuellen Wert von `yPos`.

Nach der Schleife überprüfen wir, ob noch weitere Ausgaben folgen. Wenn das der Fall ist, setzen wir den Wert der Eigenschaft `e.HasMorePages`⁹ auf `true`. Damit wird die Methode `printDocument1_PrintPage()` erneut aufgerufen. Wenn keine weiteren Zeilen mehr folgen, setzen wir die Eigenschaft `e.HasMorePages` auf `false`, damit die Druckausgabe nach der letzten Seite nicht endlos fortgesetzt wird.



Bitte beachten Sie unbedingt:

Die Eigenschaft `HasMorePages` steuert, ob die Methode `PrintPage()` noch einmal aufgerufen wird. Sie müssen deshalb den Wert der Eigenschaft sehr sorgfältig setzen und auch sicherstellen, dass die Variablen für die Steuerung der Ausgabe korrekt initialisiert werden.

Damit Sie nicht versehentlich Unmengen von bedrucktem Papier produzieren, sollten Sie die Druckausgabe aus dem vorigen Code nicht direkt über den Drucker testen. Erstellen Sie einfach eine Seitenvorschau über die Komponente **PrintPreviewDialog** und kontrollieren Sie das Ergebnis. Wenn alles klappt, können Sie die Vorschau ja wieder aus der Anwendung entfernen.



Eine Demonstration zum Drucken eines Textfelds und einer Grafik finden Sie im Projekt **Drucken** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. In dem Projekt wird auch über das Steuerelement **PrintPreviewDialog** eine Druckvorschau erzeugt.

Eine Demonstration zur Ausgabe einer Textdatei über mehrere Seiten finden Sie im Projekt **DruckenTextdatei** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Zusammenfassung

Über die Eigenschaft `Position` der Klasse `FileStream` können Sie auf eine beliebige Position innerhalb einer Datei zugreifen.

Die Eigenschaft `Length` der Klasse `FileStream` liefert Ihnen die Größe einer Datei in Bytes.

Das .NET Framework kennt zahlreiche Methoden zur Ordner- und Dateiverwaltung. So können Sie zum Beispiel über `File.Delete()` Dateien löschen und mit `File.Copy()` Dateien kopieren.

Das Drucken erfolgt über das Steuerelement **PrintDocument**. Im Ereignis **PrintPage** des Steuerelements können Sie dann auf der Zeichenfläche des Druckers zeichnen.

9. *Has More Pages* lässt sich frei mit „Hat weitere Seiten“ übersetzen.

Aufgaben zur Selbstüberprüfung

- 4.1 Sie lesen aus einer Datei int-Werte ein. Welche Position dürfen Sie maximal mit der Eigenschaft `Position` setzen, um erfolgreich Daten zu lesen? Begründen Sie bitte Ihre Antwort.

- 4.2 Sie bewegen sich versehentlich beim Verarbeiten von Daten in einer Datei mit der Eigenschaft `Position` über das Ende der Datei hinaus. Was geschieht? Wann wird die Ausnahme ausgelöst?

- 4.3 Was liefert Ihnen die Anweisung
`FileInfo.Extension("D:\\test\\dokument.doc")`?

- 4.4 Wie können Sie die Länge einer Zeichenkette mit maximal 255 Zeichen ermitteln, die in einer binären Datei über die Methode `Write()` der Klasse `BinaryWriter` gespeichert wurde? Die Länge soll vor dem eigentlichen Einlesen der Zeichenkette festgestellt werden.

- 4.5 Wie starten Sie die Druckausgabe über das Steuerelement `PrintDocument`?

- 4.6 Wann tritt das Ereignis **PrintPage** für das Steuerelement **PrintDocument** ein?

- 4.7 Welche Eigenschaft müssen Sie auf welchen Wert setzen, wenn Sie mehrere Seiten ausdrucken wollen? In welcher Methode setzen Sie diesen Wert?



Schlussbetrachtung

Herzlichen Glückwunsch! Sie kennen jetzt verschiedene Techniken und Verfahren, um Daten auf einem Datenträger zu speichern und auch wieder zu lesen. Sie wissen, wie Sie die entsprechenden Standardmethoden von Steuerelementen einsetzen, und können auch selbst beliebige Daten in einer Datei verarbeiten. Außerdem haben Sie gelernt, Daten auf dem Drucker auszugeben.

Welches Verfahren zum Speichern und Lesen von Daten Sie in Ihren Programmen einsetzen, hängt immer auch von den Aufgaben ab. Wenn Sie in einem Programm nur eine Einstellung speichern wollen, können Sie auf die Registrierung zugreifen. Für die Verarbeitung umfangreicher Daten dagegen sind oft Textdateien oder binäre Dateien die bessere Wahl. Bei der Verarbeitung strukturierter Daten können Sie gut mit XML-Dateien arbeiten.

Zum Teil sind Sie beim Programmieren der Dateiverarbeitung selbst für den korrekten Zugriff und auch für die Sicherheit verantwortlich. Wenn Sie beispielsweise versehentlich eine Datei über die Klasse `FileStream` im falschen Modus öffnen, drohen unweiterbringliche Datenverluste. Auch Dateien, die Sie über `File.Delete()` löschen, lassen sich nicht wiederherstellen, da der Windows-Papierkorb umgangen wird. Arbeiten Sie daher sehr sorgfältig und benutzen Sie für die ersten Tests in jedem Fall Daten, die Sie nicht mehr benötigen. Zur Sicherheit sollten Sie einen eigenen Testordner anlegen und auch nur die Daten in diesem Testordner verwenden.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Die Methode zum Laden heißt `LoadFile()` und die Methode zum Speichern `SaveFile()`. Als Argument wird jeweils der Dateiname übergeben.
- 1.2 Die Daten in einer `RichTextBox` werden ohne weitere Angaben im Format RTF gespeichert.
- 1.3 Die Anweisung lautet:

```
richTextBox1.LoadFile("test.txt",
RichTextBoxStreamType.PlainText);
```

Wichtig ist vor allem das zweite Argument `RichTextBoxStreamType.PlainText`. Es gibt an, dass es sich um eine Textdatei handelt.

Kapitel 2

- 2.1 Die Anweisung könnte so aussehen:

```
richTextBox1.SelectionFont = new
Font(richTextBox1.SelectionFont,
richTextBox1.SelectionFont.Style ^ FontStyle.Bold);
```

Wichtig ist dabei die bitweise Exklusiv-Oder-Verknüpfung der aktuellen Formatierung mit dem Ausdruck `FontStyle.Bold`. Dadurch wird die Formatierung entweder ein- oder ausgeschaltet.

- 2.2 Zum Kopieren in die Zwischenablage verwenden Sie die Methode `Copy()` und zum Ausschneiden die Methode `Cut()`.
- 2.3 Die Anweisung könnte zum Beispiel so aussehen:

```
menuEintrag = new ToolStripMenuItem("&1 - " + name, null,
new EventHandler(this.Neu_Click));
```

Hinter dem Namen der Methode `Neu_Click` dürfen keine Klammern stehen.

- 2.4 Um die Anzahl der Einträge in einem Menü zu ermitteln, verwenden Sie die Eigenschaft `DropDownItems.Count`. Der entsprechende Ausdruck könnte dann so aussehen:

```
dateiToolStrip.DropDownItems.Count
```

Kapitel 3

- 3.1 Die Anweisung könnte zum Beispiel so aussehen:

```
StreamWriter textDatei = new
StreamWriter("C:\\test\\versuch.txt");
```

Damit die Ressourcen nach dem Zugriff direkt wieder freigegeben werden, sollten Sie eine `using`-Konstruktion verwenden.

- 3.2 Damit eventuell vorhandene Inhalte nicht überschrieben werden, geben Sie im Konstruktor als zweites Argument den booleschen Wert `true` an. Dann werden neue Inhalte an eventuell bereits vorhandene Inhalte angehängt.
- 3.3 Die Eigenschaft heißt `EndOfStream`. Sie hat den Wert `true`, wenn das Ende der Datei erreicht wurde.
- 3.4 Sie müssen an den Konstruktor jeweils einen Datei-Stream übergeben, den Sie vorher über die Klasse `FileStream` erzeugt haben.
- 3.5 Die verschiedenen Modi haben die folgenden Wirkungen:
- Es wird eine Datei mit dem angegebenen Namen erzeugt. Wenn diese Datei bereits vorhanden ist, wird sie geöffnet. Wenn neue Daten in die Datei geschrieben werden, gehen eventuell bereits vorhandene Inhalte vollständig verloren.
 - Die Datei wird geöffnet. Wenn sie nicht vorhanden ist, wird eine Ausnahme ausgelöst.
 - Die Datei wird geöffnet. Danach wird automatisch zum Dateiende gesprungen. Daten, die neu in die Datei geschrieben werden, werden an bereits vorhandene Inhalte angehängt. Wenn die Datei nicht vorhanden ist, wird sie neu erzeugt.
- 3.6 Die Eigenschaft heißt `this.Controls`.

Die Schleife könnte zum Beispiel so aussehen:

```
foreach (Control element in this.Controls)
```

Kapitel 4

- 4.1 Die maximale Position ist das Ende der Datei – 4. Die Zahl 4 steht dabei für die Größe des Datentyps in Bytes. Andernfalls wird beim Einlesen über das Ende der Datei hinaus gelesen.
- 4.2 Zunächst einmal geschieht nichts. Eine Ausnahme tritt erst beim nächsten Schreib- oder Lesezugriff auf.
- 4.3 Die Anweisung liefert die Dateierweiterung `.doc`.
- 4.4 Sie müssen das Byte einlesen, das in der Datei direkt vor der eigentlichen Zeichenkette gespeichert ist. Hier wird die Länge der folgenden Zeichenkette abgelegt.
- 4.5 Die Druckausgabe für das Steuerelement `PrintDocument` starten Sie über die Methode `Print()`.
- 4.6 Das Ereignis tritt bei der Ausgabe jeder Seite ein.
- 4.7 Sie müssen den Wert der Eigenschaft `e.HasMorePages` in der Methode `PrintPage()` des Steuerelements `PrintDocument` auf den Wert `true` setzen.

B. Glossar

Big Endian	Die Big-Endian-Darstellung ist eine Variante für die Darstellung einer Bytereihenfolge. Das höchstwertige Byte wird zuerst dargestellt. Das entspricht der gewohnten Darstellung.
Delegat	Ein Delegat stellt einen Verweis auf eine Methode dar.
Druckauftrag	Ein Druckauftrag fasst Informationen zusammen, die auf einem Drucker ausgegeben werden sollen. Die Ausgabe erfolgt in der Regel erst dann, wenn der Druckauftrag vollständig ist.
Editor	Ein Editor ist ein Programm zum Erfassen und Bearbeiten von Informationen.
Little Endian	Die Little-Endian-Darstellung ist eine Variante für die Darstellung einer Bytereihenfolge. Das kleinstwertige Byte wird zuerst dargestellt. Hier wird die übliche Reihenfolge also umgekehrt.
Quickinfos	Quickinfos sind kurze Hinweistexte. Sie werden normalerweise angezeigt, wenn der Mauszeiger einen kurzen Moment auf einem bestimmten Element stehen bleibt.
Rich Text Format	Siehe RTF.
RTF	Das RTF-Format (<i>Rich Text Format</i> ; wörtlich übersetzt „reicher Text“) ist ein weitverbreitetes Format für die Speicherung von Textdokumenten mit Formatierungen. Es kann von nahezu allen Textverarbeitungsprogrammen gelesen und auch erzeugt werden.
Share-Modus	Der <i>Share-Modus</i> legt fest, ob andere Anwendungen oder Anwender auf eine Datei zugreifen dürfen, solange sie geöffnet ist.
Shortcut	Über einen <i>Shortcut</i> kann eine Menüfunktion mit der Tastatur aufgerufen werden. Der Aufruf ist auch dann möglich, wenn das Menü nicht geöffnet ist.
Stream	Ein <i>Stream</i> ist ein unformatierter Datenstrom. Die Interpretation erfolgt durch das Zielgerät, das den Stream verarbeitet.
Tooltipp	Siehe Quickinfos.
Unicode	<i>Unicode</i> ist ein Zeichensatz für die Darstellung von alphanumerischen Werten. Er kann mindestens 65 536 unterschiedliche Zeichen abbilden.
UTF-8	UTF-8 ist eine Variante des <i>Unicode</i> -Zeichensatzes.
Zugriffsmodus	Der Zugriffsmodus legt fest, wie auf eine Datei zugegriffen werden kann – zum Beispiel, ob die Daten nur gelesen oder sowohl gelesen als auch geschrieben werden können.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in Visual C# mit Visual Studio 2015*. Ideal für Programmieranfänger geeignet. 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Ein ToolStripContainer im Formular	5
Abb. 1.2	Der angedockte ToolStripContainer	5
Abb. 1.3	Das Formular mit der Symbolleiste	6
Abb. 1.4	Die eingefügten Standardsymbole	7
Abb. 1.5	Das Formular	7
Abb. 2.1	Das Formular mit Menü- und Symbolleiste.....	12
Abb. 2.2	Der Zähler beim Anlegen neuer Dokumente (oben in der Titelleiste)	14
Abb. 4.1	Eine Druckvorschau	56
Abb. H.1	Der Schummeleditor	74

E. Tabellenverzeichnis

Tab. 3.1	Modi für das Öffnen von Dateien	34
Tab. 3.2	Zugriffsmodi für Dateien	34
Tab. 3.3	Share-Modi für Dateien	35
Tab. 3.4	Die Methoden der Klasse BinaryReader zum Lesen und Interpretieren von Daten	37
Tab. 4.1	Methoden zur Verwaltung von Dateien	52
Tab. 4.2	Methoden für die Verwaltung von Ordnern	53
Tab. 4.3	Methoden und Eigenschaften für die Arbeit mit Laufwerken	54

F. Codeverzeichnis

Code 1.1	Das Laden und Speichern über das Steuerelement RichTextBox	3
Code 1.2	Die Methoden zum Laden und Speichern der Textdateien	9
Code 2.1	Die Methode NeuesDokument()	13
Code 2.2	Die Methode Speichern()	14
Code 2.3	Die Anweisungen für die Methode SaveFileDialog1_FileOk()	15
Code 2.4	Das Speichern	15
Code 2.5	Die Methode DateiLaden()	16
Code 2.6	Die Methode openFileDialog1_FileOk()	16
Code 2.7	Die Methode FormClosing()	17
Code 2.8	Die Methode FettToolStripButton_Click()	19
Code 2.9	Der Dialog Zeichen im Einsatz	19
Code 2.10	Die Methode ZentriertToolStripButton_Click() für die zentrierte Ausrichtung	20
Code 2.11	Die Methoden für die Funktionen im Menü Bearbeiten	21
Code 2.12	Das Speichern des Dateinamens in der Registrierung	22
Code 2.13	Das dynamische Einfügen eines Eintrags im Menü Datei	25
Code 2.14	Die Methode LetzteDatei_Click()	26
Code 2.15	Die Methode Form1_Load()	27
Code 3.1	Das Speichern von Texten aus einem Listenfeld	30
Code 3.2	Das Einlesen aus einer Textdatei in ein Listenfeld	32
Code 3.3	Das Schreiben von Daten in binärer Form	36
Code 3.4	Das Lesen von Daten im Format int	38
Code 3.5	Zugriff auf alle Textboxen in einem Formular	40
Code 3.6	Speichern und Lesen für sämtliche Textboxen in einem Formular	41
Code 3.7	Das Schreiben der Bestenliste in eine binäre Datei	44
Code 3.8	Das Lesen der Bestenliste aus der binären Datei	44
Code 4.1	Direktes Positionieren in einer Datei (es handelt sich um ein Fragment)	48
Code 4.2	Das Suchen in der Bestenliste	50
Code 4.3	Die Ausgabe einer Grafik und eines Textes auf dem Drucker	55
Code 4.4	Das Erzeugen einer Druckvorschau	56
Code 4.5	Das Drucken einer Textdatei mit mehreren Seiten	58

G. Sachwortverzeichnis

A

Absatzausrichtung 20

D

Datei- und Ordnerverwaltung 52

Dateifunktion 13

Dialog

 Zeichen 19

Druckauftrag 54

Drucken 54

Druckvorschau 55

E

Eigenschaft

 Controls 40

 DefaultExt 15

 Modified 11

Ereignis

 FileOK 8

 FormClosing 17

Eventhandler 23

Exklusiv-Oder-Verknüpfung 18

F

Format UTF-8 31

K

Klasse

 BinaryReader 37

 BinaryWriter 36

 FileStream 33

 StreamReader 30

 StreamWriter 30

M

Methode

 für die Verwaltung von Ordnern . 53

 zur Verwaltung von Dateien 52

Methoden und Eigenschaften

 für die Arbeit mit Laufwerken 54

Modus

 für das Öffnen von Dateien 34

R

RichTextBox 3

S

Share-Modus 35

 für Dateien 35

Steuerelement

 PrintDocument 54

 PrintPreviewDialog 55

 ToolStrip 5

 ToolStripContainer 4

Suchen und Positionieren

 in einer Datei 47

T

Text-Editor

 einfacher 4

Textverarbeitung

 einfache 11

W

Wert

 vieler gleicher Steuerelemente

 verarbeiten 39

Z

Zeichenformatierung 18

Zugriff

 direkter, auf das letzte Dokument 22

 direkter, auf Dateien 30

 sequenzieller 47

Zugriffsmodus 34

 für Dateien 34

Zwischenablage 20



H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPHP13D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Schicken Sie für die Lösungen bitte jeweils das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

Beschreiben Sie bei allen Aufgaben zusätzlich, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Änderungen Sie vornehmen beziehungsweise welche Steuerelemente Sie verwenden.

- Erweitern Sie die Textverarbeitung aus dem Kapitel 2 um eine Funktion, die beim Öffnen einer Datei automatisch eine Sicherungskopie dieser Datei erstellt. Die Sicherungskopie soll den ursprünglichen Namen der Datei erhalten und die Erweiterung .bak.

Stellen Sie dabei sicher, dass eine eventuell bereits vorhandene Sicherungskopie überschrieben wird.

10 Pkt.

- Programmieren Sie einen Schummeleditor für die binäre Version der Bestenliste des Pong-Spiels. Über diesen Editor soll ein direkter Zugriff von außen auf die Einträge der Bestenliste möglich sein. Die geänderte Bestenliste muss sich im Pong-Spiel verwenden lassen.

Der Editor soll sowohl eine Änderung der Punktzahlen als auch eine Änderung der Namen ermöglichen. Die Länge des Namens soll dabei beliebig sein.

Eine Funktion zum Löschen von Einträgen oder zum Hinzufügen von Einträgen müssen Sie nicht programmieren. Die Bestenliste soll immer fest zehn Einträge enthalten.

Achten Sie bitte bei der Lösung dieser Aufgabe darauf, dass die geänderten Daten auch wieder in der richtig sortierten Reihenfolge gespeichert werden müssen.

Die entsprechenden Funktionen zum Erzeugen, Sortieren, Laden und Speichern der Liste finden Sie in der Datei Score.cs, die wir unter anderem auch im Pong-Spiel eingesetzt haben. Diese Datei können Sie nach dem Anlegen des Projekts für den Schummel-Editor in den Projektordner kopieren und anschließend über die Funk-

tion Projekt/Vorhandenes Element hinzufügen ... in das Projekt für den Schummel-Editor aufnehmen. Passen Sie dann die vorhandenen Methoden so an, dass Sie sie für den Schummel-Editor benutzen können, und ergänzen Sie die erforderlichen neuen Methoden. Denken Sie dabei bitte daran, dass Sie unter Umständen auch den Namensraum in der Datei `Score.cs` verändern müssen.

Sie können die entsprechende Funktionalität aber auch komplett neu programmieren. Welche Variante Sie benutzen, ist Ihnen freigestellt. Bitte dokumentieren Sie aber in jedem Fall Ihr Vorgehen.

Der Editor könnte zum Beispiel so aussehen:

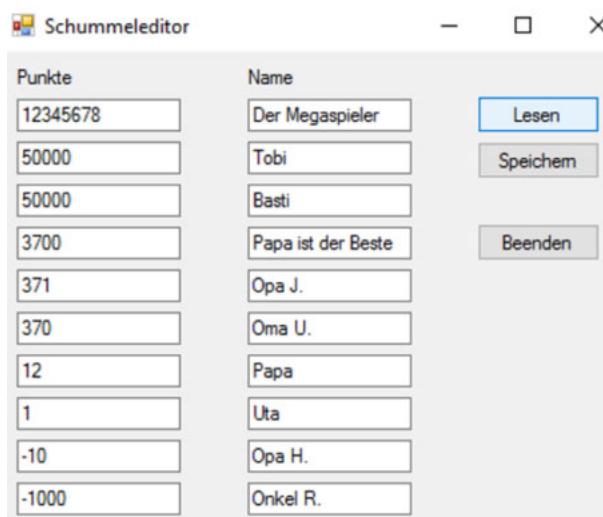


Abb. H.1: Der Schummleditor

80 Pkt.

3. Erstellen Sie eine Funktion, mit der die Bestenliste aus dem Pong-Spiel auf dem Drucker ausgegeben werden kann.

Hinweis:

Die Klasse `Score` enthält bereits eine Methode, die Sie für die Lösung dieser Aufgabe verwenden können.

10 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit der
Windows Presentation Foundation

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0919N01

CSHP14D

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit der Windows Presentation Foundation

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Arbeiten mit der Windows Presentation Foundation

Inhaltsverzeichnis

Einleitung	1
1 Grundlagen	3
1.1 Was ist die WPF?	3
1.2 Die XAML-Beschreibungen	4
1.3 Projekt für eine WPF-Anwendung erstellen	6
Zusammenfassung	11
2 Die erste WPF-Anwendung	13
Zusammenfassung	18
3 Besonderheiten der WPF	19
3.1 Container und Layouts	19
3.1.1 Das StackPanel	19
3.1.2 Das WrapPanel	21
3.1.3 Das DockPanel	21
3.1.4 Das Grid	21
3.2 Darstellung von Grafiken	22
3.3 Die Weiterleitung von Ereignissen	24
Zusammenfassung	26
4 Eine Textverarbeitung als WPF-Anwendung	28
4.1 Vorüberlegungen	28
4.2 Das Fenster	29
4.3 Das Menüband	32
4.4 Die Dateifunktionen	38
4.5 Funktionen für die Zwischenablage	44
4.6 Ein wenig Feinschliff	45
Zusammenfassung	47
Schlussbetrachtung	49

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	50
B.	Glossar	52
C.	Literaturverzeichnis	53
D.	Abbildungsverzeichnis	54
E.	Tabellenverzeichnis	55
F.	Codeverzeichnis	56
G.	Sachwortverzeichnis	57
H.	Einsendeaufgabe	59

Einleitung

Bisher haben Sie sich sehr intensiv mit Windows Forms beschäftigt. Ab diesem Studienheft werden Sie eine weitere Möglichkeit kennenlernen, grafische Oberflächen zu erstellen – die **Windows Presentation Foundation (WPF)**¹.

Im Einzelnen lernen Sie in diesem Studienheft:

- was die WPF ist,
- wie sich die WPF und Windows Forms unterscheiden,
- was sich hinter XAML verbirgt,
- wie über XAML die Oberfläche einer Anwendung beschrieben wird,
- wie Sie mit Visual Studio ein Projekt für eine WPF-Anwendung erstellen,
- wie ein WPF-Projekt in Visual Studio aufgebaut ist,
- was Code-Behind-Dateien sind,
- wie Sie Ereignisse in der WPF-Anwendung verarbeiten,
- wie Sie eigene Ereignishandler in einer WPF-Anwendung festlegen,
- was sich hinter Containern und Layouts einer WPF-Anwendung verbirgt,
- wie Sie verschiedene Container verwenden,
- welche Besonderheiten es bei der Darstellung von Grafiken in der WPF gibt,
- wie Sie grafische Elemente drehen und vergrößern beziehungsweise verkleinern,
- wie Sie Ereignisse weiterleiten,
- was Bubbling und Tunneling Events sind,
- wie Sie ein Menüband in einer WPF-Anwendung erstellen,
- wie Sie Standarddialoge wie **Offnen** und **Speichern unter** in einer WPF-Anwendung einsetzen,
- wie Sie Texte in einer WPF-Anwendung in einer **RichTextBox** speichern, laden und verarbeiten,
- was sich hinter Befehlen einer WPF-Anwendung verbirgt,
- wie Sie diese Befehle gezielt einsetzen,
- wie Sie ein Kontext-Menü für eine WPF-Anwendung erstellen und
- wie Sie für eine **RichTextBox** eine Rechtschreibprüfung durchführen.

Ein wichtiger Hinweis:

Auch beim Öffnen eines Projekts für eine WPF-Anwendung erscheint unter Umständen nicht direkt das Fenster der Anwendung. Öffnen Sie es dann über einen Doppelklick auf den Eintrag der Datei **MainWindows.xaml** im Projektexplorer.

Christoph Siebeck

1. Frei übersetzt bedeutet *Windows Presentation Foundation* so viel wie „Windows Präsentationsbasis“.

1 Grundlagen

In diesem Kapitel beschäftigen wir uns mit den Grundlagen der WPF. Schauen wir uns zuerst an, was sich überhaupt hinter der WPF verbirgt.

1.1 Was ist die WPF?

Die WPF ist genau wie Windows Forms eine Plattform für die Entwicklung von grafischen Benutzeroberflächen. Sie fasst Konzepte, Technologien und auch konkrete Klassen für die Gestaltung zusammen.

Die **Windows Presentation Foundation** – kurz WPF genannt – ist eine Plattform zur Entwicklung von grafischen Oberflächen.



Anders als Windows Forms-Anwendungen, die direkt auf Ebene des Betriebssystems ablaufen, können Sie mit der WPF sowohl Anwendungen für das Betriebssystem als auch Anwendungen für einen Internet-Browser erstellen. Diese Anwendungen werden nicht direkt über das Betriebssystem ausgeführt, sondern in einem Browser angezeigt.

Hinweis:

Wir werden uns in diesem Lehrgang vor allem auf die WPF-Anwendungen für das Betriebssystem konzentrieren.

Bei der Entwicklung von WPF-Anwendungen wird konsequent zwischen der Präsentation – dem Erscheinungsbild der Anwendung – und der Logik – also dem eigentlichen Programmverhalten – getrennt. Für die Darstellung der Oberfläche wird die Beschreibungssprache **XAML** verwendet. Das Programmverhalten wird durch den Quelltext einer Programmiersprache wie C# oder auch Visual Basic beschrieben.

XAML steht für *eXtensible Application Markup Language*^{a)}. XAML basiert auf XML und stellt die Informationen ebenfalls strukturiert in Textform dar.



a) *Extensible Application Markup Language* bedeutet übersetzt „Erweiterbare Anwendungsauszeichnungssprache“.

Durch diese konsequente Trennung können die Oberflächen von Anwendungen sehr einfach durch Grafiker beziehungsweise Designer erstellt werden, während sich Programmierer um die Logik kümmern. Zwar findet sich diese Trennung in Teilen auch in einem Windows Forms-Projekt von Visual Studio wieder, allerdings erfolgt hier auch die Beschreibung der Oberfläche in der Datei **Designer.cs** durch C#-Anweisungen und die Bearbeitung ist nur direkt mit dem Editor beziehungsweise Designer von Visual Studio möglich. Damit müsste ein Grafiker entweder über detaillierte C#-Kenntnisse verfügen oder ein Werkzeug verwenden, das sich im Wesentlichen an Programmierer richtet.

Bei einer WPF-Anwendung dagegen gibt es spezielle Programme zur Gestaltung von Oberflächen – wie zum Beispiel Blend für Visual Studio von Microsoft –, die sich vor allem an Grafiker richten.



Bei WPF-Anwendungen wird konsequent zwischen der Präsentation und der Logik getrennt. Die Darstellung erfolgt in getrennten Dateien und auch mit unterschiedlichen Sprachen.

1.2 Die XAML-Beschreibungen

Bei den XAML-Beschreibungen der grafischen Oberflächen von WPF-Anwendungen handelt es sich um strukturierte Texte, die Sie mit einem beliebigen Text-Editor erstellen können.

Die Beschreibung für eine Oberfläche eines sehr einfachen Hallo-Welt-„Programms“ könnte zum Beispiel so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<Label xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation">
    Hallo Welt. Es grüßt Dich Max Meier.</Label>
```

Code 1.1: Die XAML-Beschreibung für ein Hallo-Welt-„Programm“

Zunächst einmal wird mit der Zeile

```
<Label xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation">
```

die Vereinbarung eines Labels eingeleitet. Über den Namensraum, der hinter `xmlns` angegeben wird, wird dabei die Verbindung zwischen dem Namen `Label` und der dazugehörigen WPF-Klasse `Label` erstellt.

Danach folgt der **Content** – der Inhalt – des Labels. In unserem Fall wird der Text „Hallo Welt. Es grüßt Dich Max Meier.“ angezeigt.

Abgeschlossen wird die Vereinbarung durch `</Label>`.

Übernehmen Sie den vorigen Code einmal mit einem beliebigen Text-Editor. Speichern Sie ihn dann im Format UTF-8 mit der Erweiterung **.XAML** ab und führen Sie ihn direkt aus dem Explorer von Windows mit einem Doppelklick aus. Windows startet den Browser und zeigt die Oberfläche an. Sie sollte ungefähr so aussehen:

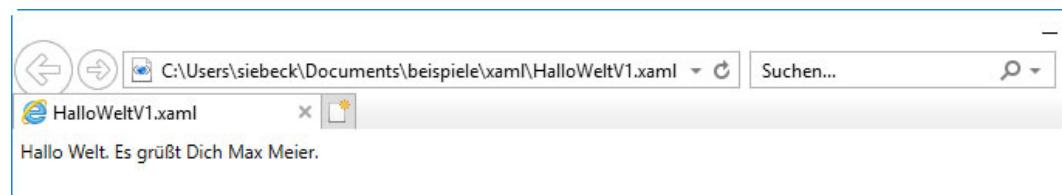


Abb. 1.1: Das erste XAML-„Programm“

Hinweis:

Wenn Sie den Editor von Windows benutzen, denken Sie daran, den Dateinamen beim Speichern in Anführungszeichen zu setzen. Andernfalls hängt der Editor automatisch noch die Erweiterung **.TXT** an.

Achten Sie auch darauf, dass Sie die Datei im UTF-8-Format speichern. Sonst erhalten Sie beim Ausführen lediglich eine Fehlermeldung, da die Sonderzeichen ß und ü nicht korrekt interpretiert werden können.

Falls Sie beim Ausführen des Programms eine Fehlermeldung erhalten, überprüfen Sie, ob Sie möglicherweise mitten in einer Anweisung einen Zeilenumbruch gesetzt haben.

Sie finden die Datei **HalloWeltV1.xaml** auch bei den Beispielen auf Ihrer Online-Lernplattform.

Sie können natürlich auch mehrere Elemente in einer XAML-Datei beschreiben – zum Beispiel ein Label und eine Schaltfläche. Dabei müssen Sie allerdings auf eine Besonderheit achten: Anders als bei einer Windows Forms-Anwendung, bei der Sie nahezu beliebig viele Steuerelemente in dem eigentlichen Fenster ablegen können, darf das Fenster einer WPF-Anwendung – das **Stammelement** – lediglich ein direkt untergeordnetes Element enthalten. Die weiteren Elemente müssen zwingend in einer Art Container abgelegt werden.

Das heißt für unser Beispiel: Wir benötigen nicht nur die Elemente für das Label und die Schaltfläche, sondern auch einen Container, in dem diese beiden Elemente abgelegt werden. Das kann zum Beispiel ein **StackPanel** sein. Es ordnet die untergeordneten Elemente automatisch in einer Art Stapel an.

Mit den Containern und Layouts werden wir uns im weiteren Verlauf dieses Studienhefts noch intensiver beschäftigen.



Damit ergibt sich eine hierarchische Struktur mit dem Fenster beziehungsweise der Seite ganz oben. Darunter liegt das **StackPanel**, das das Label und die Schaltfläche enthält. Die entsprechende XAML-Beschreibung finden Sie im folgenden Code 1.2:

```
<!-- bitte in einer Zeile eingeben -->
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Label>Hallo Welt. Es grüßt Dich Max Meier.</Label>
<Button>Beenden</Button>
</StackPanel>
```

Code 1.2: Die erweiterte XAML-Beschreibung

Das Ergebnis im Browser sieht so aus:

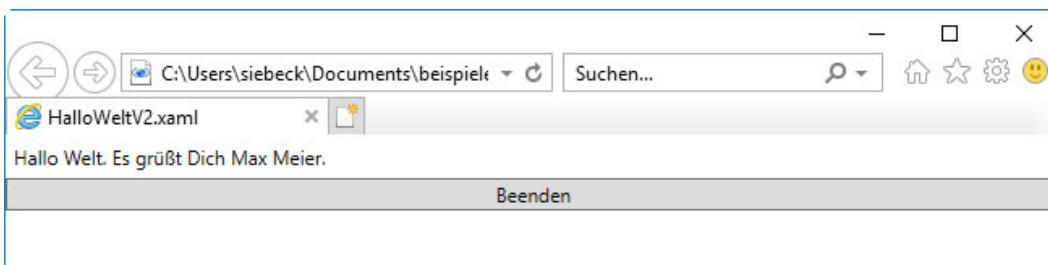


Abb. 1.2: Das erweiterte Hallo-Welt-„Programm“



Sie finden das Beispiel auch bei den Beispielen auf Ihrer Online-Lernplattform unter dem Namen **HalloWeltV2.xaml**.

Die Schaltfläche im vorigen Beispiel lässt sich auch anklicken, hat allerdings keinerlei Wirkung. Denn in der XAML-Datei wird ja lediglich die Oberfläche beschrieben und nicht die Funktionalität. Sie könnten nun die Beschreibung weiter ausbauen, in ein C#-Projekt importieren und dort die Programmlogik in einem Quelltext ergänzen. Es geht aber auch sehr viel einfacher und komfortabler – nämlich direkt über entsprechende Projekte von Visual Studio.

1.3 Projekt für eine WPF-Anwendung erstellen

Um ein Projekt für eine WPF-Anwendung zu erstellen, klicken Sie im Menü **Datei** auf **Neu/Projekt ...**

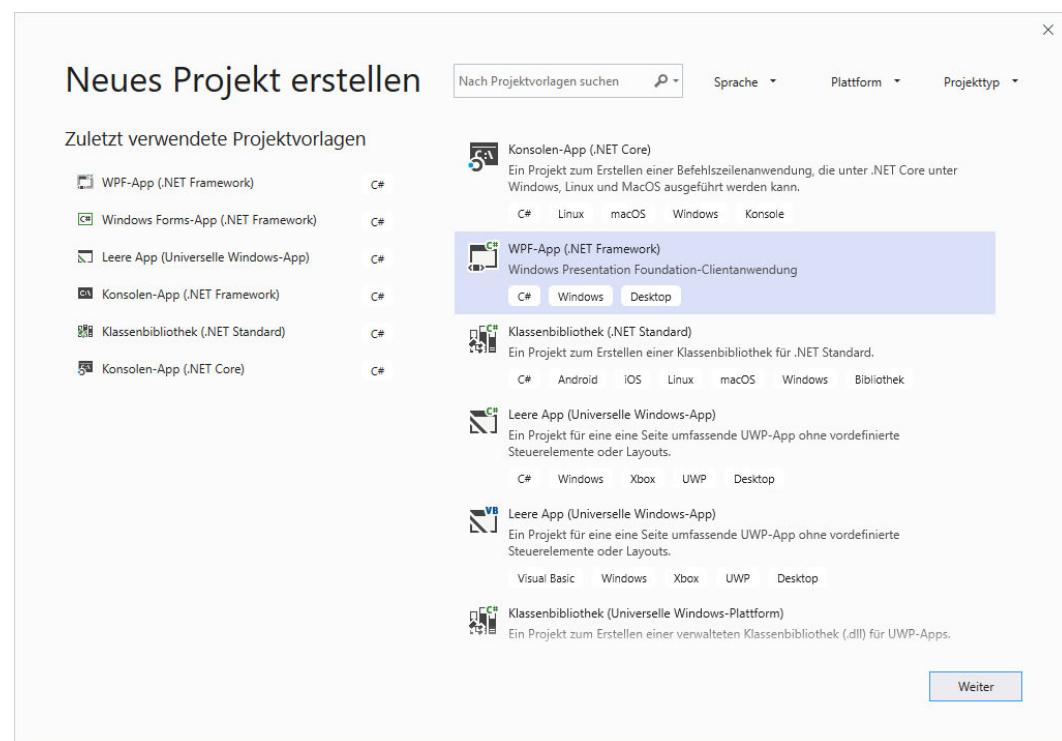


Abb. 1.3: Der Dialog **Neues Projekt erstellen** mit dem markierten Eintrag **WPF-App**

Markieren Sie dann im Dialog **Neues Projekt erstellen** in der mittleren Liste den Eintrag **WPF-App (.NET-Framework)** für die Programmiersprache C# und klicken Sie auf **Weiter**. Wählen Sie im nächsten Schritt den Speicherort aus und geben Sie wie gewohnt einen Namen in das Feld **Projektname** ein. Abschließend klicken Sie auf die Schaltfläche **Erstellen**.

Hinweis:

Wir benutzen in unserem Beispiel den Namen **WPFDemo1**.

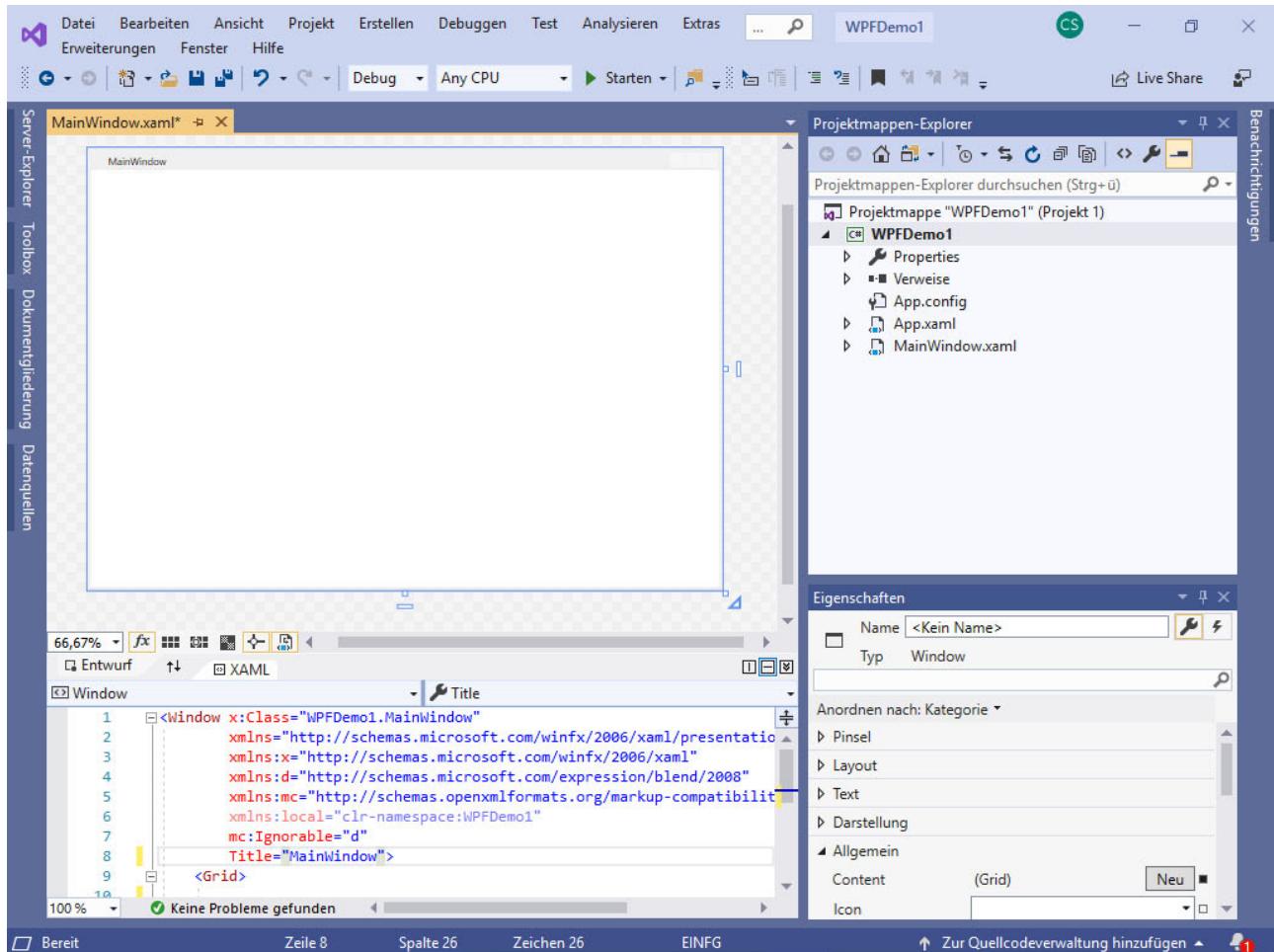


Abb. 1.4: Das Gerüst für die WPF-Anwendung

Visual Studio erstellt dann das Gerüst für die Anwendung und zeigt Ihnen in der Mitte das Fenster der Anwendung und darunter den dazugehörigen XAML-Code an. Links finden Sie unter anderem eine Registerkarte für das Einblenden der Toolbox und rechts den Projektmappen-Explorer.

Hinweis:

Welche Bereiche genau eingeblendet werden, hängt auch von Ihren Einstellungen ab. Unter Umständen werden daher bei Ihnen andere Elemente angezeigt. Sie können aber über das Menü **Ansicht** gezielt einzelne Elemente ein- und ausblenden.

Wenn Sie einmal genau hinsehen, werden Sie zum Beispiel oben an den Texten auf den Registerungen beziehungsweise an den Einträgen im Projektmappen-Explorer erkennen, dass Visual Studio nicht nur die XAML-Dateien für das Fenster (**MainWindow.xaml**) und die eigentliche Anwendung (**App.xaml**) angelegt hat, sondern auch die C#-Quelltextdateien für die Programmlogik beziehungsweise die Klassen. Sie tragen jeweils noch die Erweiterung **.cs**.

Die Quelltextdateien für die Programmlogik werden auch **Code-Behind-Dateien** genannt. *Code behind* lässt sich übersetzen mit „der Code dahinter“.



Erstellen wir jetzt einmal unser Hallo-Welt-Programm als WPF-Anwendung. Fügen Sie dazu bitte zuerst ein Label aus der Toolbox in das Fenster in der Mitte ein. Dabei gibt es keinerlei Besonderheiten.

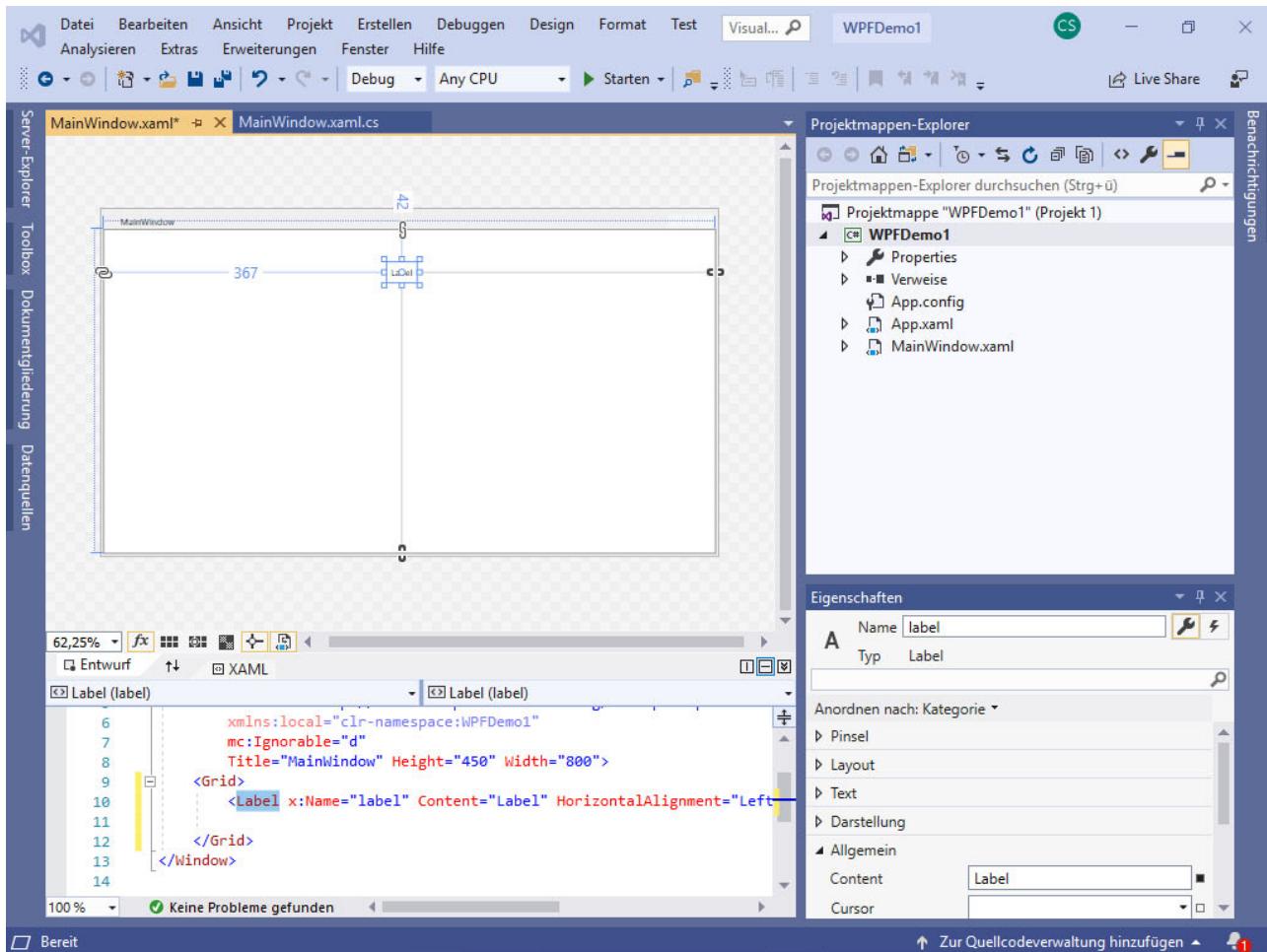


Abb. 1.5: Das eingefügte Label

Nach dem Einfügen wird das Label nicht nur oben im Designer angezeigt, sondern es werden auch die entsprechenden Anweisungen unten im XAML-Dokument automatisch ergänzt. Sie können also die Elemente direkt in der XAML-Beschreibung über die entsprechenden Attribute wie zum Beispiel `Content` für den Inhalt bearbeiten. Gerade zu Beginn ist es allerdings häufig sehr viel einfacher und auch sicherer, wenn Sie die Eigenschaften wie gewohnt über das Eigenschaftenfenster setzen.

Hinweis:

Falls das Eigenschaftenfenster bei Ihnen nicht angezeigt wird, können Sie es über den entsprechenden Eintrag im Menü **Ansicht** einblenden.

Bitte beachten Sie aber, dass sich die Eigenschaften von WPF-Steuerelementen und Windows Forms-Steuerelementen zum Teil deutlich unterscheiden. So setzen Sie den Text auf einem Element zum Beispiel nicht über eine Eigenschaft **Text**, sondern über die Eigenschaft **Content**.

Probieren Sie das jetzt bitte aus. Ändern Sie den Text auf dem Label zum Beispiel in `Hello Welt`.

Fügen Sie dann eine Schaltfläche ein und ändern Sie den Text der Schaltfläche über die Eigenschaft **Content** in `Beenden`. Passen Sie abschließend bitte noch den Titel des Fensters an. Markieren Sie dazu das gesamte Fenster im Designer und setzen Sie die Eigenschaft **Title** zum Beispiel auf `WPF Hello Welt`. Wenn Sie wollen, können Sie auch noch die Größe des Fensters ändern. Danach müssen Sie unter Umständen aber die beiden Steuerelemente wieder neu positionieren.

Bitte beachten Sie:

Der Designer fügt automatisch in jedes Fenster einer WPF-Anwendung ein **Grid**^{a)} als Container ein. Um das komplette Fenster – und nicht nur das **Grid** – zu markieren, müssen Sie relativ weit außen klicken. Wenn Sie nicht sicher sind, welches Element ausgewählt ist, sehen Sie oben im Eigenschaftenfenster nach. Dort wird Ihnen das markierte Element angezeigt.



- a) *Grid* bedeutet übersetzt so viel wie „Gitter“.

Das Fenster sollte nun ungefähr so aussehen:

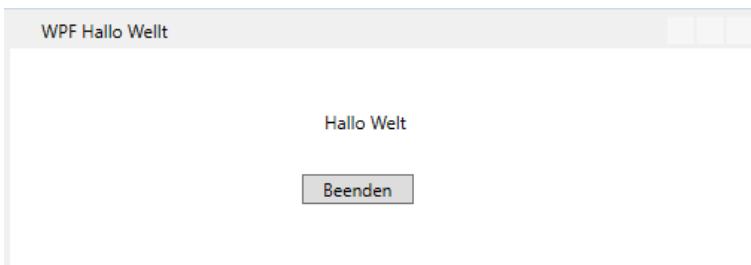


Abb. 1.6: Das Fenster nach den Änderungen

Tipp:

Über das Kombinationsfeld links unterhalb des Fensters können Sie die Größe der Darstellung verändern. Bitte beachten Sie aber, dass es zwei dieser Felder gibt. Das obere verändert die Anzeige des Fensters, das untere die Anzeige im XAML-Dokument.

Lassen Sie dann die Anwendung ausführen. Dazu rufen Sie wie gewohnt die Funktion **Debuggen/Starten ohne Debugging** auf. Alternativ können Sie auch die Tastenkombination **Strg + F5** verwenden.

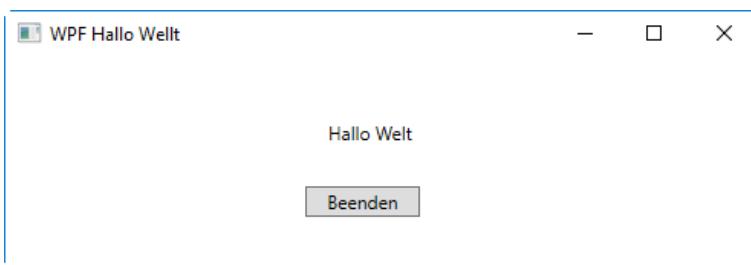


Abb. 1.7: Die erste ausgeführte WPF-Anwendung

Damit die Anwendung auch wirklich funktioniert, fehlt jetzt noch die Reaktion auf das Anklicken der Schaltfläche. Diese Reaktion können Sie nach einem Doppelklick auf das Steuerelement im Designer festlegen. Alternativ können Sie auch im Eigenschaftenfenster im Bereich **Ereignisse** wie gewohnt in das Feld hinter dem Ereignis doppelklicken. Visual Studio wechselt dann in den Quelltext-Editor und legt automatisch eine Methode für die Verarbeitung des Ereignisses an.

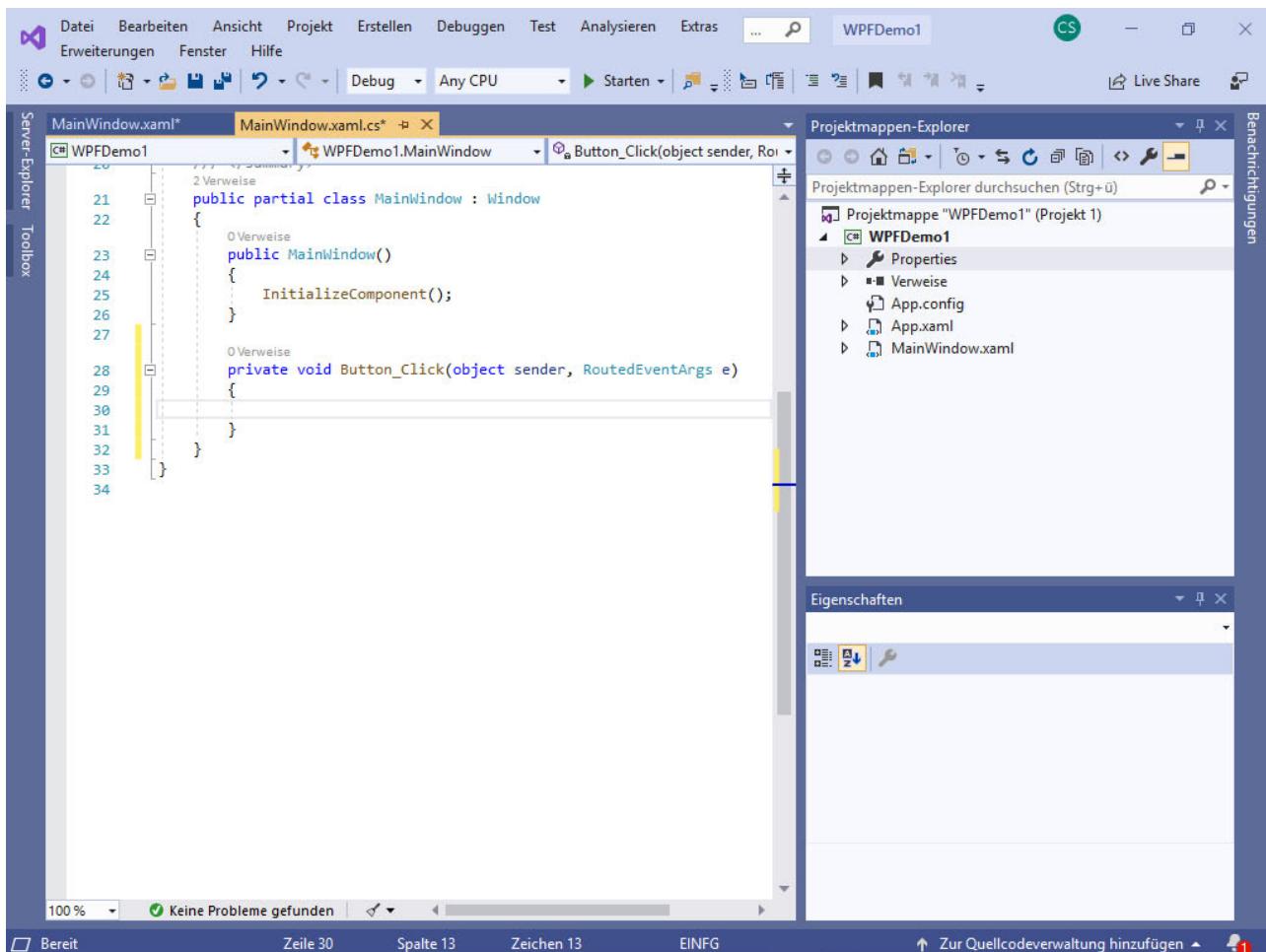


Abb. 1.8: Das Bearbeiten der Methode `Button_Click ()`

Lassen Sie die Anwendung beim Anklicken der Schaltfläche schließen. Rufen Sie dazu die Methode `Close()` in dem Ereignis `Click` für die Schaltfläche auf.

Testen Sie das Programm dann noch einmal. Jetzt sollte es sich auch mit der Schaltfläche beenden lassen.

Jetzt wollen wir uns noch kurz ansehen, wie Sie eine WPF-Browseranwendung erstellen.

Dazu wählen Sie im Fenster **Neues Projekt erstellen** den Eintrag **WPF-Browser-App** für die Programmiersprache C# und legen einen Namen fest. Visual Studio erstellt dann automatisch die einzelnen Dateien und zeigt sie im Designer beziehungsweise Editor an. Anders als bei Windows-Anwendungen wird bei einer Browser-App eine Seite – eine **Page** – als Stammelement verwendet. Außerdem erfolgt die Anzeige nach einigen Sicherheitsabfragen im Browser.

Eine sehr einfache Hallo-Welt-Anwendung als WPF-Browseranwendung finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **WPFDemo2**.



Zusammenfassung

Die Windows Presentation Foundation – kurz WPF genannt – ist eine Plattform zur Entwicklung von grafischen Oberflächen.

Mit der WPF können Sie neben Anwendungen für das Betriebssystem auch Anwendungen erstellen, die in einem Browser laufen.

Die Beschreibung der Oberfläche erfolgt mit XAML.

Visual Studio stellt Ihnen spezielle Projekte für WPF-Anwendungen zur Verfügung.

Die Eigenschaften von WPF-Steuerelementen stimmen nicht mit den Eigenschaften der entsprechenden Windows Forms-Steuerelemente überein.

Die Reaktion auf das Standardereignis für ein Steuerelement legen Sie durch einen Doppelklick auf das Steuerelement an. Visual Studio legt automatisch die entsprechende Methode an und wechselt in den Editor für den Quelltext.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Was ist ein besonderes Kennzeichen bei der Entwicklung von WPF-Anwendungen?

- 1.2 Können Sie in das Fenster einer WPF-Anwendung direkt auf der obersten Ebene ein Label und eine Schaltfläche einfügen? Begründen Sie bitte kurz Ihre Antwort.

- 1.3 Wie viele XAML-Dateien werden bei einem Projekt für eine WPF-Anwendung automatisch erstellt? Wofür stehen diese Dateien?

- 1.4 Wie heißt die Eigenschaft, mit der Sie bei einem WPF-Steuerelement **Label** den angezeigten Text setzen?

2 Die erste WPF-Anwendung

In diesem Kapitel werden wir die erste Windows Forms-Anwendung, die Sie erstellt haben, als WPF-Anwendung umsetzen. Dabei konzentrieren wir uns vor allem auf die Besonderheiten. Das grundsätzliche Vorgehen und die Programmlogik stellen wir Ihnen nicht noch einmal im Detail vor.

Legen Sie eine neue WPF-Anwendung mit Visual Studio an. Als Titel können Sie zum Beispiel **WPFspielereien** benutzen.

Setzen Sie den Titel für das Fenster auf **Spielereien**.

Fügen Sie anschließend zwei Schaltflächen mit dem Text **Kopieren** beziehungsweise dem Text **Beenden** am unteren Rand ein und vergeben Sie sprechende Namen an die Schaltflächen. Diese Namen können Sie oben im Eigenschaftenfenster im Feld **Name** eintragen.

Beim Anklicken der Schaltfläche **Beenden** lassen Sie die Anwendung über die Methode `Close()` schließen.

Fügen Sie anschließend oben in dem Fenster zwei Labels ein und vergeben Sie auch an die Labels sprechende Namen – zum Beispiel wie im Windows Forms-Projekt `labelQuelle` und `labelZiel`. Setzen Sie danach die Eigenschaft **Content** der Labels auf einen beliebigen Text. Lassen Sie dabei für das zweite Label ein wenig Platz nach rechts.

Übernehmen Sie dann für das Anklicken der Schaltfläche **Kopieren** die Anweisungen aus dem folgenden Code 2.1.

```
string tempKette;
//den Text aus dem Ziel in Sicherheit bringen
//Content muss in den Typ string konvertiert werden
tempKette = labelZiel.Content.ToString();
//den Text aus der Quelle in das Ziel kopieren
labelZiel.Content = labelQuelle.Content;
//den gesicherten Text in die Quelle kopieren
labelQuelle.Content = tempKette;
```

Code 2.1: Der Quelltext für die Methode `ButtonKopieren_Click()`

Genau wie bei den Windows Forms-Anwendungen sprechen Sie die Steuerelemente über ihren Namen an und greifen dann auf die Eigenschaften beziehungsweise Methoden zu. Dabei gibt es keine Besonderheiten. Sie müssen lediglich daran denken, dass die Eigenschaften zum Teil andere Namen haben – zum Beispiel `Content` für den Text in den Labels. Außerdem stimmt der Typ auch nicht immer mit den Typen aus den Windows Forms-Anwendungen überein. Die Eigenschaft `Content` für ein WPF-Steuerelement ist zum Beispiel vom Typ `object` und muss daher bei der ersten Zuweisung in eine Zeichenkette konvertiert werden.

Speichern Sie bitte die Änderungen und testen Sie das Programm.

Im nächsten Schritt wollen wir jetzt das Kopieren auch beim Doppelklicken auf eins der Labels durchführen lassen. Das haben wir in der Windows Forms-Anwendung im Ereignis **DoubleClick** der beiden Labels erledigt. Und hier gibt es jetzt bei der WPF-Anwendung die erste Hürde: Bei einem Doppelklick auf eins der Labels im Designer passiert nichts – und auch in der Liste der Ereignisse im Eigenschaftenfenster gibt es das Ereignis nicht. Wenn Sie die Liste der Ereignisse allerdings ein wenig verschieben, finden Sie eine gute Alternative für das Ereignis **DoubleClick** – nämlich das Ereignis **MouseDoubleClick**. Es wird beim Doppelklicken mit der Maus ausgelöst.

Legen Sie jetzt bitte für das Ereignis **MouseDoubleClick** des Labels `labelQuelle` eine Methode an und rufen Sie in dieser Methode die Methode `ButtonKopieren_Click()` auf. Denken Sie dabei bitte daran, dass Sie die Argumente `sender` und `e` übergeben müssen.

Anschließend sorgen wir dafür, dass auch bei einem Doppelklick auf das zweite Label die Texte kopiert werden. Dazu weisen Sie dem Ereignis **MouseDoubleClick** des Labels `labelZiel` die Methode für einen Doppelklick auf das Label `labelQuelle` zu. Das geht am einfachsten, wenn Sie den Namen der Methode in das Feld bei den Ereignissen eintragen. Denken Sie dabei bitte daran, dass Sie keine runden Klammern angeben dürfen, sondern nur den Namen der Methode.

Tipp:

Um Tippfehler zu vermeiden, kopieren Sie den Namen am besten aus dem Feld des anderen Ereignisses.

Legen Sie jetzt noch ein Kombinationsfeld und ein Listenfeld an. Dazu verwenden Sie wie von den Windows Forms-Anwendungen gewohnt die Steuerelemente **ComboBox** beziehungsweise **ListBox**. Die Einträge für die beiden Felder legen Sie über die Eigenschaft **Items** fest. Nach dem Anklicken des Symbols  erscheint allerdings nicht der Zeichenfolgen-Editor, sondern der Sammlungs-Editor.

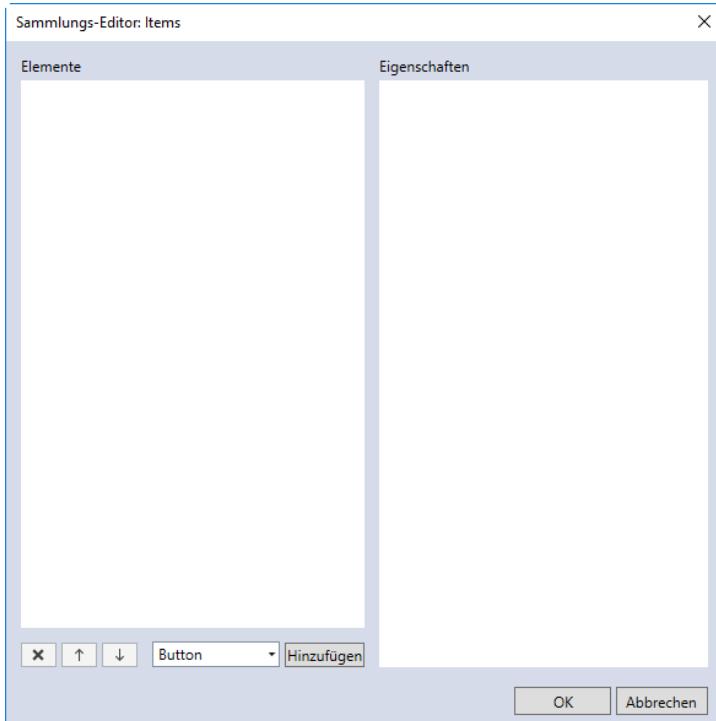


Abb. 2.1: Der Sammlungs-Editor

In diesem Editor können Sie über das Kombinationsfeld unten links die gewünschte Art des Eintrags auswählen und anschließend rechts im Bereich **Eigenschaften** weitere Einstellungen für den Eintrag vornehmen. Den Text eines Eintrags legen Sie dabei wieder über die Eigenschaft **Content** fest.

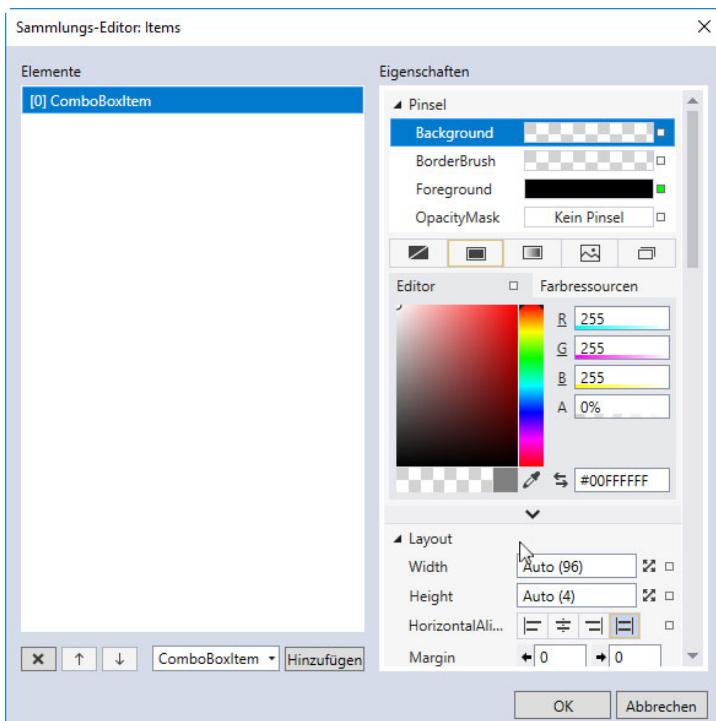


Abb. 2.2: Ein Eintrag im Sammlungs-Editor

Hinweis:

In der Standardeinstellung schlägt Ihnen der Auflistungs-Editor einen **Button** für einen Eintrag vor. Das übliche Element für ein Kombinationsfeld ist aber ein **ComboBoxItem**. Für ein Listenfeld verwenden Sie ein **ListBoxItem**.

Erstellen Sie jetzt einige Einträge für das Kombinationsfeld und das Listenfeld. Sorgen Sie dann dafür, dass der jeweils erste Eintrag direkt angezeigt wird. Dazu setzen Sie die Eigenschaft **SelectedIndex** in der Gruppe **Allgemein** jeweils auf 0.

Das Fenster der Anwendung sollte nun ungefähr so aussehen:

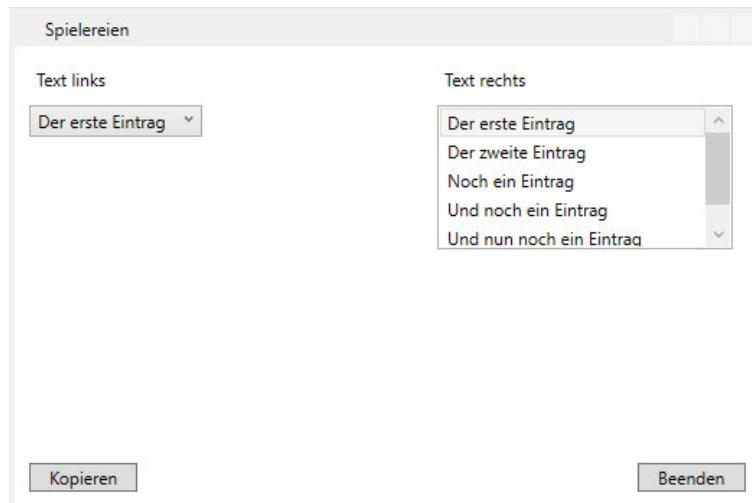


Abb. 2.3: Ein Kombinations- und ein Listenfeld

Um die ausgewählten Einträge in die Labels zu kopieren, benutzen wir das Ereignis **SelectionChanged**. Es tritt sowohl beim Kombinationsfeld als auch beim Listenfeld ein, wenn ein anderer Wert ausgewählt wird. Da es sich bei beiden Feldern um das Standardereignis handelt, können Sie den entsprechenden Ereignishandler ganz einfach durch einen Doppelklick erstellen lassen.

Die Methoden selbst könnten dann so aussehen wie im folgenden Code 2.2:

```
private void ComboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    //hier muss der Umweg über ein Item gegangen werden
    ComboBoxItem tempItem = (ComboBoxItem)comboBox.SelectedItem;
    labelQuelle.Content = tempItem.Content;
}
private void ListBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    //hier muss der Umweg über ein Item der Liste
    //gegangen werden
    ListBoxItem tempItem = (ListBoxItem)listBox.SelectedItem;
    labelZiel.Content = tempItem.Content;
}
```

Code 2.2: Die Methoden `ComboBox_SelectionChanged()` und `ListBox_SelectionChanged()`

Bitte beachten Sie:

Sie müssen beim Doppelklick auf die **ListBox** darauf achten, dass Sie nicht auf einen Eintrag doppelklicken. Denn dann legen Sie einen Ereignishandler für die Auswahl dieses einen Eintrags an. Zielen Sie zur Sicherheit beim Doppelklick auf den Rahmen. Wenn Sie ganz sichergehen wollen, können Sie die Methoden für die Ereignisse über den Bereich **Ereignisse** im Eigenschaftenfenster festlegen.

Da ein direkter Zugriff auf die Eigenschaft `SelectedItem.ToString()` bei der WPF für Listen- und Kombinationsfeld immer auch den vollständigen Typ des Steuerelements liefert und nicht nur den Text des Eintrags, müssen wir einen kleinen Umweg über einen einzelnen Eintrag eines Listen- beziehungsweise Kombinationsfelds gehen. Denn für diesen einzelnen Eintrag können Sie gezielt über die Eigenschaft `Content` auch nur den Text abrufen. Da die Eigenschaft `SelectedItem` sowohl für ein Listenfeld als auch für ein Kombinationsfeld den Typ `object` liefert, müssen Sie das Ergebnis bei der Zuweisung in den passenden Typ umwandeln.

Tipp:

Beim Steuerelement **ComboBox** können Sie den aktuell ausgewählten Eintrag auch über die Eigenschaft `Text` abrufen.

Jetzt fehlt nur noch das Kontrollkästchen zum Ein- und Ausblenden des Kombinations- und Listenfelds. Sie finden es wie bei den Windows Forms-Anwendungen in der Tool-box unter dem Eintrag **CheckBox**. Fügen Sie es bitte in das Fenster ein, ändern Sie den Text in Einblenden und setzen Sie die Eigenschaft `.IsChecked` in der Gruppe **Allgemein** auf `True`. Das Ein- beziehungsweise Ausblenden der beiden Felder erledigen wir im Ereignis **Click** des Kontrollkästchens.

Die komplette Methode für das Klicken auf die **CheckBox** sieht dann so aus:

```
private void CheckBox1_Click(object sender, RoutedEventArgs e)
{
    //ist das Kontrollkästchen markiert?
    //dann die Listen einblenden
    if (checkBox1.IsChecked == true)
    {
        listBox.Visibility = Visibility.Visible;
        comboBox.Visibility = Visibility.Visible;
    }
    //sonst ausblenden
    else
    {
        listBox.Visibility = Visibility.Hidden;
        comboBox.Visibility = Visibility.Hidden;
    }
}
```

Code 2.3: Die Methode `CheckBox1_Click()`

Bitte beachten Sie, dass die Sichtbarkeit von Steuerelementen bei der WPF über die Eigenschaft `Visibility` gesetzt wird. Um ein Element zu verstecken, weisen Sie dieser Eigenschaft den Wert `Visibility.Hidden` zu, und um ein Element anzuzeigen, benutzen Sie den Wert `Visibility.Visible`.

Übernehmen Sie jetzt die Änderungen. Speichern Sie dann das Projekt und probieren Sie es aus.

Zusammenfassung

WPF- und Windows Forms-Steuerelemente unterscheiden sich zum Teil erheblich – auch wenn sie denselben Namen haben.

Die Einträge für Kombinations- und Listenfelder legen Sie mit dem Sammlungs-Editor fest.

Die Eigenschaft `SelectedItem.ToString()` liefert Ihnen bei Listen- und Kombinationsfeldern der WPF neben dem Text des Eintrags auch immer den vollständigen Typ des Steuerelements.

Die Sichtbarkeit von WPF-Steuerelementen wird über die Eigenschaft `Visibility` gesetzt.

Aufgaben zur Selbstüberprüfung

- 2.1 Mit welcher Eigenschaft greifen Sie auf den Text in einem **Label** der WPF zu? Worauf müssen Sie dabei achten, wenn Sie diesen Text in eine Variable vom Typ `string` kopieren wollen?

- 2.2 Wie können Sie den aktuellen Eintrag in einem Kombinationsfeld der WPF ermitteln?

3 Besonderheiten der WPF

In diesem Kapitel beschäftigen wir uns mit einigen Besonderheiten der Windows Presentation Foundation im Vergleich zu Windows Forms. Dabei konzentrieren wir uns vor allem auf den Einsatz von Containern und Layouts, die Darstellung von Grafiken und die Ereignisverarbeitung.

Beginnen wir mit den Containern und Layouts.

3.1 Container und Layouts

Wie Sie ja bereits wissen, darf es bei einer WPF-Anwendung direkt unterhalb des Stammelements nur ein einziges weiteres Element geben. Dabei handelt es sich normalerweise um einen Container. Er nimmt nicht nur weitere Steuerelemente wie Schaltflächen, Labels und so weiter auf, sondern bestimmt auch, wie diese Steuerelemente angeordnet werden.

Wenn Sie mehr als ein Steuerelement in einer WPF-Anwendung benötigen, müssen Sie zwingend einen Container verwenden. Bei Windows Forms-Anwendungen dagegen können Sie beliebig viele Steuerelemente auch direkt im Formular ablegen.



In der Standardeinstellung arbeitet der Designer von Visual Studio bei einer WPF-Anwendung automatisch mit einem Container vom Typ **Grid**. Diesen Container können Sie löschen, indem Sie das Steuerelement markieren und dann die Taste **Entf** drücken. Bitte achten Sie dabei aber darauf, dass beim Löschen des Containers auch sämtliche Steuerelemente im Container ohne Rückfrage gelöscht werden.

Die anderen Container können Sie über die Toolbox einfügen. Dabei gibt es keine Besonderheiten.

Schauen wir uns jetzt einige wichtige Container und Layouts der Reihe nach an. Beginnen wir mit dem **StackPanel**, das Sie bereits eingesetzt haben.

3.1.1 Das StackPanel

Beim **StackPanel** werden die Steuerelemente entweder von oben nach unten oder von links nach rechts „gestapelt“. Die Reihenfolge der Steuerelemente hängt dabei in der Standardeinstellung davon ab, in welcher Reihenfolge Sie die Steuerelemente hinzufügen. Das zuerst hinzugefügte Steuerelement steht ganz oben beziehungsweise ganz links und das zuletzt hinzugefügte Steuerelement ganz unten beziehungsweise ganz rechts. Ohne Größenangaben wird die Breite beziehungsweise Höhe der Steuerelemente automatisch an die Breite beziehungsweise Höhe des StackPanels angepasst.

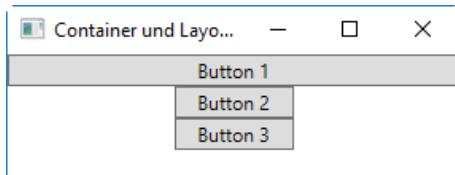


Abb. 3.1: Ein **StackPanel** (die obere Schaltfläche hat keine eigene Größenangabe)

Über die Eigenschaft **Orientation** in der Gruppe **Layout** können Sie die Ausrichtung des StackPanels festlegen. Der Wert `Horizontal` sorgt für eine horizontale Ausrichtung, der Wert `Vertical` für eine vertikale Ausrichtung. Die vertikale Ausrichtung ist auch die Standardeinstellung.

Bei der horizontalen Ausrichtung können Sie die Richtung auch umdrehen – und zwar, indem Sie die Eigenschaft **FlowDirection** in der Gruppe **Layout** auf den Wert `RightToLeft` setzen. Dann werden die Steuerelemente von rechts nach links angeordnet.

Die Eigenschaft **FlowDirection** wird aber erst angezeigt, wenn Sie die erweiterten Eigenschaften für die Gruppe im Eigenschaftenfenster einblenden lassen. Dazu klicken Sie einmal auf das Symbol mit der Pfeilspitze nach unten, das am Ende der Gruppe angezeigt wird. Sie können aber auch über das Feld oberhalb der Gruppen nach der Eigenschaft suchen.

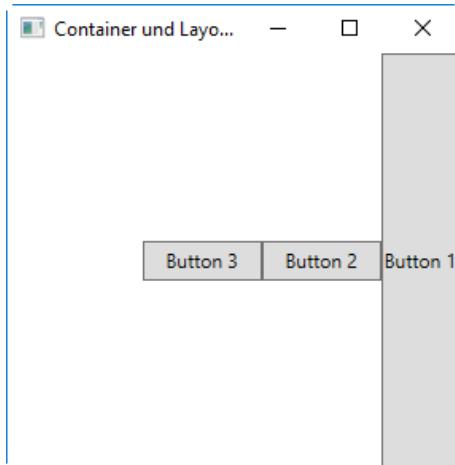


Abb. 3.2: Ein **StackPanel** in horizontaler Ausrichtung von rechts nach links
(die rechte Schaltfläche hat keine eigene Größenangabe)

3.1.2 Das WrapPanel

Das **WrapPanel**² arbeitet sehr ähnlich wie das **StackPanel**. Allerdings erfolgt automatisch ein Umbruch, wenn der Platz nicht mehr für weitere Steuerelemente ausreicht.

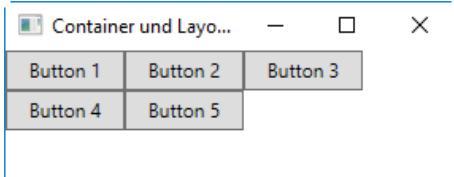


Abb. 3.3: Ein **WrapPanel**

Über die Eigenschaft **Orientation** legen Sie die Ausrichtung im Panel fest. Genau wie beim **StackPanel** lässt sich die Anordnung bei vertikaler Ausrichtung über die Eigenschaft **FlowDirection** umdrehen.

3.1.3 Das DockPanel

Das **DockPanel** arbeitet mit vier Bereichen – oben, unten, links und rechts. Wie ein Steuerelement angeordnet wird, legen Sie über die Eigenschaft **Dock** fest. Sie wird allerdings nicht beim Panel angezeigt, sondern bei den Steuerelementen, die Sie im **DockPanel** ablegen – und zwar in der Gruppe **Layout**. In der Standardeinstellung werden die Steuerelemente im linken Bereich abgelegt.

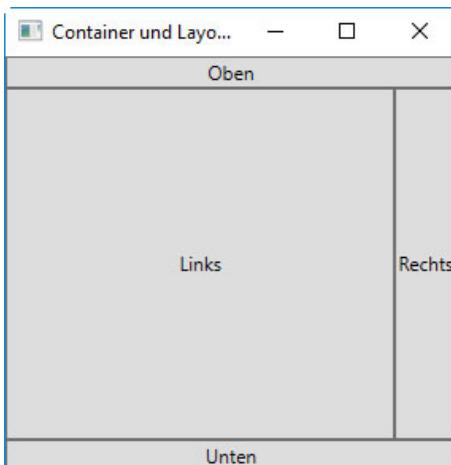


Abb. 3.4: Das **DockPanel** (für die Schaltflächen wurde keine Breite oder Höhe angegeben)

3.1.4 Das Grid

Beim **Grid** werden die Steuerelemente in einer Tabelle angeordnet. Die Einstellungen für die Zeilen beziehungsweise Spalten nehmen Sie über die Eigenschaften **RowDefinitions** beziehungsweise **ColumnDefinitions** in der Gruppe **Layout** vor. Diese Eigenschaften werden auch erst dann angezeigt, wenn Sie die erweiterten Eigenschaften einblenden.

2. *Wrap* bedeutet übersetzt so viel wie „Hülle“ oder „Decke“.

Das **Grid** ist das Standardlayout des WPF-Designers von Visual Studio. Allerdings wird beim Standardlayout nur eine einzige Zelle verwendet.

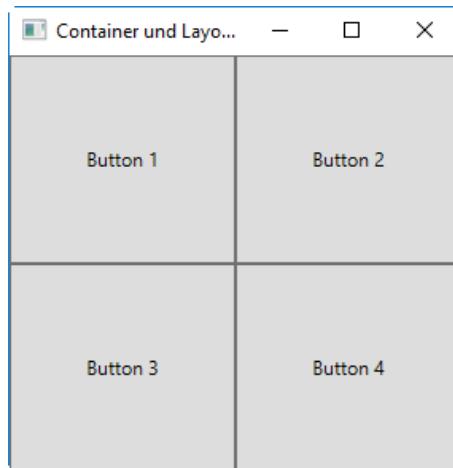


Abb. 3.5: Ein Grid mit zwei Spalten und zwei Zeilen

Neben dem **Grid**, bei dem Spalten und Zeilen nahezu beliebig konfiguriert werden können, gibt es auch noch das **UniformGrid**. Bei diesem Container sind die Zeilen und Spalten jeweils identisch.

Hinweis:

Die WPF kennt noch einige weitere Container, die wir Ihnen hier nicht im Detail vorstellen wollen. Außerdem können Sie für die Container noch zahlreiche weitere Einstellungen vornehmen und die Container auch ineinanderschachteln. Sehen Sie bei Interesse bitte in der Hilfe nach.

Kommen wir nun zu den Besonderheiten bei der Darstellung von Grafiken.

3.2 Darstellung von Grafiken

Bei den Windows Forms-Anwendungen bildet die Schnittstelle GDI+ die Grundlage für die Darstellung von Grafiken. Sie arbeitet im Wesentlichen pixelorientiert und über die Zeichenfläche `Graphics`. Zwar lassen sich auch mit GDI+ durchaus ansprechende grafische Darstellungen erzielen, allerdings ist der Aufwand oft erheblich und die Geschwindigkeit auch nicht immer optimal.

Die WPF arbeitet daher mit vektorbasierten Grafiken, die sich ohne Qualitätsverlust nahezu beliebig verändern lassen. Die Grundlage bildet DirectX – eine Sammlung von Schnittstellen für die Programmierung von Multimedia-Anwendungen. Sie ist nicht nur deutlich schneller, sondern bietet gleichzeitig auch mehr Möglichkeiten.



Vektorbasierte Grafiken werden nicht durch einzelne Bildpunkte dargestellt, sondern durch mathematische Beschreibungen einzelner Objekte.

Grafische Objekte einer WPF-Anwendung können direkt über Steuerelemente im Fenster dargestellt werden und benötigen dann keine eigene Zeichenfläche mehr. Außerdem gibt es keinen Unterschied mehr zwischen den meisten Steuerelementen und grafischen

Objekten wie zum Beispiel einem Kreis oder einem Rechteck. Das heißt, Sie können zum Beispiel auch eine Schaltfläche ohne Qualitätsverluste in der Größe verändern und drehen.

Probieren Sie das einfach einmal aus. Erstellen Sie eine neue WPF-Anwendung. Fügen Sie eine Schaltfläche und eine Ellipse ein. Für die Ellipse benutzen Sie dabei das Steuer-element **Ellipse**. Lassen Sie dann beim Anklicken der Schaltfläche die folgenden Anweisungen ausführen:

```
//bitte in einer Zeile eingeben
RotateTransform rotieren = new RotateTransform(45.0,
50.0, 50.0);
ellipse1.RenderTransform = rotieren;
button1.RenderTransform = rotieren;
```

Code 3.1: Das Drehen eines Objekts

Hinweis:

Die Namen der Steuerelemente in dem Code hängen von den Namen ab, die Sie selbst vergeben haben.

Mit der Anweisung

```
RotateTransform rotieren = new RotateTransform(45.0,
50.0, 50.0);
```

wird ein Objekt `rotieren` der Klasse `RotateTransform` erzeugt. Über den Konstruktur wird dabei festgelegt, dass die Drehung um 45 Grad um den Punkt 50, 50 erfolgen soll.

Bitte beachten Sie:

Nahezu alle Größenangaben bei Grafiken der WPF erfolgen als Typ `double`.



Die beiden Anweisungen

```
ellipse1.RenderTransform = rotieren;
button1.RenderTransform = rotieren;
```

führen dann die Drehung für die Ellipse und die Schaltfläche durch. Dazu wird der Eigenschaft `RenderTransform` jeweils das Objekt `rotieren` zugewiesen.

Mit einer ähnlichen Technik können Sie Objekte auch stufenlos vergrößern und verkleinern.

Die Anweisungen

```
ScaleTransform skalieren = new ScaleTransform(1.5, 1.5);
button1.RenderTransform = skalieren;
```

vergrößern eine Schaltfläche zum Beispiel horizontal und vertikal um den Faktor 1,5.

Damit wollen wir unseren kurzen Ausflug in die grafische Gestaltung von WPF-Anwendungen auch schon wieder beenden. Ein etwas ausführlicheres Beispiel finden Sie im Projekt **WPFGrafik** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. Wir werden uns aber auch an anderer Stelle noch einmal intensiver mit dem Thema beschäftigen.

Schauen wir uns zum Schluss dieses Kapitels noch eine weitere Besonderheit an – das Weiterleiten von Ereignissen.

3.3 Die Weiterleitung von Ereignissen

Bei Windows Forms-Anwendungen werden Ereignisse in der Regel genau von dem Steuerelement verarbeitet, das auf sie reagiert. Beim Anklicken einer Schaltfläche wird zum Beispiel genau das Ereignis **Click** für diese Schaltfläche ausgelöst und auch verarbeitet. Für andere Steuerelemente bleibt das Ereignis normalerweise unsichtbar.

Bei WPF-Anwendungen dagegen können Sie Ereignisse auch weitergeben – sowohl an übergeordnete Steuerelemente als auch an untergeordnete Steuerelemente. Schauen wir uns dazu ein Beispiel an:

Im Fenster einer Anwendung befindet sich ein **Grid**, das ein Listenfeld und eine Schaltfläche enthält. Das Fenster – das **Window** – befindet sich also ganz oben in der Hierarchie der Steuerelemente, darunter folgt das **Grid** und auf der untersten Ebene das Listenfeld und die Schaltfläche.

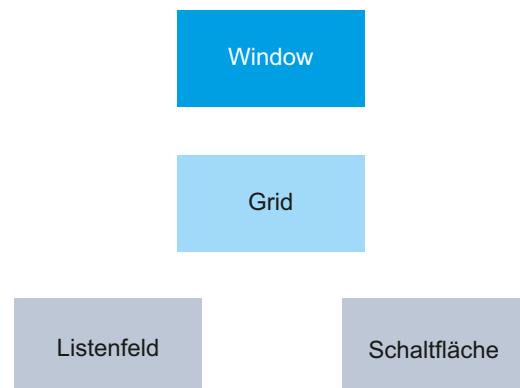


Abb. 3.6: Die Hierarchie der Steuerelemente

Auf das Anklicken der Schaltfläche können Sie nun bei der WPF auf drei Arten reagieren:

- 1) Sie verarbeiten das Ereignis ausschließlich für die Schaltfläche. Diese Variante unterscheidet sich nicht von dem Vorgehen bei Windows Forms-Anwendungen.
- 2) Sie reichen das Ereignis auch an das nächste Steuerelement weiter, das sich in der Hierarchie oberhalb der Schaltfläche befindet – also an das **Grid** und das **Window**.
- 3) Sie lassen das Ereignis in der anderen Richtung durch die Hierarchie laufen – also von oben nach unten. Das Ereignis wandert dabei also jeweils eine Ebene nach unten, bis es wieder beim eigentlichen Auslöser angekommen ist.

Weitergeleitete Ereignisse werden **Routed Events**^{a)} genannt. Die Ereignisse, die von unten nach oben weitergereicht werden, heißen auch **Bubbling Events**^{b)}. Die Ereignisse, die von oben nach unten laufen, heißen **Tunneling Events**.



- a) *Route* bedeutet übersetzt so viel wie „Weg, Strecke“.
- b) *To bubble* bedeutet übersetzt so viel wie „sprudeln“.

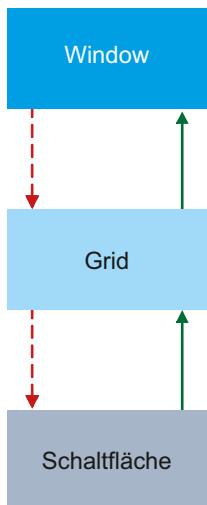


Abb. 3.7: Tunneling Events (die gestrichelte rote Linie links) und Bubbling Events (die durchgezogene grüne Linie rechts)

In unserem Beispiel könnten wir zum Beispiel die Ereignisse vom Fenster nach unten zur Schaltfläche durchlaufen lassen. Dazu legen Sie jeweils für die Steuerelemente eine Reaktion auf das Ereignis **PreviewMouseLeftButtonDown** fest.

Die entsprechenden Methoden könnten so aussehen:

```

private void Button1_PreviewMouseLeftButtonDown(object sender,
    MouseButtonEventArgs e)
{
    listBox1.Items.Add("Die Schaltfläche wurde angeklickt");
    listBox1.Items.Add("-----");
}

private void Grid_PreviewMouseLeftButtonDown(object sender,
    MouseButtonEventArgs e)
{
    listBox1.Items.Add("Das Grid wurde angeklickt");
    listBox1.Items.Add("-----");
}

private void Window_PreviewMouseLeftButtonDown(object sender,
    MouseButtonEventArgs e)
{
    listBox1.Items.Add("Das Fenster wurde angeklickt");
    listBox1.Items.Add("-----");
}
  
```

Code 3.2: Die Methoden für Tunneling Events

Die einzelnen Methoden nehmen jeweils einen Eintrag in dem Listenfeld vor.

Hinweis:

Bei getunnelten Ereignissen wird das Ereignis direkt weitergereicht – also nicht erst beim eigentlichen Steuerelement ausgelöst. In unserem Beispiel löst daher das Anklicken der Schaltfläche bei den getunnelten Ereignissen direkt das Ereignis für das Fenster aus.

Die Weitergabe der Ereignisse können Sie jederzeit auch unterbrechen. Dazu setzen Sie in der Methode des letzten Steuerelements, das auf das Ereignis reagieren soll, die Eigenschaft `e.Handled` auf `true`.



Im praktischen Einsatz finden Sie weitergeleitete Ereignisse im Projekt **WPFEVENT-Handling** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Zusammenfassung

Bei einer WPF-Anwendung benötigen Sie einen Container, wenn Sie mehr als ein Steuerelement verwenden. Der Container nimmt nicht nur die Steuerelemente auf, sondern bestimmt auch die Anordnung.

Es gibt unterschiedliche Container wie zum Beispiel das **StackPanel**, das **DockPanel** oder das **Grid**.

Die Grafikausgabe in WPF-Anwendungen basiert auf DirectX und arbeitet mit Vektorgrafiken.

Die grafischen Objekte werden direkt über Steuerelemente dargestellt und benötigen keine eigene Zeichenfläche mehr.

In einer WPF-Anwendung können Sie Ereignisse an unter- und übergeordnete Steuerelemente weitergeben.

Aufgaben zur Selbstüberprüfung

- 3.1 Mit welchem Standard-Container arbeitet der Designer von Visual Studio bei einer WPF-Anwendung?

- 3.2 Können Sie eine Schaltfläche in einem WPF-Fenster stufenlos vergrößern und verkleinern? Begründen Sie bitte kurz Ihre Antwort.

- 3.3 Wie wird die Weiterleitung von Ereignissen an übergeordnete Steuerelemente im Fachjargon genannt?

- 3.4 Womit beginnen Ereignisse, mit denen Sie auf getunnelte Ereignisse reagieren können, normalerweise?

- 3.5 Wie können Sie die Weiterleitung von Ereignissen unterbrechen?

4 Eine Textverarbeitung als WPF-Anwendung

In diesem Kapitel werden wir die Textverarbeitung, die wir bereits als Windows Forms-Anwendung umgesetzt haben, als WPF-Anwendung erstellen. Dabei konzentrieren wir uns vor allem wieder auf die Besonderheiten im Vergleich zur Windows Forms-Anwendung.

Beginnen wir mit einigen Vorüberlegungen.

4.1 Vorüberlegungen

Die Textverarbeitung soll wieder sowohl Funktionen zur Dateihandhabung wie Öffnen, Speichern und Schließen als auch Funktionen zum Bearbeiten und Formatieren der Texte zur Verfügung stellen. Der Aufruf dieser Funktionen erfolgt über ein Menüband. In diesem Menüband verwenden wir sowohl eine Symbolleiste für den Schnellzugriff, ein Anwendungsmenü als auch eine Registerkarte. Was sich genau hinter diesen Elementen verbirgt, erfahren Sie gleich, wenn wir das Menüband erstellen.

Die fertige Anwendung soll ungefähr so aussehen:

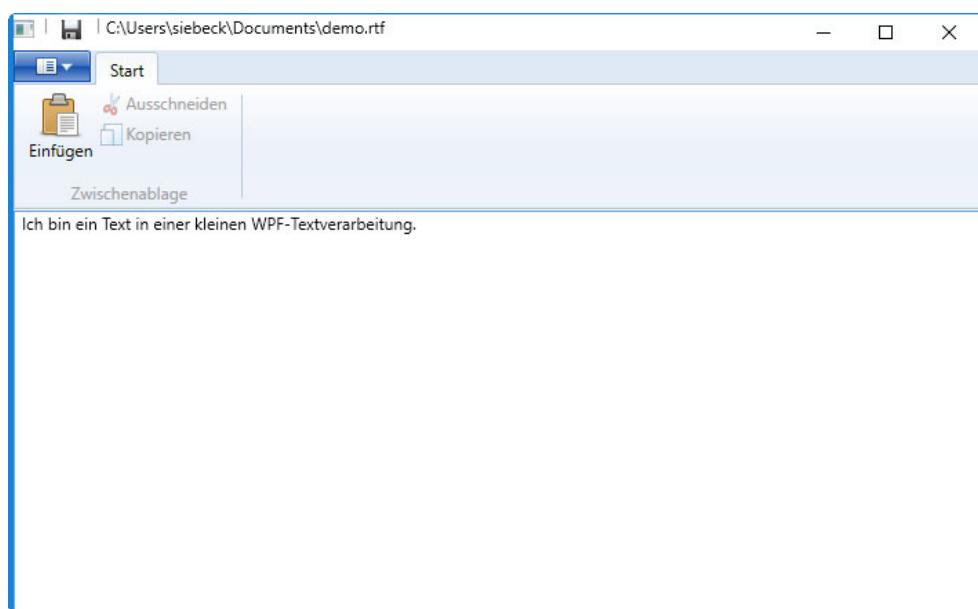


Abb. 4.1: Die WPF-Textverarbeitung

Bei der Dateihandhabung berücksichtigen wir wieder den Unterschied zwischen den Funktionen **Speichern unter** und **Speichern**. Das heißt, beim ersten Speichern wird der Dialog **Speichern unter** angezeigt. Alle weiteren Speichervorgänge benutzen dann die aktuellen Einstellungen – es sei denn, der Anwender speichert das Dokument über die Funktion **Speichern unter** im Anwendungsmenü. Um unterscheiden zu können, ob das Dokument bereits einmal gespeichert wurde, verwenden wir ein Feld `dateiname`, das zunächst eine leere Zeichenkette enthält. Beim Speichern weisen wir dem Feld dann den Dateinamen zu, den der Anwender im Dialog **Speichern unter** eingegeben hat.

Damit die Anwendung etwas komfortabler wird, zeigen wir den Dateinamen zusätzlich noch in der Titelleiste des Fensters an.

Für die Eingabe der Texte benutzen wir das WPF-Steuerelement **RichTextBox**.

Beim Laden eines Dokuments soll zunächst geprüft werden, ob es in dem alten Dokument nicht gesicherte Änderungen gibt. Wenn das der Fall ist, soll eine Sicherheitsabfrage erscheinen, die es dem Anwender ermöglicht, die Daten zu speichern. Da eine WPF-RichTextBox keine Eigenschaft `Modified` hat, setzen wir ein Feld `geaendert` auf `true`, sobald Änderungen am Inhalt der RichTextBox vorgenommen worden sind. Beim Speichern, Öffnen und Erstellen eines Dokuments setzen wir das Feld wieder zurück auf `false`.

Die fertige Textverarbeitung finden Sie im Projekt **WPFMinitext** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.



Legen wir jetzt das Fenster für die Anwendung an.

4.2 Das Fenster

Erstellen Sie bitte eine neue leere WPF-Anwendung. Setzen Sie die Eigenschaft **Title** für den Text in der Titelleiste des Fensters auf **WPF-MiniText** und lassen Sie das Fenster über die Eigenschaft **WindowState** in der Gruppe **Allgemein** maximiert darstellen.

Damit wir das Menüband nutzen können, sind einige Vorarbeiten nötig. Im ersten Schritt müssen Sie einen Verweis auf die Datei **System.Windows.Controls.Ribbon.dll** zum Projekt hinzufügen. Wählen Sie dazu die Funktion **Projekt/Verweis hinzufügen ...**. Im Fenster **Verweis-Manager – WPFMinitext** verschieben Sie dann die Anzeige in der mittleren Liste, bis der Eintrag **System.Windows.Controls.Ribbon** angezeigt wird.

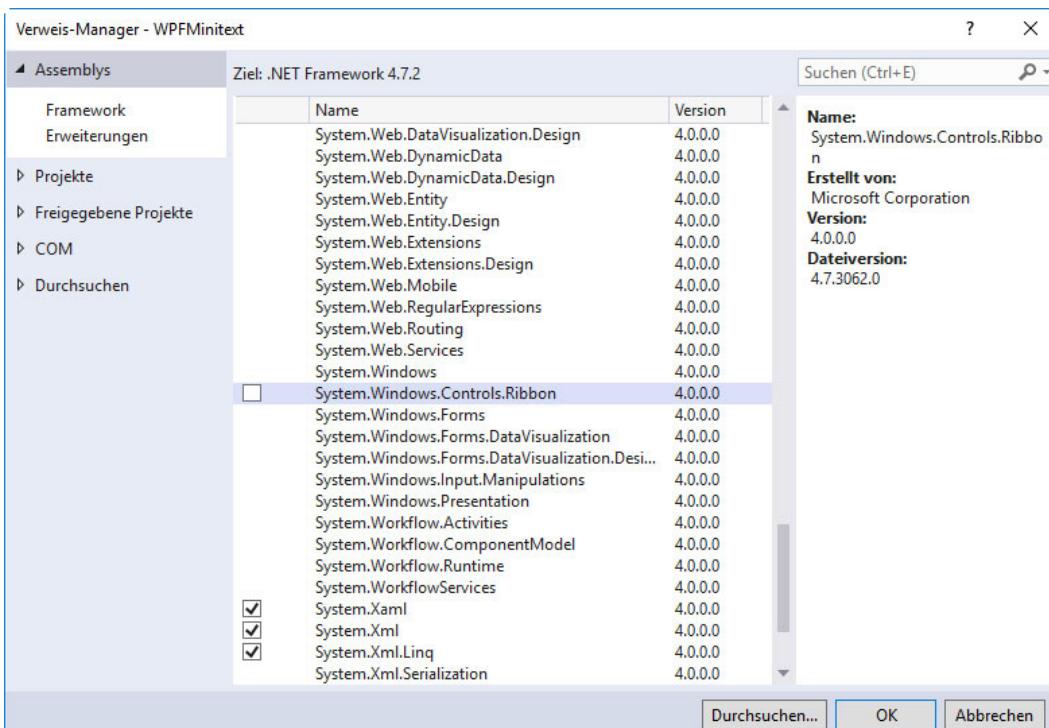


Abb. 4.2: Der Eintrag **System.Windows.Controls.Ribbon**

Markieren Sie das Kontrollkästchen links vor dem Eintrag und klicken Sie anschließend auf **OK**. Der neu hinzugefügte Verweis wird danach auch im Projektmappen-Explorer im Zweig **Verweise** angezeigt.

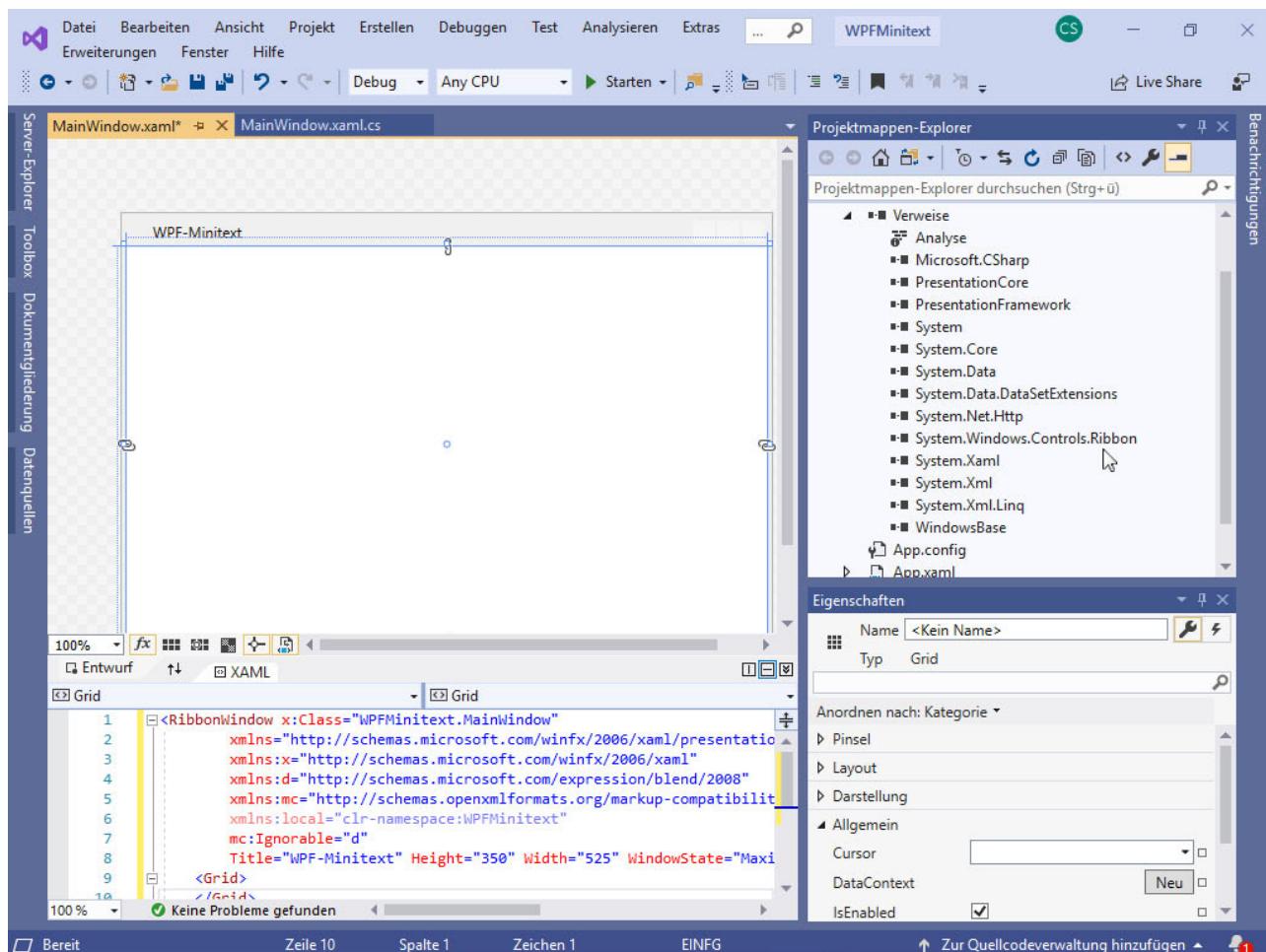


Abb. 4.3: Der eingefügte Verweis (oben rechts im Projektmappen-Explorer am Mauszeiger)

Damit die Symbolleiste für den Schnellzugriff gleich optimal angezeigt wird, benutzen wir für das Fenster der Anwendung nicht mehr die Klasse `Window`, sondern die Klasse `RibbonWindow`³. Sie fügt die Symbolleiste für den Schnellzugriff in die Titelleiste ein. Bei der Klasse `Window` dagegen wird die Symbolleiste für den Schnellzugriff unterhalb der Titelleiste angezeigt.

Ändern Sie das Start-Tag direkt zu Beginn des XAML-Codes von `Window` in `RibbonWindow`. Passen Sie auch das End-Tag für das Fenster entsprechend an, falls diese Änderung nicht automatisch durchgeführt wird.

Wechseln Sie dann in die Datei `MainWindow.xaml.cs`. Sorgen Sie dort dafür, dass unsere Klasse für das Fenster nicht mehr von `Window` abgeleitet wird, sondern von `RibbonWindow`. Ändern Sie dazu die Vereinbarung der Klasse von

```
public partial class MainWindow : Window
in
public partial class MainWindow : RibbonWindow
```

3. *Ribbon* bedeutet übersetzt „Band“.

Damit die Klasse `RibbonWindow` bekannt ist, müssen Sie noch den Namensraum `System.Windows.Controls.Ribbon` bekannt machen. Die entsprechende Anweisung sieht so aus:

```
using System.Windows.Controls.Ribbon;
```

Speichern Sie anschließend die Änderungen und wechseln Sie wieder in den Designer. Fügen Sie dort im XAML-Code das Tag `<Ribbon>` zwischen den Tags für das Grid ein. Das End-Tag sollte automatisch vom Editor ergänzt werden. Nach dem Einfügen der Tags erscheint auch das Menüband im Fenster.

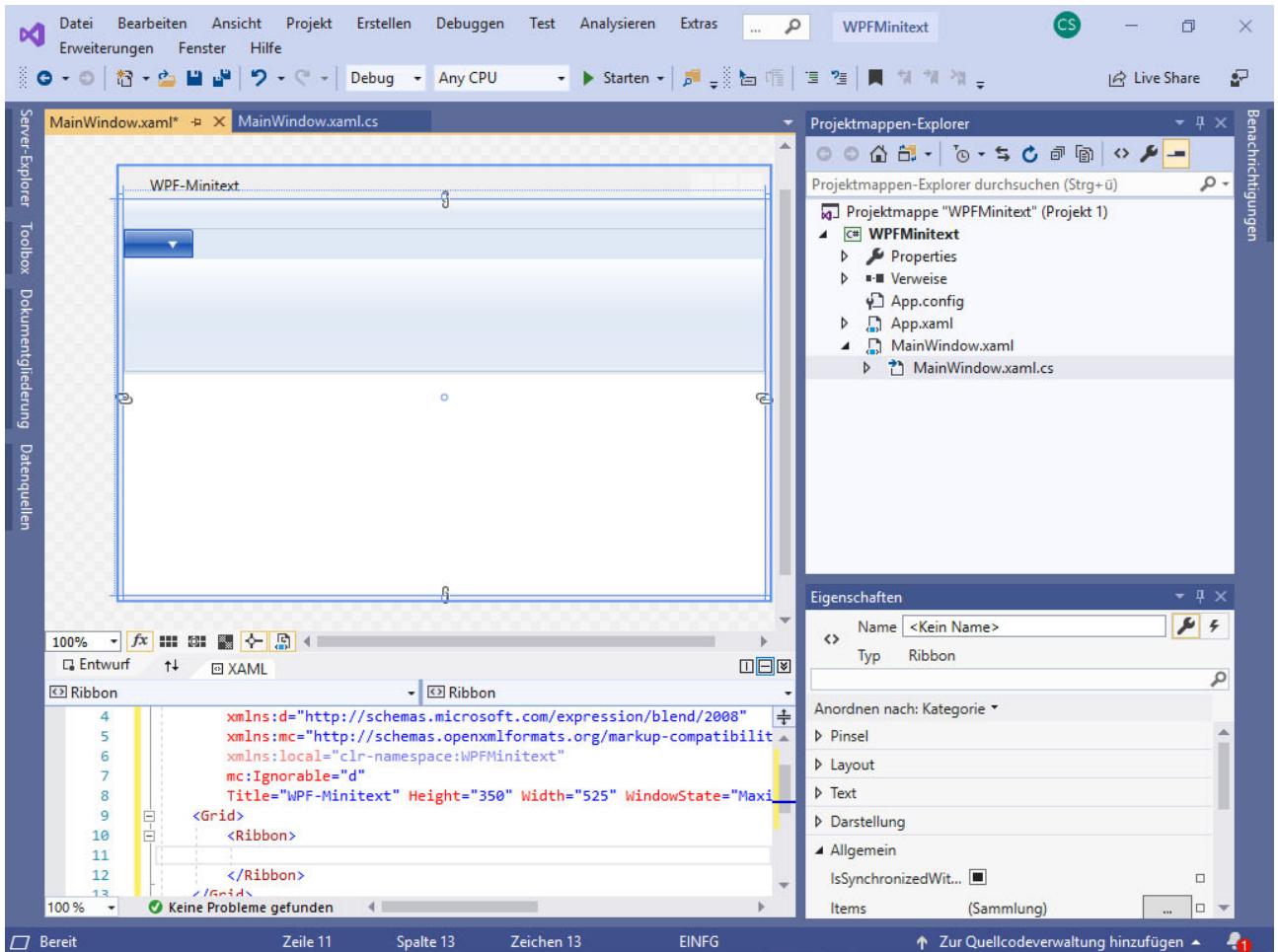


Abb. 4.4: Das Menüband im Fenster

Hinweis:

Sie können das Menüband nur direkt über Tags einfügen. In der Toolbox wird es nicht angezeigt.

Bitte beachten Sie:

Gehen Sie beim Arbeiten in der XAML-Ansicht sehr sorgfältig vor. Achten Sie genau darauf, dass Sie nicht versehentlich Texte überschreiben. Prüfen Sie auch sorgfältig, ob es für jedes Start-Tag ein entsprechendes End-Tag gibt. Speichern Sie vor allem bei den ersten Versuchen zur Sicherheit das komplette Projekt, bevor Sie Änderungen in der XAML-Ansicht vornehmen.



Fügen Sie dann noch ein RichTextBox-Steuerelement aus der Toolbox ein. Sie finden es in der Gruppe **Alle WPF-Steuerelemente**. Achten Sie beim Einfügen bitte darauf, dass Sie das Steuerelement nicht im Menüband ablegen, sondern im Grid – also im Container des Fensters. Dazu müssen Sie das Menüband unter Umständen verkleinern. Alternativ können Sie den XAML-Code der **RichTextBox** auch im XAML-Editor nach dem Einfügen an die richtige Stelle kopieren – nämlich vor das schließende Tag des Grids.

Positionieren Sie die **RichTextBox** dann so, dass sie links im Fenster unterhalb des Menübands angezeigt wird. Prüfen Sie anschließend, ob die Werte im ersten, zweiten und vierten Feld der Eigenschaft **Margin**⁴ in der Gruppe **Layout** jeweils auf 0 gesetzt sind. Dadurch werden die Abstände vom linken, rechten und unteren Rand festgelegt.

Damit das Steuerelement immer in der Breite des Fensters angezeigt wird, ändern Sie die Einträge in den Feldern für die Eigenschaften **Width** und **Height** in der Gruppe **Layout** in **Auto**. Klicken Sie dazu auf das Symbol **Auf Auto festlegen**  hinter dem Feld. Setzen Sie dann gegebenenfalls noch die Eigenschaften **HorizontalAlignment** und **VerticalAlignment**⁵ auf **Stretch**. Prüfen Sie dazu, ob jeweils das rechte Symbol bei den Eigenschaften markiert ist. Wenn das nicht der Fall ist, klicken Sie es bitte an.

Das Fenster sollte nun ungefähr so aussehen:

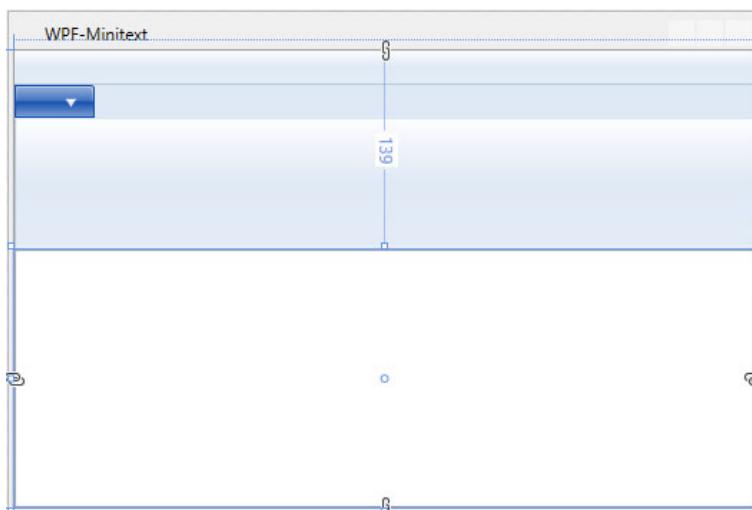


Abb. 4.5: Die Rohform des Fensters

4.3 Das Menüband

Im nächsten Schritt werden wir jetzt das Menüband mit Leben füllen. Dabei müssen wir selbst Hand anlegen und die verschiedenen Elemente über die XAML-Ansicht einfügen.

Beginnen wir mit der Symbolleiste für den Schnellzugriff. Sie wird oben links in der Titelleiste angezeigt und enthält zum Beispiel Symbole zum Speichern, zum Rückgängigmachen und zum Wiederholen. Das Erstellen erfolgt über die Tags `Ribbon.QuickAccessToolbar` und `RibbonQuickAccessToolBar`. Das Tag `RibbonQuickAccessToolbar` muss dabei innerhalb des Tags `Ribbon`.

4. Margin bedeutet übersetzt so viel wie „Rand“.

5. HorizontalAlignment bedeutet übersetzt so viel wie „horizontale Ausrichtung“ und VerticalAlignment bedeutet „vertikale Ausrichtung“.

QuickAccessToolbar liegen. Im Tag RibbonQuickAccessToolbar können Sie dann weitere Tags für die Symbole erstellen – zum Beispiel über RibbonButton für eine einfache Schaltfläche.

Stellen Sie die Einfügemarken in der XAML-Ansicht zwischen die Tags für das Menüband. Sie lauten <Ribbon> beziehungsweise </Ribbon>.

Tipp:

Wenn Sie die Zeile in der XAML-Ansicht nicht finden, klicken Sie einmal mit der Maus auf das Steuerelement im Fenster. Danach wird der Beginn der dazugehörigen Zeile in der XAML-Ansicht markiert.

Legen Sie dann gegebenenfalls eine neue Zeile an und fügen Sie ein Start-Tag <Ribbon.QuickAccessToolBar> ein. Das End-Tag wird vom Editor automatisch ergänzt. Fügen Sie zwischen dem Start-Tag und diesem End-Tag ein Start-Tag <RibbonQuickAccessToolBar> ein. Auch hier ergänzt der Editor wieder das End-Tag automatisch.

Da sich über die Eigenschaften der Symbolleiste für den Schnellzugriff leider keine Symbole anlegen lassen, müssen wir auch hier selbst Hand anlegen. Ergänzen Sie ein Tag <RibbonButton> und prüfen Sie, ob das dazugehörige End-Tag angelegt wird.

Die kompletten Zeilen mit der Anweisung für das Menüband in der XAML-Ansicht sollten ungefähr so aussehen:

```
<Ribbon>
  <Ribbon.QuickAccessToolBar>
    <RibbonQuickAccessToolBar>
      <RibbonButton></RibbonButton>
    </RibbonQuickAccessToolBar>
  </Ribbon.QuickAccessToolBar>
</Ribbon>
```

Code 4.1: Die XAML-Anweisungen für das Symbol in der Symbolleiste für den Schnellzugriff

Bitte beachten Sie:

Überprüfen Sie bei der Eingabe unbedingt die korrekte Schachtelung. Das Symbol wird mit den beiden Tags <RibbonButton> und </RibbonButton> vereinbart. Diese beiden Tags müssen zwischen den Tags <RibbonQuickAccessToolBar> und </RibbonQuickAccessToolBar> liegen. Diese Tags wiederum liegen zwischen den Tags <Ribbon.QuickAccessToolBar> und </Ribbon.QuickAccessToolBar>, die wiederum zwischen den Tags <Ribbon> und </Ribbon> liegen.



Über die Eigenschaft **SmallImageSource**⁶ in der Gruppe **Sonstiges** im Eigenschaftenfenster können Sie auch ein Symbol für den **RibbonButton** festlegen. Dazu müssen Sie allerdings den gesamten Pfad zur Datei per Hand eintragen. Etwas einfacher und auch übersichtlicher wird es, wenn Sie die Grafikdateien in einem eigenen Ordner in Ihrem Projekt ablegen.

6. SmallImageSource bedeutet übersetzt so viel wie „Quelle für kleines Bild“.

Markieren Sie dazu den Eintrag des Projekts im Projektmappen-Explorer. Das ist in der Regel der Eintrag direkt unterhalb des Eintrags für die Projektmappe. Wählen Sie dann im Menü **Projekt** die Funktion **Neuer Ordner**.

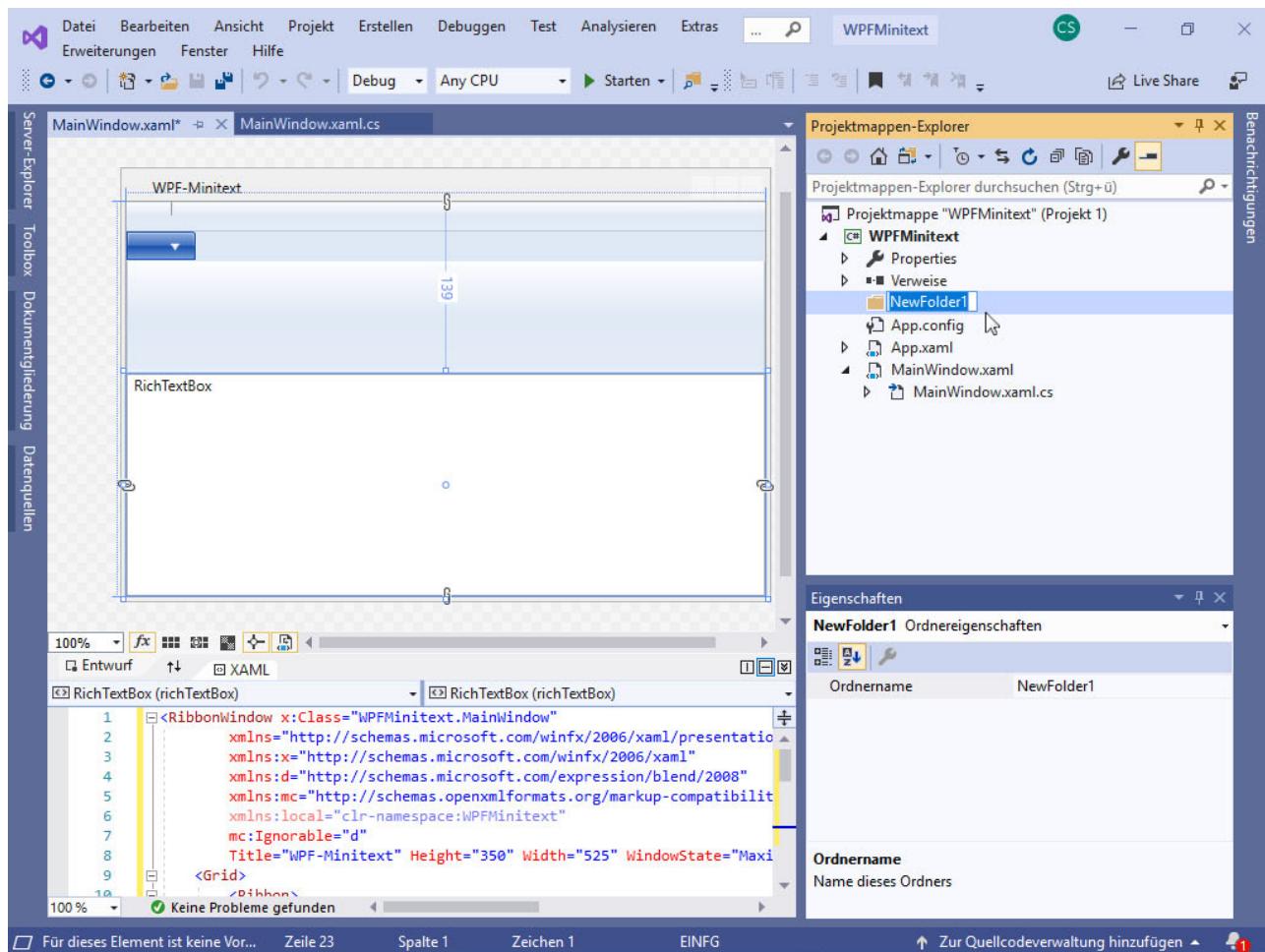


Abb. 4.6: Der neue Ordner im Projektmappen-Explorer
(der Mauszeiger steht auf dem Eintrag des neuen Ordners)

Geben Sie anschließend im Projektmappen-Explorer einen Namen für den Ordner ein. Wir benutzen in unserem Beispiel den Namen **symbole**. In den neuen Ordner können Sie jetzt entweder über den Explorer von Windows oder direkt über die Zwischenablage Dateien für die Symbole kopieren. Einige Beispiele im Format PNG finden Sie im Ordner **\cshp14d_icons** bei den Beispielen im heftbezogenen Downloadbereich Ihrer Online-Lernplattform.

Hinweis:

Falls das Hinzufügen nicht funktioniert, benutzen Sie die Funktion **Hinzufügen/Vorhandenes Element ...** im Kontext-Menü des Ordners. Wählen Sie dann die Symbole im folgenden Dialog aus.

Nach dem Kopieren können Sie die Symbole dann über das Kombinationsfeld für die Eigenschaft **SmallImageSource** zuweisen.

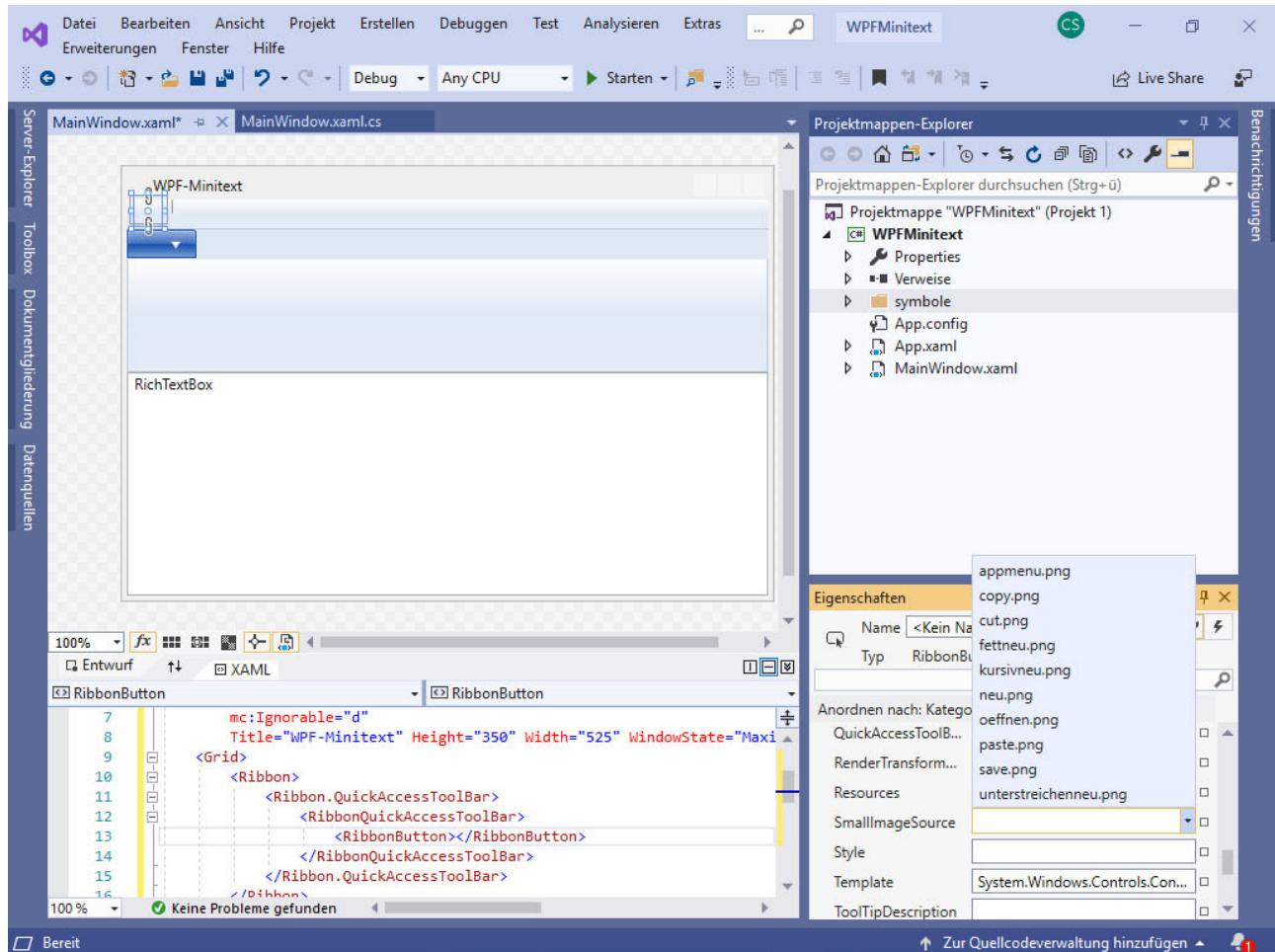


Abb. 4.7: Die Liste der Grafikdateien für die Eigenschaft **SmallImageSource** (unten rechts im Eigenschaftenfenster am Mauszeiger)

In unserem Beispiel wollen wir eine Schaltfläche zum Speichern einfügen. Weisen Sie das Symbol `save.png` zu.

So viel zur Symbolleiste für den Schnellzugriff. Wenn Sie Lust haben, können Sie ja noch weitere Symbole hinzufügen – zum Beispiel zum Rückgängigmachen oder zum Wiederholen.

Im nächsten Schritt ergänzen wir ein Anwendungsmenü. Es wird immer links im Menüband angezeigt und enthält zum Beispiel Befehle zum Speichern, Öffnen und Erstellen von Dokumenten sowie einen Befehl zum Beenden der Anwendung.

Das Anwendungsmenü wird durch die geschachtelten Tags `<Ribbon.ApplicationMenu>` und `<RibbonApplicationMenuItem>` erstellt. Über die Eigenschaft **SmallImageSource** für das Menü können Sie dabei ein Symbol festlegen.

Einen Eintrag in dem Menü erstellen Sie über das Tag `<RibbonApplicationMenuItem>`. Über die Eigenschaft **Header**⁷ in der Gruppe **Allgemein** des Eigenschaftenfensters legen Sie den Text für den Eintrag fest und über die Eigenschaft **ImageSource** in der Gruppe **Sonstiges** ein Symbol.

7. *Header* bedeutet übersetzt so viel wie „Kopf, Kopfzeile“.

Die XAML-Anweisungen für ein Anwendungsmenü mit den Einträgen **Neu**, **Öffnen**, **Speichern**, **Speichern unter** und **Beenden** könnten zum Beispiel so aussehen:

```
<Ribbon.ApplicationMenu>
    <RibbonApplicationMenu SmallImageSource="symbole/
    appmenu.png">
        <RibbonApplicationMenuItem Header="Neu"
        ImageSource="symbole/neu.png"></RibbonApplicationMenuItem>
        <RibbonApplicationMenuItem Header="Öffnen"
        ImageSource="symbole/oeffnen.png">
        </RibbonApplicationMenuItem>
        <RibbonApplicationMenuItem Header="Speichern"
        ImageSource="symbole/save.png">
        </RibbonApplicationMenuItem>
        <RibbonApplicationMenuItem Header="Speichern unter">
        </RibbonApplicationMenuItem>
        <RibbonApplicationMenuItem Header="Beenden">
        </RibbonApplicationMenuItem>
    </RibbonApplicationMenu>
</Ribbon.ApplicationMenu>
```

Code 4.2: Die XAML-Anweisungen für das Anwendungsmenü

Hinweis:

Achten Sie auch hier sehr sorgfältig auf die korrekte Schachtelung.

Symbole für die Einträge finden Sie ebenfalls im Ordner **\icons** bei den Beispielen auf Ihrer Online-Lernplattform. Die Symbole stammen zum Teil aus Windows-Anwendungen und zum Teil aus der kostenlosen Icon-Bibliothek Open Icon Library. Sie finden diese Bibliothek im Internet auf der Seite <https://sourceforge.net/projects/openiconlibrary/>.

Damit Sie die Menüeinträge eindeutig unterscheiden können, sollten Sie ihnen noch sprechende Namen geben. Wir benutzen in unserem Beispiel die Namen `menuNeu`, `menuOeffnen`, `menuSpeichern`, `menuSpeichernUnter` und `menuBeenden`.

Kommen wir jetzt zu den Registern und Gruppen im Menüband. Die Register erzeugen Sie über das Tag `<RibbonTab>`. In diesem Tag legen Sie dann über das Tag `<RibbonGroup>` die Gruppen an. Die Beschriftung für die Register und auch die Gruppen nehmen Sie über die Eigenschaft **Header** in der Gruppe **Allgemein** des Eigenschaftenfensters vor.

Für die Elemente in einer Gruppe können Sie dann zum Beispiel wieder das Tag `<RibbonButton>` für eine Schaltfläche verwenden.

Die XAML-Anweisungen für ein Register **Start** mit der Gruppe **Zwischenablage** und drei Symbolen sehen dann so aus:

```
<RibbonTab Header="Start">
    <RibbonGroup Header="Zwischenablage">
        <RibbonButton></RibbonButton>
        <RibbonButton></RibbonButton>
        <RibbonButton></RibbonButton>
    </RibbonGroup>
</RibbonTab>
```

Code 4.3: Die XAML-Anweisungen für das Register **Start** mit der Gruppe **Zwischenablage** und drei Symbolen

Über die Eigenschaft **Label** in der Gruppe **Sonstiges** im Eigenschaftenfenster legen Sie den Text fest, der auf der Schaltfläche erscheinen soll. Über die Eigenschaften **LargeImageSource**⁸ beziehungsweise **SmallImageSource** hinterlegen Sie Symbole in der Größe 32×32 Pixel beziehungsweise 16×16 Pixel. Sie finden diese Eigenschaften ebenfalls in der Gruppe **Sonstiges** des Eigenschaftenfensters. Damit Sie die Schaltflächen besser auseinanderhalten können, sollten Sie auch hier wieder sprechende Namen vergeben.

Das komplette Menüband könnte jetzt so aussehen:

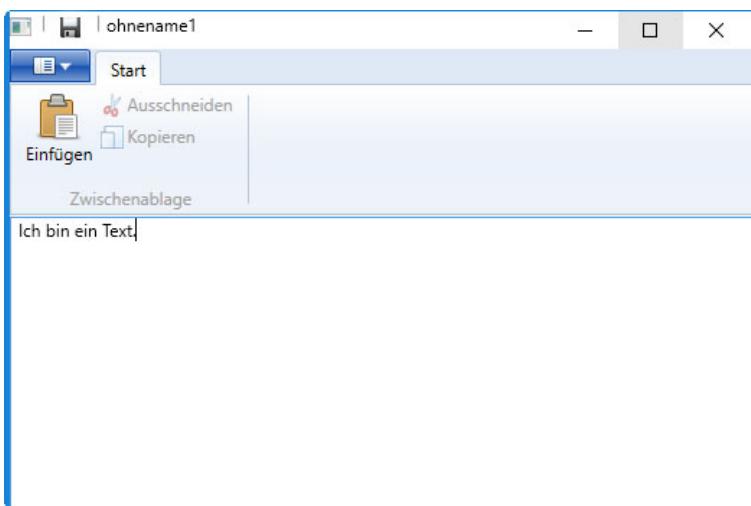


Abb. 4.8: Das Menüband

Damit wollen wir die Arbeit an unserem Menüband beenden – auch wenn an der einen oder anderen Stelle noch ein paar optische Verbesserungen erforderlich sind. Mehr zu den umfangreichen Möglichkeiten, die Ihnen das Menüband bietet, finden Sie in der Hilfe unter dem Stichwort **Microsoft.Windows.Controls.Ribbon**. Probieren Sie diese Möglichkeiten am besten selber in Ruhe aus.

8. LargeImageSource bedeutet übersetzt so viel wie „Quelle für großes Bild“.

4.4 Die Dateifunktionen

Im nächsten Schritt können wir jetzt die Funktionen zum Anlegen, Öffnen, Speichern beziehungsweise Speichern unter, Schließen und Beenden erstellen. Dazu verwenden wir wie bei der Windows Forms-Textverarbeitung Standardmethoden und einige selbst erstellte Methoden.

Hinweis:

Wir stellen Ihnen in diesem Kapitel vor allem die Änderungen vor. Wenn Sie nicht mehr genau wissen, wie die einzelnen Methoden im Detail arbeiten, sehen Sie bitte noch einmal bei der Windows Forms-Anwendung nach.

Im ersten Schritt sorgen wir wieder dafür, dass jedes neue Dokument einen „leeren“ Dateinamen erhält und eine Anzeige in der Titelleiste erscheint. Dabei soll neben dem Text „ohnename“ noch eine laufende Nummer angezeigt werden.

Vereinbaren Sie bitte ein Feld `zaehler` vom Typ `int` in der Klasse `MainWindow`. Setzen Sie den Wert dieses Feldes im Konstruktor auf den Wert 1. Legen Sie außerdem ein Feld `dateiname` vom Typ `string` und ein Feld `geaendert` vom Typ `bool` an.



Zur Erinnerung:

Den Quelltext der Klasse `MainWindow` finden Sie in der Datei `MainWindow.xaml.cs`.

Erstellen Sie dann eine Methode `NeuesDokument()`. In dieser Methode setzen wir einige Standardeinstellungen und löschen den Inhalt der `RichTextBox`. Die Methode sieht so aus:

```
void NeuesDokument()
{
    //den Text "löschen"
    richTextBox.Document = new FlowDocument();
    //die Anzeige in der Titelleiste setzen
    this.Title = "ohnename" + zaehler;
    //den Zähler um 1 erhöhen
    zaehler++;
    //es liegen keine Änderungen vor
    geaendert = false;
    //der Dateiname ist leer
    dateiname = string.Empty;
}
```

Code 4.4: Die Methode `NeuesDokument()`

Hinweis:

Damit Sie aus dem Quelltext auf die `RichTextBox` zugreifen können, müssen Sie zwingend einen Namen an das Steuerelement vergeben. Wir haben in unserem Beispiel den Namen `richTextBox` verwendet. Diesen Namen müssen Sie an den Namen anpassen, den Sie in Ihrem Projekt verwenden.

Besonderheiten gibt es eigentlich nur beim „Löschen“ der **RichTextBox**. Hier weisen wir der Eigenschaft **Document** von `richTextBox` eine neue Instanz der Klasse `FlowDocument` zu. Diese Klasse steht für den Inhalt einer **RichTextBox**.

Rufen Sie die Methode `NeuesDokument()` dann im Ereignis **Loaded** des Fensters auf. Dieses Ereignis tritt bei einer WPF-Anwendung nach dem Laden des Fensters ein. Lassen Sie die Methode außerdem beim Anklicken des Eintrags **Neu** im Anwendungsmenü ausführen.

Hinweis:

Den Menüeintrag wählen Sie am einfachsten durch einen Klick auf das entsprechende Tag in der XAML-Ansicht aus.

Führen Sie anschließend einen ersten Test durch.

Kommen wir nun zum Speichern. Hier sind deutlich mehr Änderungen und Erweiterungen erforderlich. Denn zum einen kennt die WPF keine Steuerelemente für Standarddialoge und zum anderen gibt es für das WPF-Steuerelement **RichTextBox** keine fertigen Methoden zum Speichern und Laden. Wir benutzen daher Standarddialoge von Windows und speichern beziehungsweise laden die Dateien über einen Dateistream der Klasse `FileStream`.

Die Methode `Speichern()`, die die Datei unter einem vorgegebenen Namen speichert, sieht dann so aus:

```
void Speichern(string name)
{
    dateiname = name;
    //den Text in der Titelleiste setzen
    this.Title = dateiname;
    //einen Stream erzeugen
    //die Datei wird dabei erzeugt bzw. überschrieben
    //bitte jeweils in einer Zeile eingeben
    using (System.IO.FileStream dateistream = new
        System.IO.FileStream(name, System.IO.FileMode.Create))
    {
        //den Bereich für das Speichern festlegen
        //wir nehmen den gesamten Inhalt der RichTextBox
        TextRange textBereich = new
            TextRange(richTextBox.Document.ContentStart,
            richTextBox.Document.ContentEnd);
        //nun über den Stream im RTF-Format speichern
        textBereich.Save(dateistream, DataFormats.Rtf);
    }
    //geändert auf false setzen, da die Änderungen
    //gespeichert sind
    geändert = false;
}
```

Code 4.5: Die Methode `Speichern()`

Mit der Anweisung

```
System.IO.FileStream dateiStream = new
System.IO.FileStream(name, System.IO.FileMode.Create);
```

erzeugen wir einen Stream aus dem Namen der Datei, der als Parameter übergeben wird. Die Datei wird dabei angelegt oder überschrieben.

Danach beschaffen wir uns mit der Anweisung

```
TextRange textBereich = new
TextRange(richtTextBox.Document.ContentStart,
richtTextBox.Document.ContentEnd);
```

einen Bereich vom Typ `TextRange`. Über dieses Objekt können wir gleich den Inhalt der **RichTextBox** speichern. Unser Bereich beginnt mit dem ersten Zeichen (`richtTextBox.Document.ContentStart`) und endet mit dem letzten Zeichen (`richtTextBox.Document.ContentEnd`) im Dokument der **RichTextBox**.

Mit der Zeile

```
textBereich.Save(dateiStream, DataFormats.Rtf);
```

schließlich speichern wir den Text über die Methode `Save` der Klasse `TextRange`. Als Argumente übergeben wir dabei den Stream und das Format RTF über die Konstante `DataFormats.Rtf`.

Danach setzen wir das Feld `geaendert` auf `false`. Dabei gibt es keine Besonderheiten.

In der Methode `DateiSpeichern()`, die die Datei entweder unter einem bereits vorhandenen Namen speichert oder den Dialog **Speichern unter** anzeigt, müssen wir den Speicherndialog selbst per Hand erzeugen. Die Methode sieht so aus:

```
void DateiSpeichern()
{
    //wenn der Dateiname leer ist, einen Speicherndialog
    //anzeigen
    if (dateiname == string.Empty)
    {
        //bitte in einer Zeile eingeben
        Microsoft.Win32.SaveFileDialog
        speichernDialog = new Microsoft.Win32.SaveFileDialog();
        //die Eigenschaften für den Dialog setzen
        speichernDialog.Filter = "RTF-Dateien|*.rtf";
        speichernDialog.FileName = string.Empty;
        //wurde der Dialog über Speichern geschlossen?
        //dann die Datei unter dem angegebenen Namen speichern
        if (speichernDialog.ShowDialog() == true)
            Speichern(speichernDialog.FileName);
    }
    //wenn der Dateiname nicht leer ist, direkt speichern
    else
        Speichern(dateiname);
}
```

Code 4.6: Die Methode `DateiSpeichern()`

Die drei Anweisungen

```
Microsoft.Win32.SaveFileDialog speichernDialog = new
Microsoft.Win32.SaveFileDialog();
speichernDialog.Filter = "RTF-Dateien|*.rtf";
speichernDialog.FileName = string.Empty;
```

erzeugen den Dialog und setzen den Filter sowie den Eintrag im Feld **Dateiname** des Dialogs.

Mit der Anweisung

```
if (speichernDialog.ShowDialog() == true)
```

überprüfen wir, ob der Dialog über die Standardschaltfläche verlassen wurde – also über die Schaltfläche **Speichern**. Dazu vergleichen wir den Rückgabewert der Methode `ShowDialog()` mit `true`. Wenn der Vergleich zutrifft, rufen wir unsere Methode `Speichern()` mit dem Dateinamen aus dem Dialog auf.

Lassen Sie die Methode `DateiSpeichern()` jetzt beim Anklicken des Symbols **Speichern**  in der Symbolleiste für den Schnellzugriff und beim Anklicken des Eintrags **Speichern** im Anwendungsmenü ausführen. Beim Anklicken des Eintrags **Speichern unter** im Anwendungsmenü zeigen wir in jedem Fall den Dialog **Speichern unter** an. Die entsprechende Methode sieht so aus:

```
private void MenuSpeichernUnter_Click(object sender,
RoutedEventArgs e)
{
    //bitte in einer Zeile eingeben
    Microsoft.Win32.SaveFileDialog speichernDialog =
    new Microsoft.Win32.SaveFileDialog();
    //die Eigenschaften für den Dialog setzen
    speichernDialog.Filter = "RTF-Dateien|*.rtf";
    speichernDialog.FileName = string.Empty;
    //wurde der Dialog über OK geschlossen?
    //dann die Datei unter dem angegebenen Namen speichern
    if (speichernDialog.ShowDialog() == true)
        Speichern(speichernDialog.FileName);
}
```

Code 4.7: Die Methode `MenuSpeichernUnter_Click()`

Hinweis:

Der Name der Methode hängt auch vom Namen des Steuerelements ab.

Da es in dieser Methode keine Besonderheiten gibt, kümmern wir uns direkt um das Öffnen von Dateien. Hier erzeugen wir ebenfalls per Hand einen Öffnendialog und laden dann die ausgewählte Datei über einen Stream in den Textbereich des Dokuments der **RichTextBox**. Die Methode sieht so aus:

```
//wenn nicht gespeicherte Änderungen vorliegen
if (geaendert == true)
{
    //Meldung mit Ja und Nein erzeugen und überprüfen, ob
    //Ja angeklickt wurde
```

```

//bitte in einer Zeile eingeben
if (MessageBox.Show("Wollen Sie die Änderungen
speichern?", "Abfrage", MessageBoxButton.YesNo,
MessageBoxImage.Question) == MessageBoxResult.Yes)
    DateiSpeichern();
}
//den Öffnendialog erzeugen
//bitte in einer Zeile eingeben
Microsoft.Win32.OpenFileDialog oeffnenDialog = new
Microsoft.Win32.OpenFileDialog();
//den Filter setzen
oeffnenDialog.Filter = "RTF-Dateien|*.rtf";
oeffnenDialog.FileName = string.Empty;
//wurde der Dialog über OK geschlossen?
//dann die Datei laden
if (oeffnenDialog.ShowDialog() == true)
{
    //einen Stream erzeugen
    //die Datei wird dabei geöffnet
    //bitte jeweils in einer Zeile eingeben
    using (System.IO.FileStream dateiStream = new
    System.IO.FileStream(oeffnenDialog.FileName,
    System.IO FileMode.Open))
    {
        //den Bereich für das Speichern festlegen
        //wir nehmen den gesamten Inhalt der RichTextBox
        TextRange textBereich = new
        TextRange(richTextBox.Document.ContentStart,
        richTextBox.Document.ContentEnd);
        //nun über den Stream im RTF-Format laden
        textBereich.Load(dateiStream, DataFormats.Rtf);
    }
    //geaendert auf false setzen, da es erst einmal keine
    //Änderungen gibt
    geaendert = false;
    //den Namen setzen
    dateiname = oeffnenDialog.FileName;
    this.Title = dateiname;
}

```

Code 4.8: Die Methode zum Öffnen von Dokumenten

Hinweis:

Auch hier hängt der Name der Methode vom Namen des Steuerelements ab.

Zunächst einmal überprüfen wir mit der Abfrage

```
if (geaendert == true),
```

ob Änderungen an dem Dokument vorliegen. Um das Setzen des Feldes `geaendert` kümmern wir uns allerdings erst gleich.

Wenn Änderungen vorliegen, erzeugen wir eine Meldung und speichern das Dokument auf Wunsch. Das übernehmen die Zeilen

```
if (MessageBox.Show("Wollen Sie die Änderungen  
speichern?", "Abfrage", MessageBoxButton.YesNo,  
MessageBoxImage.Question) == MessageBoxResult.Yes)  
    DateiSpeichern();
```

Bitte beachten Sie:

Die Konstanten für die Schaltflächen, die Symbole und auch die Rückgabe aus einem Dialog heißen bei der WPF anders als bei Windows Forms.



Mit den Anweisungen

```
Microsoft.Win32.OpenFileDialog oeffnenDialog = new  
Microsoft.Win32.OpenFileDialog();  
oeffnenDialog.Filter = "RTF-Dateien|*.rtf";  
oeffnenDialog.FileName = string.Empty;
```

erzeugen wir den Öffnendialog und setzen wieder den Filter sowie den Dateinamen. Anschließend überprüfen wir, ob die Schaltfläche **Öffnen** angeklickt wurde, und erzeugen einen Stream sowie einen Textbereich. Über diese beiden Objekte laden wir die Datei. Das wesentliche Verfahren kennen Sie dabei ja bereits vom Speichern.

Übernehmen Sie jetzt bitte die Anweisungen aus dem vorigen Code und lassen Sie sie beim Anklicken des Eintrags **Öffnen** im Anwendungsmenü ausführen.

Sorgen Sie dann noch dafür, dass das Feld `geaendert` bei Änderungen am Text den Wert `true` erhält. Die entsprechende Anweisung können Sie zum Beispiel im Ereignis **TextChanged** der **RichTextBox** ausführen lassen.

Nun fehlt nur noch die Sicherheitsabfrage beim Schließen der Anwendung. Sie sieht fast genauso aus wie bei der Windows Forms-Anwendung aus dem letzten Studienheft. Wir erzeugen beim Ereignis **Closing** einen Dialog und brechen das Schließen der Anwendung gegebenenfalls ab. Die Methode sieht so aus:

```
private void RibbonWindow_Closing(object sender,  
System.ComponentModel.CancelEventArgs e)  
{  
    //wenn nicht gespeicherte Änderungen vorliegen  
    if (geaendert == true)  
    {  
        //Meldung mit Ja, Nein und Abbrechen erzeugen  
        //und auswerten  
        //bitte in einer Zeile eingeben  
        switch (MessageBox.Show("Wollen Sie die  
Änderungen speichern?", "Abfrage",  
MessageBoxButton.YesNoCancel,  
MessageBoxImage.Question))  
        {  
            case MessageBoxResult.Yes:  
                //das Dokument speichern  
                DateiSpeichern();  
                break;  
        }  
    }  
}
```

```

        case MessageBoxResult.Cancel:
            //Abbrechen
            e.Cancel = true;
            break;
        }
    }
}

```

Code 4.9: Die Sicherheitsabfrage beim Schließen

Damit sind die Dateifunktionen für unsere WPF-Anwendung erst einmal komplett und wir können uns um die Funktionen für die Zwischenablage kümmern.

4.5 Funktionen für die Zwischenablage

Die Funktionen für die Zwischenablage lassen sich in der WPF-Anwendung sehr einfach umsetzen. Sie müssen nichts weiter machen, als einen Befehl – ein **Command** – aufzurufen.



Ein Befehl oder Command fasst mehrere Anweisungen für eine Aktion unter einem Namen zusammen. Die WPF kennt zahlreiche unterschiedliche Befehle für unterschiedlichste Bereiche.

Der Befehl für das Kopieren in die Zwischenablage lautet zum Beispiel `Copy` und der Befehl für das Einfügen aus der Zwischenablage `Paste`. Die Befehle können Sie sehr einfach aufrufen, indem Sie sie zum Beispiel der Eigenschaft **Command** einer Schaltfläche zuordnen. Sie finden diese Eigenschaft in der Gruppe **Sonstiges** im Eigenschaftenfenster.

Weisen Sie jetzt bitte den Schaltflächen im Register **Start** die folgenden Commands zu. Geben Sie dazu den Text für die Eigenschaft **Command** in das entsprechende Feld im Eigenschaftsfenster ein. Achten Sie bitte genau auf die korrekte Schreibweise und auch auf Groß- beziehungsweise Kleinschreibung.

Tab. 4.1: Die Befehle für die Menüeinträge

Schaltfläche	Command
Kopieren	Copy
Ausschneiden	Cut
Einfügen	Paste

Testen Sie die Erweiterungen dann. Sie werden sehen, die Schaltflächen werden jetzt auch automatisch aktiviert beziehungsweise deaktiviert.

So viel an dieser Stelle zu den Funktionen für die Zwischenablage.

4.6 Ein wenig Feinschliff

Abschließend wollen wir die Textverarbeitung noch mit ein wenig Feinschliff versehen.

Im ersten Schritt erstellen wir ein Kontext-Menü mit einigen Funktionen für die Zwischenablage und das Rückgängigmachen.

Markieren Sie bitte das RichTextBox-Steuerelement im Designer. Klicken Sie dann auf die Schaltfläche **Neu** hinter der Eigenschaft **ContextMenu** im Bereich **Sonstiges** und wählen Sie im folgenden Fenster den Eintrag **ContextMenu** aus.

Über die Untereigenschaften, die dann erscheinen, können Sie das Erscheinungsbild des Kontext-Menüs festlegen. Die Einträge selbst müssen Sie dabei allerdings wieder per Hand erstellen. Das Erstellen ist aber nicht sehr aufwendig, da Sie lediglich zwei Eigenschaften setzen müssen – den Text über die Eigenschaft **Header** und den Befehl über die Eigenschaft **Command**.

Ein Menüeintrag **Kopieren**, der den Befehl `Copy` ausführt, könnte dann so aussehen:

```
<MenuItem Header="Kopieren" Command="Copy"></MenuItem>
```

Eingeleitet wird die Vereinbarung durch das Start-Tag `MenuItem`. Dann folgen die beiden Eigenschaften mit der Zuweisung der Werte und abschließend das End-Tag für `MenuItem`.

Ein komplettes Kontext-Menü mit mehreren Einträgen für unsere **RichTextBox** würde dann so aussehen:

```
<RichTextBox.ContextMenu>
<ContextMenu>
    <MenuItem Header="Kopieren" Command="Copy"></MenuItem>
    <MenuItem Header="Ausschneiden" Command="Cut"></MenuItem>
    <MenuItem Header="Einfügen" Command="Paste"></MenuItem>
    <Separator></Separator>
    <MenuItem Header="Rückgängig" Command="Undo"></MenuItem>
    <MenuItem Header="Wiederholen" Command="Redo"></MenuItem>
</ContextMenu>
</RichTextBox.ContextMenu>
```

Code 4.10: Das Kontext-Menü für die **RichTextBox**

Das fertige Kontext-Menü sieht so aus:

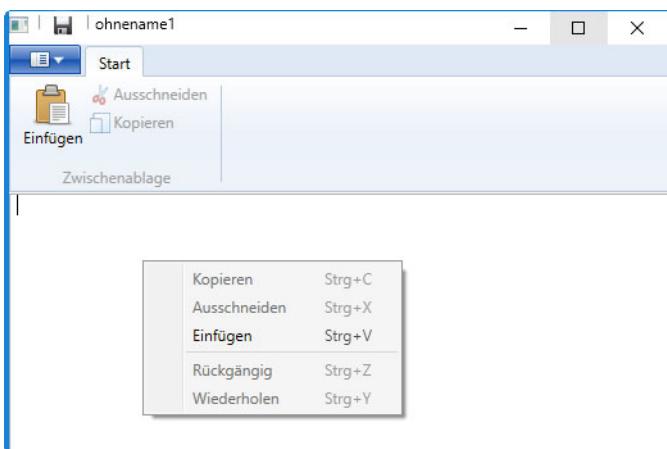


Abb. 4.9: Das Kontext-Menü

Hinweis:

Das RichTextBox-Steuerelement verfügt in der Standardeinstellung über ein Kontext-Menü mit den Funktionen **Ausschneiden**, **Kopieren** und **Einfügen**. Diese Funktionen werden allerdings überschrieben, wenn Sie ein eigenes Kontext-Menü erstellen.

Als I-Tüpfelchen ganz zum Schluss sorgen wir nun noch dafür, dass die Textverarbeitung eine Rechtschreibprüfung erhält. Dazu müssen Sie lediglich die Eigenschaft **SpellChecked.IsEnabled** für das RichTextBox-Steuerelement auf `true` setzen. In unserem Fall erledigen wir das in der Methode `NeuesDokument()`. Die vollständige Methode sieht nach dieser Änderung so aus:

```
void NeuesDokument()
{
    //den Text "löschen"
    richTextBox.Document = new FlowDocument();
    //die Anzeige in der Titelleiste setzen
    this.Title = "ohnename" + zaehler;
    //den Zähler um 1 erhöhen
    zaehler++;
    //es liegen keine Änderungen vor
    geaendert = false;
    //der Dateiname ist leer
    dateiname = string.Empty;
    //die Rechtschreibprüfung aktivieren
    richTextBox.SpellCheck.IsEnabled = true;
}
```

Code 4.11: Die Methode `NeuesDokument()` (die neue Anweisung ist fett markiert)

Die Rechtschreibprüfung erfolgt dann automatisch. Unbekannte Wörter werden mit einer roten Wellenlinie markiert und können – wie von den meisten Office-Anwendungen gewohnt – auch über das Kontext-Menü korrigiert werden.

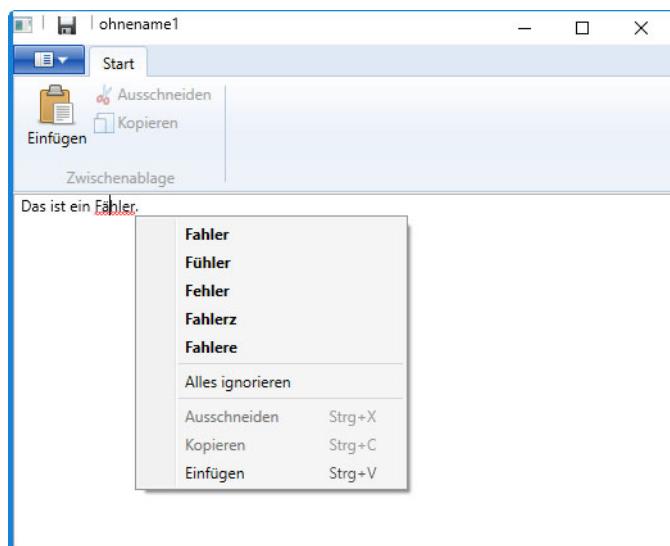


Abb. 4.10: Die Rechtschreibprüfung

Bitte beachten Sie:

Wenn Sie ein eigenes Kontext-Menü für ein RichTextBox-Steuerelement erstellt haben, deckt dieses Kontext-Menü das Kontext-Menü der Rechtschreibprüfung zu. Sie müssen dann Ihr eigenes Kontext-Menü entfernen.



Außerdem können Sie die Methode `NeuesDokument()` noch so ändern, dass auch hier vor dem Anlegen eines neuen Dokuments Abfragen erfolgen, ob eventuelle Änderungen gesichert werden sollen.

Damit wollen wir die Arbeit an der WPF-Textverarbeitung an dieser Stelle beenden. Wenn Sie Lust haben, können Sie ja selbst noch weitere Funktionen hinzufügen. Eine Erweiterung wartet auch als Einsendeaufgabe auf Sie.

Zusammenfassung

Über das Steuerelement **Ribbon** können Sie ein Menüband in eine WPF-Anwendung einfügen. Die Elemente müssen Sie allerdings selbst über XAML erstellen.

Es gibt in der WPF keine Steuerelemente für Standarddialoge wie **Öffnen** und **Speichern unter**. Sie müssen diese Dialoge selbst per Hand erstellen.

Das Laden und Speichern von Texten in einem RichTextBox-Steuerelement der WPF erfolgt über einen Stream und einen Textbereich.

Befehle oder Commands fassen mehrere Anweisungen für eine Aktion unter einem Namen zusammen. Der Befehl `Copy` kopiert zum Beispiel Objekte in die Zwischenablage.

Aufgaben zur Selbstüberprüfung

- 4.1 Wie fügen Sie ein Menüband in eine WPF-Anwendung ein?

- 4.2 Welche Steuerelemente können Sie wie schachteln, um ein Symbol in einer Symbolleiste für den Schnellzugriff in einer WPF-Anwendung zu erstellen?

- 4.3 Über welche Eigenschaft legen Sie das kleine Symbol für eine Schaltfläche in einer Registerkarte in einem Menüband fest?

- 4.4 Sie wollen einen Standarddialog **Speichern unter** für eine WPF-Anwendung erzeugen. Wie lautet die entsprechende Anweisung? Der Bezeichner für den Dialog ist beliebig.

Schlussbetrachtung

Sie kennen jetzt mit der Windows Presentation Foundation eine weitere Möglichkeit zur Entwicklung von Programmen mit grafischen Oberflächen im .NET Framework. Windows Forms und Windows Presentation Foundation sind sich in vielen Dingen zumindest beim flüchtigen Hinsehen recht ähnlich, weisen aber auch zum Teil deutliche Unterschiede auf. Für den Wechsel von Windows Forms-Anwendungen auf Windows Presentation Foundation-Anwendungen ist daher recht viel Aufwand erforderlich.

Dafür sind aber einige Sachen in WPF-Anwendungen nach einer kurzen Zeit der Umgewöhnung sehr viel einfacher – zum Beispiel durch den Einsatz von Commands.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Ein besonderes Kennzeichen bei der Entwicklung von WPF-Anwendungen ist die konsequente Trennung bei der Entwicklung der Oberfläche und bei der Entwicklung der Programmlogik.
- 1.2 Nein, auf der obersten Ebene eines Fensters einer WPF-Anwendung kann nur ein Stammelement liegen. Die weiteren Elemente müssen in Container-Elemente eingefügt werden.
- 1.3 Bei einem Projekt für eine WPF-Anwendung werden zwei XAML-Dateien erstellt. Die eine Datei steht für das Fenster und die andere Datei für die eigentliche Anwendung.
- 1.4 Um bei einem WPF-Steuerelement **Label** den Text zu setzen, ändern Sie die Eigenschaft **Content**.

Kapitel 2

- 2.1 Auf den Text in einem Label greifen Sie mit der Eigenschaft **Content** zu. Diese Eigenschaft liefert den Wert als Typ `object`. Vor der Zuweisung auf eine Variable vom Typ `string` müssen Sie den Wert also noch konvertieren.
- 2.2 Es gibt zwei Möglichkeiten:
 - Sie erstellen ein Objekt der Klasse `ComboBoxItem` und weisen diesem Objekt den aktuell ausgewählten Eintrag zu. Über die Eigenschaft `Content` des Objekts können Sie dann auf den Text zugreifen.
 - Sie benutzen die Eigenschaft `Text`. Sie liefert Ihnen direkt den Text in dem Kombinationsfeld.

Für die richtige Lösung reicht es aus, wenn Sie eine der beiden Möglichkeiten genannt haben.

Kapitel 3

- 3.1 Der Designer arbeitet mit einem **Grid** als Standard-Container. Es besteht allerdings nur aus einer einzigen Zelle.
- 3.2 Ja. Schaltflächen werden genauso behandelt wie alle anderen grafischen Elemente und können damit ohne Qualitätsverlust nahezu beliebig vergrößert und verkleinert werden.
- 3.3 Ereignisse, die von unten nach oben weitergereicht werden, heißen im Fachjargon **Bubbling Events**.

- 3.4 Ereignisse für die Reaktion auf getunnelte Ereignisse beginnen normalerweise mit `Preview`.
- 3.5 Um die Weiterleitung von Ereignissen zu unterbrechen, setzen Sie in dem Ereignis, in dem Sie die Verarbeitung abbrechen wollen, die Eigenschaft `e.Handled` auf `true`.

Kapitel 4

- 4.1 Das Einfügen muss über XAML erfolgen. In der Toolbox gibt es kein entsprechendes Steuerelement.
- 4.2 Um ein Symbol in einer Symbolleiste für den Schnellzugriff in einer WPF-Anwendung zu erstellen, schachteln Sie die Steuerelemente **RibbonQuickAccessToolBar** und **RibbonButton**. Das Steuerelement **RibbonButton** liegt dabei in dem Steuerelement **RibbonQuickAccessToolBar**.
- 4.3 Das kleine Symbol für eine Schaltfläche in einer Registerkarte legen Sie über die Eigenschaft **SmallImageSource** fest.
- 4.4 Die Anweisung zum Erzeugen eines Standarddialogs **Speichern unter** in einer WPF-Anwendung lautet

```
Microsoft.Win32.SaveFileDialog speichernDialog = new  
Microsoft.Win32.SaveFileDialog();
```

Wenn Sie den Namensraum `Microsoft.Win32` global vereinbart haben, kann die Angabe auch wegfallen.

Statt `speichernDialog` können Sie auch einen anderen Bezeichner verwenden.

B. Glossar

Befehl (WPF)	Ein Befehl der WPF fasst mehrere Anweisungen für eine Aktion unter einem Namen zusammen. Die WPF kennt zahlreiche unterschiedliche Befehle für unterschiedlichste Bereiche.
Bubbling Events	<i>Bubbling Events</i> werden die Ereignisse genannt, die in einer WPF-Anwendung von unten nach oben weitergereicht werden.
Code-Behind-Dateien	Die <i>Code-Behind</i> -Dateien sind die Quelltextdateien für die Programmlogik einer WPF-Anwendung oder einer Universal Windows Platform App.
Command (WPF)	Siehe Befehl (WPF).
Container	Ein Container nimmt weitere Steuerelemente in einer WPF-Anwendung auf.
Extensible Application Markup Language	Siehe XAML.
Rich-Text-Format	Siehe RTF.
Routed Events	<i>Routed Events</i> steht für weitergeleitete Ereignisse in einer WPF-Anwendung.
RTF	Das RTF-Format (<i>Rich-Text-Format</i> ; wörtlich übersetzt „reicher Text“) ist ein weitverbreitetes Format für die Speicherung von Textdokumenten mit Formatierungen. Es kann von nahezu allen Textverarbeitungsprogrammen gelesen und auch erzeugt werden.
Stream	Ein <i>Stream</i> ist ein unformatierter Datenstrom. Die Interpretation erfolgt durch das Zielgerät, das den Stream verarbeitet.
Tunneling Events	<i>Tunneling Events</i> werden die Ereignisse in einer WPF-Anwendung genannt, die von oben nach unten laufen.
Windows Presentation Foundation	Die <i>Windows Presentation Foundation</i> ist eine Plattform zur Entwicklung von grafischen Oberflächen. Die Entwicklung ist eindeutig getrennt in die Logik auf der einen Seite und die Präsentation auf der anderen Seite.
WPF	Siehe <i>Windows Presentation Foundation</i> .
XAML	XAML steht für <i>eXtensible Application Markup Language</i> . Es handelt sich um die Beschreibungssprache für die Oberfläche von WPF-Anwendungen. XAML basiert auf XML und stellt die Informationen ebenfalls strukturiert in Textform dar.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 6. Aufl., Bonn: Rheinwerk.

Speziell mit dem Thema WPF beschäftigt sich das folgende Buch:

Huber, T.C. (2019). *Windows Presentation Foundation*. Das umfassende Handbuch. 5. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Das erste XAML-„Programm“	4
Abb. 1.2	Das erweiterte Hallo-Welt-„Programm“	5
Abb. 1.3	Der Dialog Neues Projekt erstellen mit dem markierten Eintrag WPF-App	6
Abb. 1.4	Das Gerüst für die WPF-Anwendung	7
Abb. 1.5	Das eingefügte Label	8
Abb. 1.6	Das Fenster nach den Änderungen	9
Abb. 1.7	Die erste ausgeführte WPF-Anwendung	9
Abb. 1.8	Das Bearbeiten der Methode Button_Click ()	10
Abb. 2.1	Der Sammlungs-Editor	15
Abb. 2.2	Ein Eintrag im Sammlungs-Editor	15
Abb. 2.3	Ein Kombinations- und ein Listenfeld	16
Abb. 3.1	Ein StackPanel	20
Abb. 3.2	Ein StackPanel in horizontaler Ausrichtung von rechts nach links	20
Abb. 3.3	Ein WrapPanel	21
Abb. 3.4	Das DockPanel	21
Abb. 3.5	Ein Grid mit zwei Spalten und zwei Zeilen	22
Abb. 3.6	Die Hierarchie der Steuerelemente	24
Abb. 3.7	Tunneling Events (die gestrichelte rote Linie links) und Bubbling Events (die durchgezogene grüne Linie rechts)	25
Abb. 4.1	Die WPF-Textverarbeitung	28
Abb. 4.2	Der Eintrag System.Windows.Controls.Ribbon	29
Abb. 4.3	Der eingefügte Verweis	30
Abb. 4.4	Das Menüband im Fenster	31
Abb. 4.5	Die Rohform des Fensters	32
Abb. 4.6	Der neue Ordner im Projektmappen-Explorer	34
Abb. 4.7	Die Liste der Grafikdateien für die Eigenschaft SmallImageSource	35
Abb. 4.8	Das Menüband	37
Abb. 4.9	Das Kontext-Menü	45
Abb. 4.10	Die Rechtschreibprüfung	46
Abb. H.1	Der Taschenrechner als WPF-Anwendung	59

E. Tabellenverzeichnis

Tab. 4.1 Die Befehle für die Menüeinträge	44
---	----

F. Codeverzeichnis

Code 1.1	Die XAML-Beschreibung für ein Hallo-Welt-„Programm“	4
Code 1.2	Die erweiterte XAML-Beschreibung	5
Code 2.1	Der Quelltext für die Methode ButtonKopieren_Click()	13
Code 2.2	Die Methoden ComboBox_SelectionChanged() und ListBox_SelectionChanged()	16
Code 2.3	Die Methode CheckBox1_Click()	17
Code 3.1	Das Drehen eines Objekts	23
Code 3.2	Die Methoden für Tunneling Events	25
Code 4.1	Die XAML-Anweisungen für das Symbol in der Symbolleiste für den Schnellzugriff	33
Code 4.2	Die XAML-Anweisungen für das Anwendungsmenü	36
Code 4.3	Die XAML-Anweisungen für das Register Start mit der Gruppe Zwischenablage und drei Symbolen	37
Code 4.4	Die Methode NeuesDokument()	38
Code 4.5	Die Methode Speichern()	39
Code 4.6	Die Methode DateiSpeichern()	40
Code 4.7	Die Methode MenuSpeichernUnter_Click()	41
Code 4.8	Die Methode zum Öffnen von Dokumenten	42
Code 4.9	Die Sicherheitsabfrage beim Schließen	44
Code 4.10	Das Kontext-Menü für die RichTextBox	45
Code 4.11	Die Methode NeuesDokument()	46

G. Sachwortverzeichnis

B

Bubbling Event 25

C

Code-Behind-Datei 7
 Command 44
 Container 19
 Content 4

D

Dateifunktion 38
 DirectX 22
 DockPanel 21

E

Ereignis
 Weiterleitung 24
 eXtensible Application Markup
 Language 3

F

Funktion
 für die Zwischenablage 45

G

Grafik
 Darstellung 22
 vektorbasierte 22
 Grid 19, 21

K

Kontext-Menü 45

P

Projekt
 für eine WPF-Anwendung erstellen ..
 6

R

Rechtschreibprüfung 46
 Routed Event 25

S

Sammlungs-Editor 14

StackPanel 19

Stammelement 5

T

Textverarbeitung
 als WPF-Anwendung 28
 Tunneling Event 25

W

Windows Presentation Foundation 3
 WPF 3
 WrapPanel 21

X

XAML 3
 XAML-Beschreibung 4

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH14D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:
Datum:
Note:
Unterschrift Fernlehrer/in:

Schicken Sie für die Lösungen bitte jeweils das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

Beschreiben Sie bei allen Aufgaben zusätzlich, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Änderungen Sie vornehmen beziehungsweise welche Steuerelemente Sie verwenden.

- Setzen Sie den Taschenrechner, den Sie bereits als Windows Forms-Anwendung erstellen haben, als WPF-Anwendung um. Die Eingabe der Zahlen soll über zwei **TextBoxen** erfolgen und die Auswahl der Rechenoperationen über ein Optionsfeld.

Die Oberfläche des Taschenrechners sollte ungefähr so aussehen:

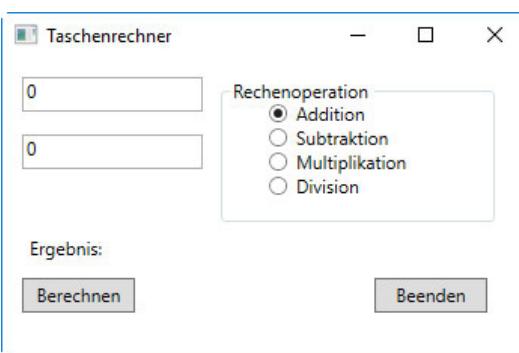


Abb. H.1: Der Taschenrechner als WPF-Anwendung

Wie Sie die Oberfläche genau erstellen, bleibt Ihnen überlassen. Bitte achten Sie aber darauf, dass Sie direkt in einer **GroupBox** bei einer WPF-Anwendung nur ein Steuerelement einfügen können. Weitere Elemente lassen sich nur dann einfügen, wenn Sie in die **GroupBox** noch einmal einen Container setzen.

50 Pkt.

2. Erweitern Sie das Menüband für die Textverarbeitung aus diesem Studienheft um eine Gruppe **Format** in der Registerkarte **Start**. In dieser Gruppe sollen drei Schaltflächen für die Zeichenformatierungen **Fett**, **Kursiv** und **Unterstreichen** angezeigt werden.

Bilder für die Symbole finden Sie bei den Beispielen auf Ihrer Online-Lernplattform im Ordner \icons unter den Namen **fettneu.png**, **kursivneu.png** und **unterstreichneu.png**.

Ein kleiner Hinweis zur Lösung:

Die Zeichenformatierungen können Sie sehr einfach über Commands ausführen lassen, die wie ein Schalter arbeiten. Sehen Sie sich dazu die Hilfe zur Klasse **Editing-Commands** an. Die verschiedenen Commands finden Sie im Bereich **Eigenschaften**.

50 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Fortgeschrittene Techniken
mit der Windows Presentation Foundation

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

0919N01

CSHP15D

Objektorientierte Software-Entwicklung mit C#

Fortgeschrittene Techniken mit der Windows Presentation Foundation

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Fortgeschrittene Techniken mit der Windows Presentation Foundation

Inhaltsverzeichnis

Einleitung	1
1 Ein Medioplayer	3
1.1 Die Klasse MediaElement	3
1.2 Die Oberfläche	5
1.3 Die Funktionalität	9
Zusammenfassung	15
2 Ein kleines Malprogramm	17
2.1 Das Verarbeiten von Mausereignissen	17
2.2 Die Oberfläche	20
2.3 Die Zeichenoperationen	26
2.4 Die Dateioperationen	30
Zusammenfassung	33
3 Animationen und 3D-Grafiken	36
3.1 Animationen	36
3.2 3D-Grafiken	40
3.3 Grafische Effekte	45
Zusammenfassung	46
4 Datenbindung	48
4.1 Grundlagen	48
4.2 Einige einfache Beispiele	51
4.3 Ein komplexeres Beispiel	54
Zusammenfassung	60
Schlussbetrachtung	63

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	64
B.	Glossar	66
C.	Literaturverzeichnis	68
D.	Abbildungsverzeichnis	69
E.	Tabellenverzeichnis	70
F.	Codeverzeichnis	71
G.	Sachwortverzeichnis	72
H.	Einsendeaufgabe	75

Einleitung

In diesem Studienheft tauchen wir tiefer in die Programmierung von WPF-Anwendungen ein. Dabei beschäftigen wir uns unter anderem mit der Wiedergabe von Medien und der Darstellung von 2D- und 3D-Grafiken. Außerdem sehen wir uns an, wie Sie Animationen erstellen. Abschließend werfen wir noch einen Blick auf die Datenbindung.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie Audio- und Videodateien über die Klasse `MediaElement` wiedergeben,
- wie Sie Symbolleisten für eine WPF-Anwendung erstellen,
- wie Sie Schaltflächen mit Symbolen erstellen,
- wie Sie Hinweise für Symbole zur Laufzeit setzen,
- wie Sie Symbole deaktivieren und wieder aktivieren,
- wie Sie eine Statusleiste in einer WPF-Anwendung erstellen,
- wie Sie Funktionen zur Steuerung der Audio- und Videowiedergabe umsetzen,
- wie Sie auf das Drehen des Mausrads reagieren,
- wie Sie Shapes im Programmcode erzeugen,
- wie Sie Objekte mit der Maus zeichnen können,
- wie Sie Objekte als XAML-Code speichern und wieder laden,
- wie Sie Animationen mit Objekten erstellen,
- wie Sie 3D-Grafiken erstellen,
- wie Sie Videos auf anderen Objekten abspielen lassen,
- was sich hinter der Datenbindung verbirgt,
- wie Sie Eigenschaften über die Datenbindung automatisch aktualisieren lassen,
- wie Sie ein 3D-Objekt über eine Datenbindung transformieren,
- wie Sie Daten bei der Datenbindung konvertieren und
- wie Sie mehrere Quellen bei der Datenbindung benutzen.

Christoph Siebeck

1 Ein Medioplayer

In diesem Kapitel erstellen wir einen einfachen Medioplayer, der sowohl Videos als auch Audiodateien abspielen kann. Neben der reinen Wiedergabe programmieren wir dabei auch einige Steuerungsfunktionen – zum Beispiel das Unterbrechen und Fortsetzen der Wiedergabe und das Verändern der Lautstärke.

1.1 Die Klasse MediaElement

Für die Wiedergabe von Audio- und Videodateien können Sie die Klasse **MediaElement** verwenden. Sie greift intern auf den Media Player von Windows zurück und unterstützt die Wiedergabe aller gängigen Formate.

Bitte beachten Sie:

Sie können die Klasse nur dann einsetzen, wenn der Media Player ab der Version 10 installiert ist. Außerdem können Sie mit der Klasse keine Dateien wiedergeben, die Sie als Ressource in ein Projekt einbetten.



Schauen wir uns die Klasse an einem ganz einfachen Beispiel im praktischen Einsatz an. Legen Sie eine neue WPF-Anwendung an. Fügen Sie dann ein **MediaElement**-Steuerelement aus der Toolbox ein. Sie finden es in der Gruppe **Alle WPF-Steuerelemente**. Positionieren Sie das Steuerelement so, dass es links oben im Fenster angezeigt wird. Dazu können Sie die Werte in den Feldern der Eigenschaft **Margin** in der Gruppe **Layout** jeweils auf 0 setzen. Damit das **MediaElement** immer in der Breite des Fensters angezeigt wird, ändern Sie die Einträge in den Feldern für die Eigenschaften **Width** und **Height** in der Gruppe **Layout** in **Auto**. Klicken Sie dazu auf das Symbol **Auf Auto festlegen** hinter dem Feld. Setzen Sie dann gegebenenfalls noch die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** auf **Stretch**. Prüfen Sie dazu, ob jeweils das rechte Symbol bei den Eigenschaften markiert ist. Wenn das nicht der Fall ist, klicken Sie es bitte an.

Über die Eigenschaft **Source** in der Gruppe **Medien** legen Sie nun die Datei fest, die wiedergegeben werden soll. Dazu tragen Sie den kompletten Pfad in das Feld ein. Für ein Video mit dem Namen **demo.mp4**, das sich im Ordner **\beispiele** auf der Festplatte C: befindet, muss der Eintrag dann so aussehen: **c:\beispiele\demo.mp4**.

Hinweis:

Ob der Pfad korrekt ist und die Datei wiedergegeben werden kann, können Sie ganz einfach im Fenster erkennen. Wenn nach der Eingabe ein Vorschaubild im Steuerelement erscheint, ist die Wiedergabe möglich. Wenn die Datei nicht gefunden wird, oder nicht wiedergegeben werden kann, ändert sich die Anzeige im Steuerelement nicht.

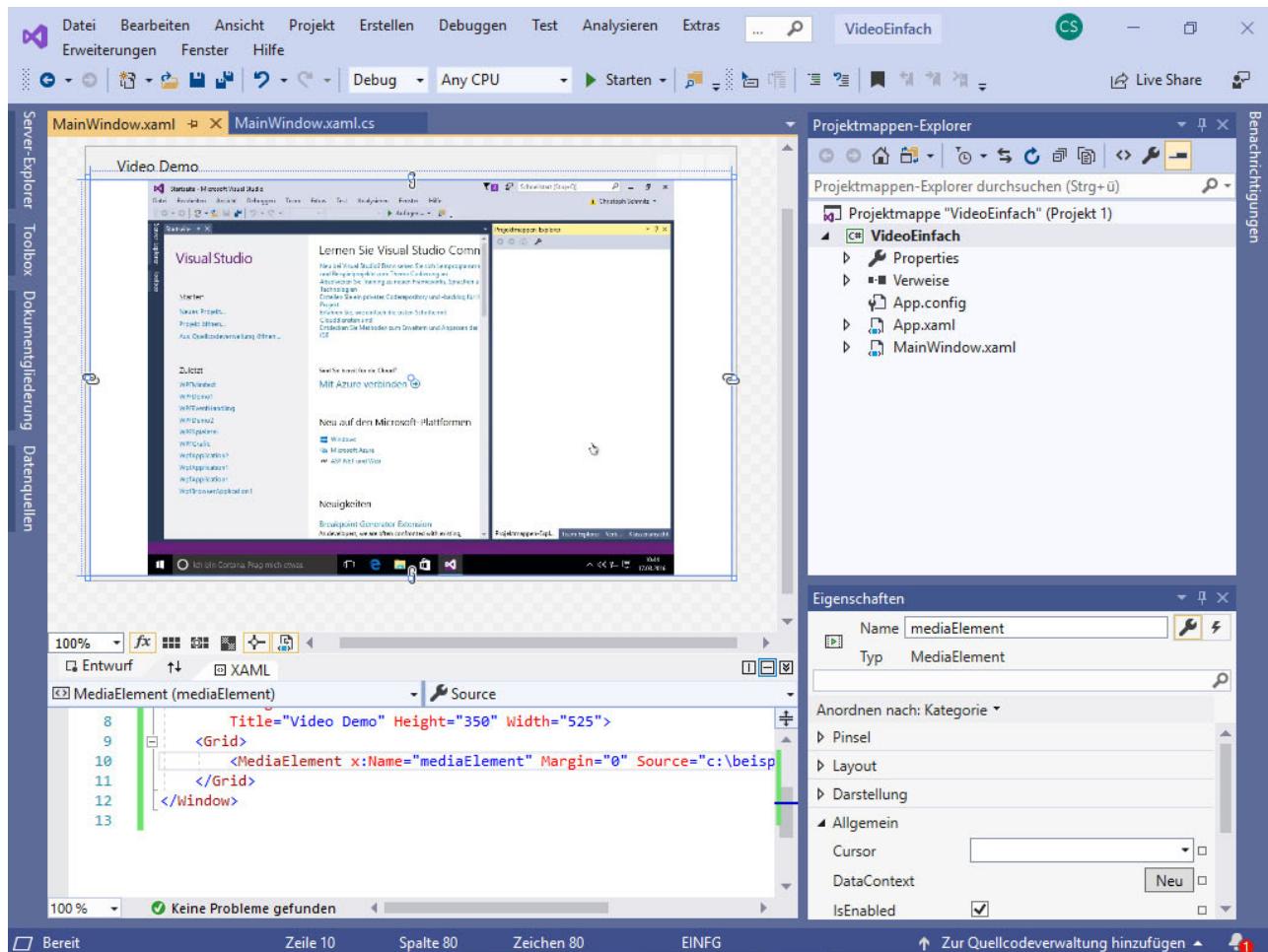


Abb. 1.1: Die Vorschau eines Videos im Steuerelement

Über weitere Eigenschaften in der Gruppe Medien können Sie unter anderem auch die Lautstärke (**Volume**) oder die Startposition (**Position**) festlegen.

Hinweis:

Die maximale Lautstärke beträgt 1. Mit der Standardeinstellung 0,5 erfolgt die Wiedergabe also mit 50 Prozent der maximalen Lautstärke.

In der Standardeinstellung wird die Wiedergabe automatisch gestartet, wenn die Datei komplett geladen wurde. Beim Entladen – also zum Beispiel beim Schließen des Fensters – wird die Wiedergabe automatisch beendet und die Datei geschlossen.

Probieren Sie das Verhalten jetzt einmal selbst aus. Laden Sie über die Eigenschaft **Source** eine Video- oder Audiodatei und starten Sie das Programm dann. Nach kurzer Zeit sollte die Wiedergabe beginnen.

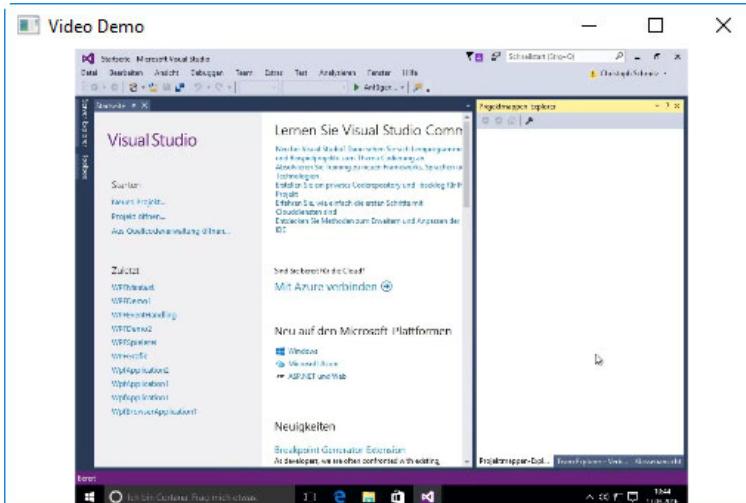


Abb. 1.2: Die Wiedergabe eines Videos

Die einfache Wiedergabe finden Sie im Projekt **VideoEinfach** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. In dem Projekt wird ein Video mit dem Namen **demo.mp4** aus dem Ordner **C:\beispiele** wiedergegeben. Wenn dieses Video auf Ihrem Rechner nicht vorhanden ist, geschieht beim Start des Programms nichts.



Sie finden das Video bei den Beispielen im Download-Bereich Ihrer Online-Lernplattform. Sie können aber auch ein beliebiges anderes Video oder eine andere Audiodatei benutzen.

1.2 Die Oberfläche

Machen wir uns jetzt an die Umsetzung des eigenen Mediaplayers. Er soll über folgende Grundfunktionen verfügen:

- Öffnen von beliebigen Mediendateien,
- Wiedergabe der Dateien,
- Beenden der Wiedergabe,
- Unterbrechen der Wiedergabe und
- Fortführen der Wiedergabe.

Diese Grundfunktionen bieten wir über eine Symbolleiste an. Die Symbole sollen dabei je nach Zustand entweder aktiv oder inaktiv sein. Für das Unterbrechen und das Fortführen der Wiedergabe benutzen wir dabei nur ein Symbol. Wenn die Wiedergabe aktuell läuft, soll es zum Unterbrechen dienen. Wenn die Wiedergabe unterbrochen ist, soll es zum Fortführen dienen.

Zusätzlich lassen wir noch den Namen der aktuell wiedergegebenen Datei in einem Label in einer Statusleiste anzeigen. In diese Statusleiste setzen wir außerdem einen Schieberegler zum Verändern der Lautstärke.

Der fertige Medioplayer soll so aussehen:

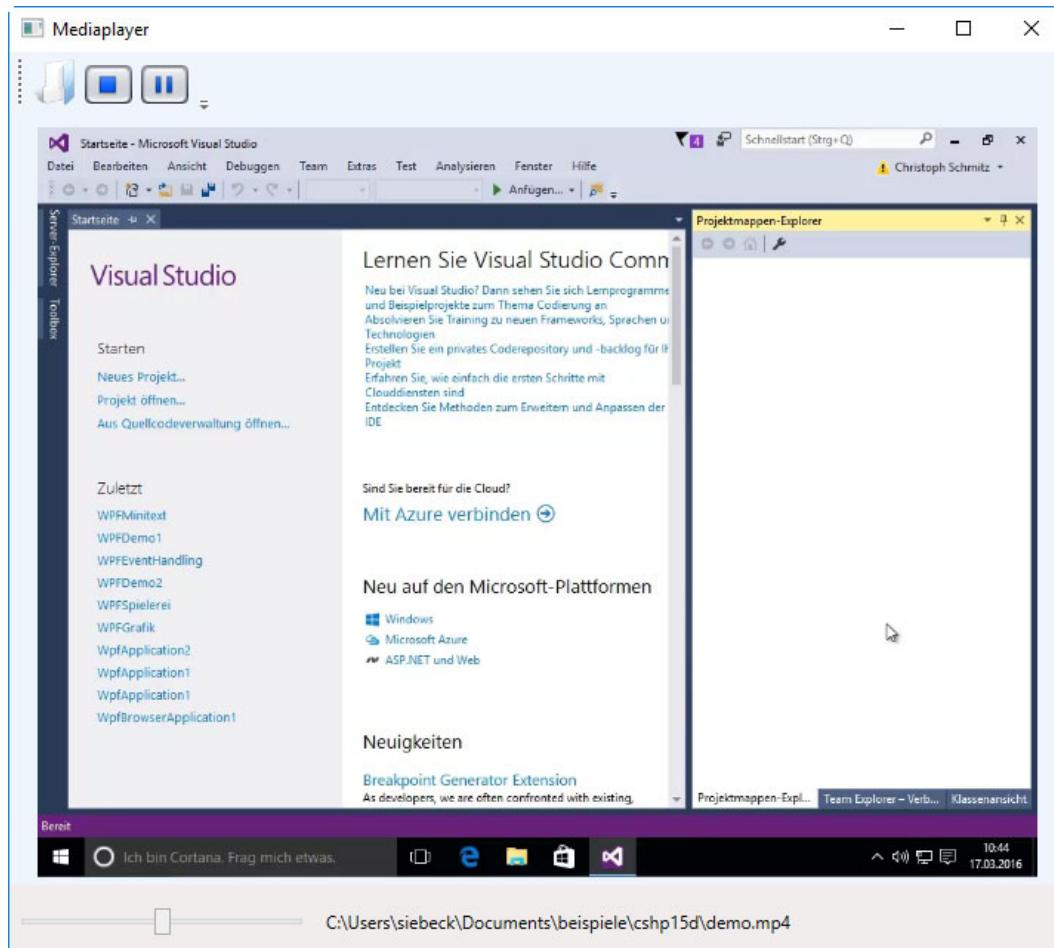


Abb. 1.3: Der Medioplayer



Den fertigen Medioplayer finden Sie im Projekt **MediaPlayer** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Legen Sie bitte eine neue WPF-Anwendung an. Setzen Sie die Eigenschaft **Title** für den Text in der Titelleiste des Fensters auf **Mediaplayer** und lassen Sie das Fenster über die Eigenschaft **WindowState** maximiert darstellen.

Im nächsten Schritt kommt die Symbolleiste an die Reihe. Sie wird bei einer WPF-Anwendung in einem Steuerelement **ToolBarTray**¹ abgelegt. Dieses Steuerelement dient als Container für die eigentlichen Symbolleisten. Sie werden über ein Steuerelement vom Typ **Toolbar** abgebildet, das in den Container vom Typ **ToolBarTray** platziert wird. In die Symbolleisten können Sie wie gewohnt Schaltflächen, Kombinationsfelder oder auch Trennlinien einfügen. Dazu können Sie zum Beispiel über das Symbol für die Eigenschaft **Items** der Symbolleiste den Sammlungs-Editor öffnen.

1. *Tray* lässt sich mit „Ablage“ übersetzen.

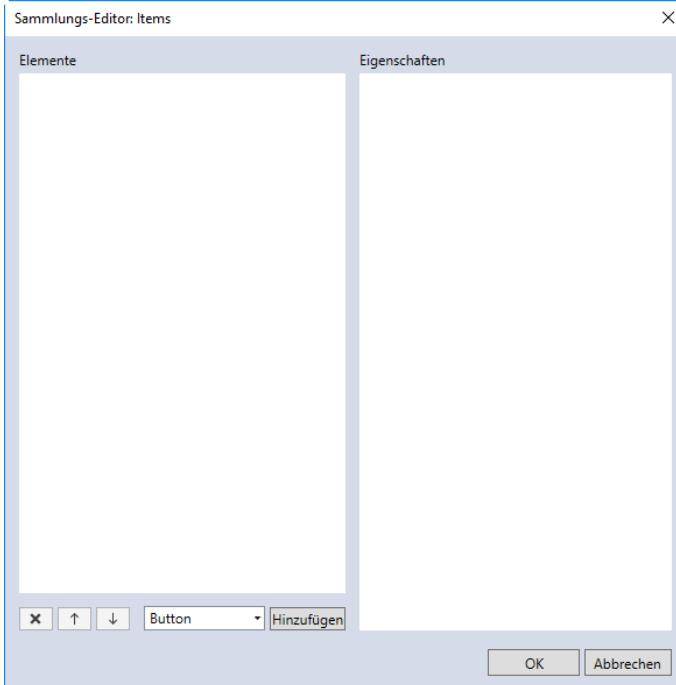


Abb. 1.4: Der Sammlungs-Editor

Im Sammlungs-Editor können Sie dann über das Kombinationsfeld auswählen, was Sie hinzufügen wollen, und anschließend wie gewohnt die Eigenschaften im rechten Bereich bearbeiten.

Ein Symbol für eine Schaltfläche können Sie allerdings mit dem Sammlungs-Editor nicht direkt festlegen. Denn dazu müssen Sie zunächst ein Steuerelement vom Typ **Image** auf der Schaltfläche platzieren. Und spätestens dann wird die Arbeit mit dem grafischen Editor für die Oberfläche sehr unübersichtlich. Wir legen daher die entsprechenden Strukturen per Hand im XAML-Code des Fensters an. Diese Struktur sieht erst einmal so aus:

```
<ToolBarTray>
  <ToolBar>
    <Button>
      <Image>
        </Image>
      </Button>
    </ToolBar>
  </ToolBarTray>
```

Code 1.1: Die XAML-Anweisungen für die Symbolleiste

Ganz außen befindet sich der Container vom Typ **ToolBarTray**. Dann folgt eine Symbolleiste, die eine Schaltfläche enthält. Die Schaltfläche wiederum enthält ein Bild.

Da wir gleich insgesamt drei Symbole brauchen, kopieren Sie die Anweisungen für die Schaltfläche noch zweimal. Achten Sie dabei sorgfältig darauf, dass Sie die Tags korrekt platzieren.

Besonders viel zu sehen ist von unseren Schaltflächen allerdings noch nicht. Das liegt daran, dass wir weder einen Text noch ein Bild zugewiesen haben. Holen wir das jetzt nach.

Legen Sie dazu bitte im ersten Schritt einen Ordner für die Symbole im Projekt an. Kopieren Sie in diesen Ordner die Grafiken für das Öffnen einer Datei sowie das Stoppen, das Unterbrechen und Fortsetzen der Wiedergabe.

Hinweis:

Die entsprechenden Symbole finden Sie im Ordner `\icons` bei den Beispielen auf Ihrer Online-Lernplattform. Die Symbole stammen aus der kostenlosen Icon-Bibliothek Open Icon Library. Sie finden diese Bibliothek im Internet auf der Seite <https://sourceforge.net/projects/openiconlibrary/>.

Weisen Sie den Schaltflächen dann von links nach rechts die Eigenschaften aus der Tab. 1.1 zu:

Tab. 1.1: Eigenschaften für die Schaltflächen

Name	ToolTip	IsEnabled
buttonOeffnen	Öffnen	Ja
buttonStopp	Wiedergabe beenden	Nein
buttonPlayPause	Wiedergeben	Nein

Setzen Sie anschließend noch über die Eigenschaft **Source** die Bilder für die Image-Komponenten der einzelnen Schaltflächen. Die Schaltfläche ganz links soll das Öffnensymbol erhalten, die Schaltfläche in der Mitte das Stoppsymbol und die Schaltfläche ganz rechts das Wiedergebensymbol.

Passen Sie anschließend – wenn erforderlich – die Breite und Höhe der Schaltflächen so an, dass sie jeweils 38 Pixel breit und 38 Pixel hoch sind. Die Symbolleiste sollte nun so aussehen:

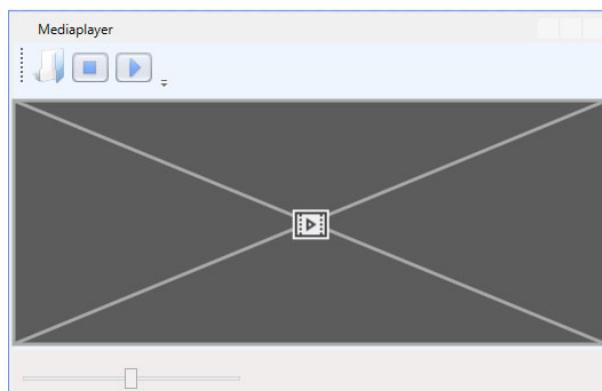


Abb. 1.5: Die Symbolleiste mit den drei Symbolen

Fügen Sie dann unten in das Fenster ein Steuerelement **StatusBar** für die Statusleiste ein. Setzen Sie in die Leiste ein Steuerelement vom Typ **Slider** für den Schieberegler und ein Label. Lassen Sie den Schieberegler ein wenig breiter darstellen – zum Beispiel mit 200 Pixeln. Die Eigenschaft **Maximum** setzen Sie bitte auf 100 und die Eigenschaft **Value** auf 50. Das entspricht 50 % der Lautstärke. Den Text im Label können Sie löschen.

Damit die Statusleiste immer unten im Fenster in der kompletten Breite angezeigt wird, setzen Sie die Breite bitte auf `Auto`. Die Eigenschaft **HorizontalAlignment** setzen Sie auf `Stretch` und die Eigenschaft **VerticalAlignment** auf `Bottom`. Passen Sie außerdem die Höhe an. Sie können sie zum Beispiel auf 50 setzen.

Setzen Sie abschließend in die Mitte des Fensters ein Steuerelement vom Typ **MediaElement**. Positionieren Sie es unterhalb der Symbolleiste und setzen Sie die Höhe so, dass es den gesamten freien Bereich im Fenster einnimmt. Die Breite setzen Sie auf `Auto`. Die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** setzen Sie jeweils auf `Stretch`.

Um zu testen, ob die Positionen korrekt gesetzt werden, können Sie im Steuerelement für die Wiedergabe eine beliebige Datei anzeigen lassen. Falls die Anzeige nicht korrekt ist, passen Sie die Größe des MediaElements nachträglich noch einmal an.

Das Fenster mit einem Platzhalter für die Anzeige sollte jetzt ungefähr so aussehen:

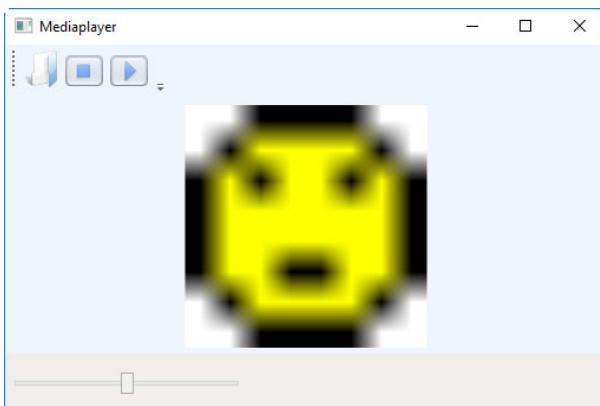


Abb. 1.6: Das Fenster in der Rohform
(das Label ist nicht zu sehen, da es keinen Inhalt hat)

1.3 Die Funktionalität

Kümmern wir uns jetzt um die Funktionalität. Beginnen wir mit dem Laden einer Datei.

Hier erzeugen wir einen Standarddialog zum Öffnen von Dateien und weisen die Auswahl der Eigenschaft `Source` des `MediaElements` zu. Dabei gibt es nur eine Besonderheit: Die Eigenschaft `Source` erwartet den Typ `Uri`². Wir können die Rückgabe des Öffnendialogs aber einfach in diesen Typ umbauen.

2. `Uri` steht für *Uniform Resource Identifier*. Es handelt sich dabei um die eindeutige Identifizierung einer Ressource.

Die Methode für das Öffnen sieht dann so aus:

```
private void Buttonoeffnen_Click(object sender,
RoutedEventArgs e)
{
    //den Dialog erzeugen
    //bitte in einer Zeile eingeben
    Microsoft.Win32.OpenFileDialog oeffnenDialog= new
    Microsoft.Win32.OpenFileDialog();
    //das Feld für den Namen ist leer
    oeffnenDialog.FileName = String.Empty;
    //wurde der Dialog über OK geschlossen?
    //dann die Datei laden
    if (oeffnenDialog.ShowDialog() == true)
        mediaElement.Source = new Uri(oeffnenDialog.FileName);
}
```

Code 1.2: Die Methode für das Öffnen

Richtig neu ist hier nur das Erzeugen einer Instanz der Klasse `Uri` mit dem Namen, der vom Öffnendialog geliefert wird. Das erledigt die letzte Anweisung

```
mediaElement.Source = new Uri(oeffnenDialog.FileName);
```

Legen Sie jetzt eine Methode für das Anklicken des Öffnensymbols an und übernehmen Sie die Anweisungen aus dem vorigen Code.

Hinweis:

Wir haben bewusst keine Filter für den Dialog gesetzt, um den Code möglichst einfach zu halten. Wenn Sie das stört, können Sie die Filter über die entsprechende Eigenschaft ja selbst setzen.

Im nächsten Schritt sorgen wir dafür, dass die Wiedergabe nicht mehr automatisch gestartet wird, sondern erst nach dem Anklicken des entsprechenden Symbols. Setzen Sie dazu zunächst die Eigenschaft **LoadedBehaviour**³ für das MediaElement auf `Manual`. Sie finden diese Eigenschaft bei den erweiterten Eigenschaften für die Gruppe **Medien**.

Damit wir einfach unterscheiden können, ob aktuell ein Medium wiedergegeben wird oder nicht, vereinbaren Sie im nächsten Schritt bitte ein Feld `wiedergabe` vom Typ `bool`. Wir werden es je nach Status auf `true` oder `false` setzen und abhängig vom Status dieses Feldes auch die Schaltfläche aktivieren beziehungsweise die Symbole auf den Schaltflächen setzen. Da direkt nach dem Start des Programms noch keine Wiedergabe läuft, setzen Sie das Feld bitte zunächst auf `false`.

Ergänzen Sie nach dem Laden der Datei auch noch eine Anweisung, um das Symbol für die Wiedergabe zu aktivieren. Die entsprechende Anweisung könnte so aussehen:

```
buttonPlayPause.IsEnabled = true;
```

Lassen Sie außerdem den Namen der geladenen Datei in dem Label anzeigen. Die Anweisung könnte so aussehen:

```
label.Content = oeffnenDialog.FileName;
```

³. `LoadedBehaviour` lässt sich mit „Verhalten nach dem Laden“ übersetzen.

Starten Sie anschließend beim Anklicken der Schaltfläche für die Wiedergabe das Abspielen der Datei. Dazu rufen Sie die Methode `Play()` für das `MediaElement` auf. Sorgen Sie außerdem dafür, dass nach dem Start der Wiedergabe die Schaltfläche zum Abbrechen aktiviert wird, und rufen Sie beim Anklicken dieser Schaltfläche die Methode `Stop()` für das `MediaElement` auf.

Die beiden Methoden für das Anklicken der Schaltflächen könnten dann so aussehen:

```
private void ButtonPlayPause_Click(object sender, RoutedEventArgs e)
{
    //die Wiedergabe starten
    mediaElement.Play();
    //die Schaltfläche für das Stoppen aktivieren
    buttonStop.IsEnabled = true;
}

private void ButtonStop_Click(object sender, RoutedEventArgs e)
{
    //die Wiedergabe stoppen
    mediaElement.Stop();
}
```

Code 1.3: Die Methoden für das Anklicken der Schaltflächen

Damit ist die wichtigste Grundfunktionalität bereits fertiggestellt. Wir können eine Datei laden, die Wiedergabe starten und auch wieder beenden.

Bitte beachten Sie unbedingt:

Wenn Sie selbst die Steuerung über die Wiedergabe und das Stoppen übernehmen wollen, müssen Sie die Eigenschaft `LoadedBehaviour` für das `MediaElement` zwingend auf `Manual` setzen. Andernfalls kann der Aufruf der Methoden zu einem Absturz des Programms führen.



Kommen wir jetzt zum Unterbrechen der Wiedergabe. Dazu rufen wir die Methode `Pause()` für das `MediaElement` auf. Der Aufruf soll dabei über dieselbe Schaltfläche erfolgen, die wir auch für den Start der Wiedergabe benutzen. Wir müssen also dafür sorgen, dass die Schaltfläche unterschiedliche Symbole anzeigen und auch unterschiedliche Funktionen aufrufen kann. Dazu verwenden wir unser Feld `wiedergabe`. Wenn es den Wert `false` hat, soll beim Anklicken der Schaltfläche die Wiedergabe starten und das Symbol für eine Pause angezeigt werden. Hat das Feld dagegen den Wert `true`, soll beim Anklicken der Schaltfläche die Wiedergabe unterbrochen und das Symbol für die Wiedergabe angezeigt werden.

Damit das funktioniert, müssen wir dem Steuerelement `Image` für die Schaltfläche zunächst einmal einen Namen geben. Wir benutzen in unserem Fall den Bezeichner `imagePlayPause`. Über die Eigenschaft `Source` können wir dann im Quelltext ein anderes Symbol zuweisen. Dazu müssen wir allerdings eine neue Instanz der Klasse `BitmapImage` erzeugen und die Daten für diese Instanz über eine neue Instanz der Klasse `Uri` laden. Da wir die Symbole in unserem Beispiel in einem Unterordner des

Projekts abgelegt haben, müssen wir über das Argument `UriKind.Relative` noch festlegen, dass die Suche nach dem Symbol über relative Pfadangaben ausgehend vom Projektordner erfolgt.

Die Anweisung zum Setzen des neuen Symbols sieht also so aus:

```
imagePlayPause.Source = new BitmapImage(new
Uri("symbole/pause.png", UriKind.Relative));
```

Hinweis:

Wenn Sie einen absoluten Pfad verwenden möchten, müssen Sie die Angabe `UriKind.Absolute` setzen und den absoluten Pfad vollständig im ersten Argument angeben.

Die geänderte Methode für das Anklicken der Schaltfläche sieht dann so aus wie im folgenden Code. Hier werden die Symbole gesetzt, die Tooltips geändert und die entsprechenden Funktionen aufgerufen.

```
private void ButtonPlayPause_Click(object sender,
RoutedEventArgs e)
{
    //wenn aktuell keine Wiedergabe läuft
    if (wiedergabe == false)
    {
        //die Wiedergabe starten
        mediaElement.Play();
        //das Symbol wechseln
        //bitte in einer Zeile eingeben
        imagePlayPause.Source = new BitmapImage(new
Uri("symbole/pause.png", UriKind.Relative));
        //und auch den Tooltip
        buttonPlayPause.ToolTip = "Unterbrechen";
        //jetzt läuft die Wiedergabe
        wiedergabe = true;
    }
    //wenn aktuell eine Wiedergabe läuft
    else
    {
        //die Wiedergabe unterbrechen
        mediaElement.Pause();
        //das Symbol wechseln
        //bitte in einer Zeile eingeben
        imagePlayPause.Source = new BitmapImage(new
Uri("symbole/play.png", UriKind.Relative));
        //und auch den Tooltip
        buttonPlayPause.ToolTip = "Wiedergeben";
        //jetzt läuft keine Wiedergabe
        wiedergabe = false;
    }
    //die Schaltfläche für das Stoppen aktivieren
    buttonStopp.IsEnabled = true;
}
```

Code 1.4: Die geänderte Methode für das Symbol

Damit die Anzeige der Symbole beim Laden und Stoppen nicht durcheinanderkommt, sollten Sie nach dem Laden und dem Stoppen in jedem Fall das Symbol für die Wiedergabe anzeigen lassen. Ergänzen Sie dazu die Anweisung

```
imagePlayPause.Source = new BitmapImage(new Uri("symbole/play.png", UriKind.Relative));
```

am Ende der Methoden zum Laden einer Datei und zum Stoppen der Wiedergabe. Setzen Sie außerdem in den beiden Methoden das Feld `wiedergabe` wieder auf den Standardwert `false` und lassen Sie den Tooltipp `Wiedergeben` für die Schaltfläche anzeigen.

Kommen wir jetzt noch zum Ändern der Lautstärke. Hier passen wir im Ereignis `ValueChanged` für den Schieberegler die Eigenschaft `Volume` für das `MediaElement` an. Da die Angabe im Bereich zwischen 0,0 und 1,0 erfolgt, dividieren wir den aktuellen Wert des Schiebereglers durch 100. Da das Ereignis `ValueChanged` aber unter Umständen beim Erstellen des Fensters bereits vor dem Erstellen des Steuerelements für die Medienwiedergabe ausgelöst wird, prüfen wir zur Sicherheit, ob der Wert von `mediaElement` ungleich `null` ist. Nur dann ändern wir die Lautstärke.

Die vollständige Methode besteht nur aus wenigen Anweisungen und sieht so aus:

```
private void Slider_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
{
    //gibt es mediaElement bereits?
    if (mediaElement != null)
        //dann setzen wir die Lautstärke
        mediaElement.Volume = slider.Value / 100;
}
```

Code 1.5: Die Methode zum Verändern der Lautstärke

Hinweis:

Wenn Sie auf die Abfrage

```
if (mediaElement != null)
```

verzichten, startet das Programm unter Umständen nicht mehr beziehungsweise bleibt hängen. Denn dann versuchen Sie, die Lautstärke für ein nicht vorhandenes `MediaElement` zu setzen. Und das führt zu einer Ausnahme, die aber beim normalen Ausführen des Programms nicht angezeigt wird. Dazu müssen Sie das Programm im Debug-Modus starten. Dann erscheint auch eine entsprechende Meldung und das Programm wird unterbrochen.

Wenn Sie also beim Ausführen eines Programms auf seltsame Effekte stoßen, testen Sie es immer auch im Debugger.

Im letzten Schritt wollen wir noch ein wenig Feinschliff an dem Medioplayer vornehmen. So kann ein Anwender ja zum Beispiel nicht erkennen, dass die beiden Symbole zunächst deaktiviert sind. Sie sehen ja genauso aus wie aktive Symbole. Um den Unterschied deutlicher zu machen, setzen wir zunächst die Eigenschaft `Opacity`⁴ in der Gruppe **Darstellung** der beiden Schaltflächen auf einen Wert um 50. Dadurch erscheinen die

4. `Opacity` bedeutet übersetzt „Deckkraft, Deckfähigkeit“.

Schaltflächen durchlässiger. Nach dem Laden einer Datei setzen wir die Eigenschaft dann wieder auf den Standardwert 100. Damit werden die Schaltflächen wieder normal dargestellt.



Der Wert der Eigenschaft **Opacity** wird prozentual angegeben. Im Eigenschaftenfenster geben Sie zum Beispiel den Wert 50 ein. Im Code sollten Sie den Wert 0.5 benutzen.

Als kleines Bonbon wollen wir am Ende noch die Änderung der Lautstärke über das Mausrad zulassen. Dazu verwenden wir das Ereignis **MouseWheel** für das Fenster. In diesem Ereignis werten wir die Drehrichtung des Rads aus und setzen die Lautstärke neu. In welche Richtung das Mausrad gedreht wurde, können wir über die Eigenschaft `e.Delta` abfragen. Wenn sie einen positiven Wert liefert, wurde das Rad nach vorn gedreht, andernfalls nach hinten. Damit alles seine Ordnung hat, aktualisieren wir auch die Anzeige des Schiebereglers. Die Methode sieht dann so aus:

```
private void Window_MouseWheel(object sender,
MouseEventArgs e)
{
    //in welche Richtung wurde das Mausrad gedreht?
    if (e.Delta > 0)
        //nach vorne erhöhen wir die Lautstärke
        mediaElement.Volume = mediaElement.Volume + 0.1;
    else
        //nach hinten reduzieren wir sie
        mediaElement.Volume = mediaElement.Volume - 0.1;
    //die Anzeige des Schiebereglers aktualisieren
    slider.Value = mediaElement.Volume * 100;
}
```

Code 1.6: Das Ändern der Lautstärke über das Mausrad

Damit wollen wir die Arbeit an dem Medioplayer beenden – auch wenn es noch einige unschöne Problemchen gibt. So startet zum Beispiel die Wiedergabe einer anderen Datei, die Sie bei einer laufenden Wiedergabe laden, automatisch. Das führt dann dazu, dass die Anzeige auf unseren Symbolen nicht mehr stimmt. Außerdem ist es unter Umständen irritierend, dass die Wiedergabe nach dem Laden nicht automatisch startet. Diese Kleinigkeiten können Sie aber selbst anpassen.

Hinweis:

Der Code in dem Projekt lässt sich etwas kompakter gestalten. Sie können zum Beispiel die Anweisungen zum Setzen der verschiedenen Symbole und Tooltipps in eigenen Methoden zusammenfassen. Dann müssen Sie die Anweisungen nicht immer wiederholen. Wir haben darauf aber aus Gründen der Einfachheit verzichtet.

Zusammenfassung

Für die Wiedergabe von Audio- und Videodateien können Sie die Klasse `MediaElement` verwenden.

In der Standardeinstellung wird die Wiedergabe automatisch gestartet.

Sie können die Wiedergabe aus dem Code über die Methoden `Play()`, `Stop()` und `Pause()` steuern.

Eine Symbolleiste fügen Sie über das Steuerelement **ToolBar** ein. Sie sollten Symbolleisten in einem Steuerelement vom Typ **ToolBarTray** ablegen.

Um ein Symbol auf einer Schaltfläche anzuzeigen, setzen Sie ein Steuerelement vom Typ **Image** auf die Schaltfläche.

Über ein Steuerelement **StatusBar** fügen Sie eine Statusleiste ein.

Über die Eigenschaft **Opacity** können Sie die Durchlässigkeit von Steuerelementen steuern.

Mit dem Ereignis **MouseWheel** können Sie auf das Drehen des Mausrads reagieren.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Worauf müssen Sie beim Einsatz der Klasse `MediaElement` unbedingt achten?

- 1.2 In welchem Wertebereich legen Sie die Lautstärke für die Wiedergabe über die Klasse `MediaElement` fest?

- 1.3 Welchen Typ erwartet die Eigenschaft **Source** für die Klasse `MediaElement`?

- 1.4 Welche Eigenschaft müssen Sie unbedingt auf welchen Wert setzen, wenn Sie aus dem Code die Steuerung der Medienwiedergabe in einem `MediaElement` übernehmen wollen?

- 1.5 Sie wollen auf einer Schaltfläche ein Bild aus der Datei `bild.png` anzeigen lassen. Diese Datei liegt in einem Ordner `bilder` des Projekts. Wie lautet die entsprechende Anweisung? Für das `Image`-Steuerelement können Sie einen beliebigen Bezeichner wählen.

- 1.6 Mit welcher Eigenschaft können Sie feststellen, in welche Richtung das Mausrad gedreht wurde?

2 Ein kleines Malprogramm

In diesem Kapitel erstellen wir ein kleines Malprogramm, mit dem sich einfache geometrische Figuren erstellen lassen. Außerdem soll das Programm über Funktionen zum Laden und Speichern von Grafikdateien verfügen.

Bevor wir uns an die praktische Umsetzung machen, wollen wir uns aber erst noch einmal die Mausereignisse in der WPF ein wenig genauer ansehen.

2.1 Das Verarbeiten von Mausereignissen

Die wichtigsten Mausereignisse der WPF finden Sie in der Tab. 2.1. Einige davon kennen Sie bereits von den Windows-Forms-Anwendungen.

Tab. 2.1: Mausereignisse der WPF

Ereignis	Auslöser
MouseDown	Dieses Ereignis wird ausgelöst, wenn eine beliebige Maustaste gedrückt wird.
MouseUp	Dieses Ereignis wird ausgelöst, wenn eine beliebige Maustaste losgelassen wird.
MouseLeftButtonDown	Dieses Ereignis wird ausgelöst, wenn die linke Maustaste gedrückt wird.
MouseLeftButtonUp	Dieses Ereignis wird ausgelöst, wenn die linke Maustaste losgelassen wird.
MouseRightButtonDown	Dieses Ereignis wird ausgelöst, wenn die rechte Maustaste gedrückt wird.
MouseRightButtonUp	Dieses Ereignis wird ausgelöst, wenn die rechte Maustaste losgelassen wird.
MouseEnter	Dieses Ereignis wird ausgelöst, wenn die Maus ein Steuerelement betritt – also nachdem der Mauszeiger über dem Steuerelement positioniert wurde.
MouseLeave	Dieses Ereignis wird ausgelöst, wenn die Maus ein Steuerelement wieder verlässt.
MouseMove	Dieses Ereignis wird ausgelöst, wenn die Maus bewegt wird.
MouseWheel	Dieses Ereignis wird ausgelöst, wenn das Mausrad gedreht wird.

Die Klassen für die Mausereignisse kennen auch einige interessante Methoden und Eigenschaften. Mit der Methode `GetPosition()` können Sie zum Beispiel die Mauskordinaten abfragen. Über die Eigenschaft `ButtonState` können Sie den Status einer be-

liebigen Maustaste abfragen. Über die Eigenschaften `LeftButton`, `MiddleButton` und `RightButton` können Sie auch gezielt auf die einzelnen Maustasten zugreifen. Die Anzahl der Mausklicks liefert Ihnen die Eigenschaft `ClickCount`.

Die Verarbeitung der Mausereignisse erfolgt wie gewohnt über Methoden, die Sie dem jeweiligen Ereignis zuordnen.

Schauen wir uns das jetzt in der Praxis an. Das folgende Fragment zeigt die Mauskoordinaten und die gedrückte Maustaste an. Dazu verwenden wir die Mausereignisse **MouseMove** und **MouseDown** für ein Steuerelement vom Typ **Canvas**.



Das Steuerelement **Canvas** ist ein Layout-Container, in dem untergeordnete Elemente über absolute Koordinaten positioniert werden können. Wir werden dieses Steuerelement gleich auch in unserem Zeichenprogramm verwenden.

```
private void Canvas_MouseDown(object sender,
MouseEventArgs e)
{
    //Wie oft wurde die linke Maustaste gedrückt?
    if (e.LeftButton == MouseButtons.Pressed)
        //bitte in einer Zeile eingeben
        label.Content = "Die linke Maustaste wurde " +
        e.ClickCount.ToString() + "mal gedrückt an ";
    //Wie oft wurde die rechte Maustaste gedrückt?
    if (e.RightButton == MouseButtons.Pressed)
        //bitte in einer Zeile eingeben
        label.Content = "Die rechte Maustaste wurde " +
        e.ClickCount.ToString() + "mal gedrückt an ";
    //Wo befand sich die Maus?
    Point position = e.GetPosition(meinCanvas);
    //bitte in einer Zeile eingeben
    label.Content = label.Content + position.X.ToString() + ";" + 
    position.Y.ToString();
}

private void Canvas_MouseMove(object sender, MouseEventArgs e)
{
    //wo befindet sich die Maus?
    Point position = e.GetPosition(meinCanvas);
    //bitte in einer Zeile eingeben
    label.Content = position.X.ToString() + ";" + 
    position.Y.ToString();
}
```

Code 2.1: Die Verarbeitung von Mausereignissen

Die Überprüfung der Maustasten erfolgt in dem Code durch die Abfragen

```
if (e.LeftButton == MouseButtons.Pressed)
```

und

```
if (e.RightButton == MouseButtons.Pressed)
```

Hier überprüfen wir jeweils, ob die linke oder rechte Maustaste den Status `MouseButtons.Pressed` hat.

Die Position ermitteln wir durch den Ausdruck `e.GetPosition(meinCanvas)`. Er liefert einen Typ `Point` mit den Koordinaten. Die Position wird dabei relativ zu dem Steuerelement angegeben, das Sie als Argument übergeben. In unserem Fall ist das der Container vom Typ `Canvas`. Wir sprechen ihn über den Bezeichner `meinCanvas` an.

Einen vollständigen Code, in dem auf unterschiedliche Mausereignisse reagiert wird, finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **MausEreignisseWPF**.



Kommen wir nun zum Zeichnen mit der Maus.

Dazu müssen wir zwischen dem Drücken und dem Loslassen der Maustaste unterscheiden können. Beim Drücken soll das Zeichnen beginnen und beim Loslassen wieder beendet werden. Wir benutzen also die Ereignisse `MouseLeftButtonDown` und `MouseLeftButtonUp`.

Beim Drücken der Maustaste machen wir nichts weiter, als die Position zu speichern, an der die Maustaste gedrückt wurde. Beim Loslassen der Maustaste zeichnen wir eine Figur zu der Position, an der die Maustaste losgelassen wurde.

Der folgende Code 2.2 erlaubt zum Beispiel das Zeichnen von Linien.

```
private void MeinCanvas_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{
    //start muss in der Klasse als Feld vom Typ Point vereinbart
    //werden
    //den Startpunkt sichern
    start = e.GetPosition(meinCanvas);
}

private void MeinCanvas_MouseLeftButtonUp(object sender,
MouseButtonEventArgs e)
{
    //eine neue Linie erzeugen
    Line linie = new Line();
    //der "Umriss" ist Schwarz
    linie.Stroke = Brushes.Black;
    //und ein wenig dicker
    linie.StrokeThickness = 5;
    //die Startpunkte setzen
    linie.X1 = start.X;
    linie.Y1 = start.Y;
    //und die Endpunkte
    linie.X2 = e.GetPosition(meinCanvas).X;
    linie.Y2 = e.GetPosition(meinCanvas).Y;

    //die Linie zum Canvas hinzufügen
    //sonst wird sie nicht angezeigt
    meinCanvas.Children.Add(linie);
}
```

Code 2.2: Ein erstes, sehr einfaches Malprogramm

In der Methode `MeinCanvas_MouseLeftButtonDown()` speichern wir lediglich die Koordinaten, an der die Maustaste gedrückt wurde. Dazu verwenden wir ein Feld vom Typ `Point` mit dem Namen `start`.

In der Methode `MeinCanvas_MouseLeftButtonUp()` erzeugen wir eine neue Instanz der Klasse `Line` und setzen die Farbe des „Umrisses“ über die Eigenschaft `Stroke` über den Ausdruck `Brushes.Black` auf Schwarz. Damit die Linien besser zu sehen sind, setzen wir die Breite über die Eigenschaft `StrokeThickness` auf 5. Die folgenden Anweisungen setzen dann den Startpunkt über die Koordinaten `x1` und `y1` und den Endpunkt über die Koordinaten `x2` und `y2`. Als Endpunkt verwenden wir dabei die Position, an der die Maustaste losgelassen wurde.

Damit die Linien auch tatsächlich angezeigt werden, müssen wir sie als untergeordnetes Element dem Container hinzufügen. Das erledigt die Anweisung:

```
meinCanvas.Children.Add(linie);
```

Sie fügt über die Methode `Add()` der Eigenschaft `Children` das Element ein, das als Argument angegeben wird. Die Eigenschaft `Children` steht dabei für die untergeordneten Elemente des Steuerelements beziehungsweise Containers – in unserem Fall also für die untergeordneten Elemente von `meinCanvas`.



Den vollständigen Code für das erste einfache Malprogramm finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **EinfachMalen-WPF**.

Bitte beachten Sie unbedingt:

Mausereignisse für einen Container vom Typ `Canvas` werden nur dann ausgelöst, wenn der Container keinen transparenten Hintergrund hat. Das ist in der Standardeinstellung des Containers allerdings der Fall. Wie Sie den Hintergrund ändern, erfahren Sie gleich, wenn wir das Malprogramm erstellen.

Im nächsten Schritt werden wir das Programm zu einem echten kleinen Malprogramm ausbauen.

2.2 Die Oberfläche

Unser Malprogramm soll folgende Funktionen anbieten:

- Zeichnen von Linien mit der Maus
- Zeichnen von Rechtecken mit der Maus
- Zeichnen von Kreisen mit der Maus
- Setzen einer anderen Zeichenfarbe über einen Standarddialog von Windows sowie
- Laden und Speichern der Grafiken.

Die Auswahl der Funktionen erfolgt dabei vor allem über ein Menüband.

Das Malprogramm soll später ungefähr so aussehen:

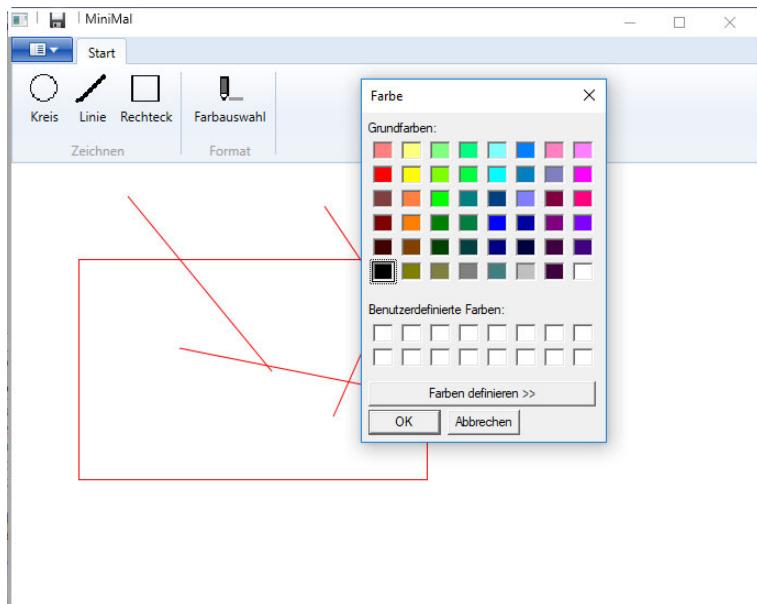


Abb. 2.1: Das Malprogramm

Beim Zeichnen setzen wir im Wesentlichen dieselben Techniken ein, die wir auch schon im vorherigen Code beim Zeichnen einer einfachen Linie benutzt haben.

Das fertige Malprogramm finden Sie im Projekt **MiniMal** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.



Beginnen wir jetzt mit der Umsetzung. Legen Sie bitte eine neue WPF-Anwendung an. Setzen Sie die Eigenschaft **Title** für den Text in der Titelleiste des Fensters auf **MiniMal** und lassen Sie das Fenster über die Eigenschaft **WindowState** maximiert darstellen.

Im nächsten Schritt kommt das Menüband an die Reihe. Da Sie solch ein Menüband ja bereits eingesetzt haben, stellen wir Ihnen die nötigen Schritte nur noch einmal in der Übersicht vor.

Fügen Sie im ersten Schritt über die Funktion **Projekt/Verweis hinzufügen ...** einen Verweis auf die Datei **System.Windows.Controls.Ribbon.dll** zum Projekt hinzu.

Damit die Symbolleiste für den Schnellzugriff optimal angezeigt wird, ändern Sie das Start-Tag direkt zu Beginn des XAML-Codes von **Window** in **RibbonWindow**. Passen Sie auch das End-Tag für das Fenster entsprechend an, falls diese Änderung nicht automatisch durchgeführt wird.

Machen Sie in der Datei **MainWindow.xaml.cs** den Namensraum `System.Windows.Controls.Ribbon` über die Anweisung

```
using System.Windows.Controls.Ribbon;
bekannt.
```

Leiten Sie die Klasse für das Fenster dann von der Klasse `RibbonWindow` ab. Ändern Sie dazu die Vereinbarung der Klasse von

```
public partial class MainWindow : Window
in
public partial class MainWindow : RibbonWindow
```

Fügen Sie im Designer im XAML-Code das Tag `<Ribbon>` zwischen den Tags für das Grid ein. Das End-Tag sollte automatisch vom Editor ergänzt werden.

Legen Sie dann eine Symboleiste für den Schnellzugriff an. Wir verwenden in unserem Beispiel lediglich ein Symbol zum Speichern. Wenn Sie Lust haben, können Sie später ja auch noch Funktionen wie das Laden ergänzen.

Stellen Sie die Einfügemarkie in der XAML-Ansicht zwischen die Tags für das Menüband. Übernehmen Sie anschließend die Anweisungen aus dem folgenden Code 2.3.

```
<Ribbon.QuickAccessToolBar>
  <RibbonQuickAccessToolBar>
    <RibbonButton></RibbonButton>
  </RibbonQuickAccessToolBar>
</Ribbon.QuickAccessToolBar>
```

Code 2.3: Die XAML-Anweisungen für das Symbol in der Symboleiste für den Schnellzugriff

Erstellen Sie dann im Projekt einen Ordner für die Symbole. Kopieren Sie in diesen Ordner die Grafiken für das Neuerstellen, Öffnen und Speichern einer Datei, für das Zeichnen der geometrischen Figuren und für das Öffnen des Dialogs zur Farbauswahl.

Hinweis:

Die entsprechenden Symbole finden Sie ebenfalls im Ordner `\icons` bei den Beispielen auf Ihrer Online-Lernplattform. Die Symbole stammen zum Teil aus der kostenlosen Icon-Bibliothek Open Icon Library. Sie finden diese Bibliothek im Internet auf der Seite <https://sourceforge.net/projects/openiconlibrary/>.

Weisen Sie dem `RibbonButton` über die Eigenschaft `SmallImageSource` in der Gruppe **Sonstiges** im Eigenschaftenfenster ein Symbol für das Speichern zu.

Im nächsten Schritt ergänzen wir ein Anwendungsmenü. Es soll Befehle zum Speichern, Öffnen und Erstellen von neuen Grafiken sowie einen Befehl zum Beenden der Anwendung enthalten.

Die XAML-Anweisungen für unser Anwendungsmenü mit den Einträgen **Neu**, **Öffnen**, **Speichern** und **Beenden** finden Sie im folgenden Code 2.4:

```
<Ribbon.ApplicationMenu>
  <RibbonApplicationMenu SmallImageSource="symbole/appmenu.png">
    <!-- bitte jeweils in einer Zeile eingeben -->
    <RibbonApplicationMenuItem x:Name="menuNeu" Header="Neu"
      ImageSource="symbole/neu.png"></RibbonApplicationMenuItem>
    <RibbonApplicationMenuItem x:Name="menuOeffnen"
      Header="Öffnen" ImageSource="symbole/oeffnen.png">
    </RibbonApplicationMenuItem>
    <RibbonApplicationMenuItem x:Name="menuSpeichern"
      Header="Speichern" ImageSource="symbole/save.png">
    </RibbonApplicationMenuItem>
    <RibbonApplicationMenuItem x:Name="menuBeenden"
      Header="Beenden"></RibbonApplicationMenuItem>
  </RibbonApplicationMenu>
</Ribbon.ApplicationMenu>
```

Code 2.4: Die XAML-Anweisungen für das Anwendungsmenü

Kommen wir jetzt zu den Registern und Gruppen im Menüband. Hier legen wir eine Gruppe **Zeichnen** und eine Gruppe **Format** an. In der Gruppe **Zeichnen** sollen sich drei Schaltflächen für die Auswahl der Zeichenwerkzeuge befinden und in der Gruppe **Format** eine Schaltfläche für das Öffnen des Dialogs zur Farbauswahl.

Die entsprechenden XAML-Anweisungen sehen dann so aus:

```
<RibbonTab Header="Start">
  <RibbonGroup Header="Zeichnen">
    <!-- bitte jeweils in einer Zeile eingeben -->
    <RibbonButton x:Name="buttonKreis"
      LargeImageSource="symbole/kreis.gif"
      Label="Kreis"></RibbonButton>
    <RibbonButton x:Name="buttonLinie"
      LargeImageSource="symbole/linie.gif"
      Label="Linie"></RibbonButton>
    <RibbonButton x:Name="buttonRechteck"
      LargeImageSource="symbole/rechteck.gif"
      Label="Rechteck"></RibbonButton>
  </RibbonGroup>
  <RibbonGroup Header="Format">
    <!-- bitte in einer Zeile eingeben -->
    <RibbonButton x:Name="buttonFarbauswahl"
      LargeImageSource="symbole/farbauswahl.gif"
      Label="Farbauswahl"></RibbonButton>
  </RibbonGroup>
</RibbonTab>
```

Code 2.5: Die XAML-Anweisungen für das Register **Start** mit den Gruppen **Zeichnen** und **Format**

Das komplette Menüband könnte jetzt so aussehen:

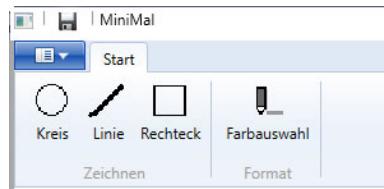


Abb. 2.2: Das Menüband für das Malprogramm

Fügen Sie dann noch ein Canvas-Steuerelement aus der Toolbox ein. Sie finden es in der Gruppe **Alle WPF-Steuerelemente**. Legen Sie es im Grid des Fensters ab und positionieren Sie es so, dass es links im Fenster unterhalb des Menübands angezeigt wird. Prüfen Sie danach, ob die Werte im ersten, zweiten und vierten Feld der Eigenschaft **Margin** in der Gruppe **Layout** jeweils auf 0 gesetzt sind. Dadurch werden die Abstände vom linken, rechten und unteren Rand festgelegt.

Hinweis:

Unter Umständen nimmt das Menüband den kompletten freien Platz im Fenster ein. Verkleinern Sie es dann, bevor Sie das Canvas-Steuerelement einfügen. Alternativ können Sie das neue Steuerelement auch über den XAML-Code einfügen.

Ändern Sie anschließend die Einträge in den Feldern für die Eigenschaften **Width** und **Height** in der Gruppe **Layout** in **Auto**, damit das Steuerelement immer den kompletten noch freien Raum im Fenster einnimmt. Setzen Sie dann gegebenenfalls noch die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** auf **Stretch**. Prüfen Sie dazu, ob jeweils das rechte Symbol bei den Eigenschaften markiert ist, und klicken Sie es – wenn erforderlich – an.

Im letzten Schritt müssen Sie noch dafür sorgen, dass das Canvas-Steuerelement einen Hintergrund bekommt. Sonst werden die Mausereignisse nicht ausgelöst. Klicken Sie dazu in der Gruppe **Pinsel** auf das Symbol **Pinsel mit Volltonfarbe** . Danach können Sie dann eine Farbe auswählen.

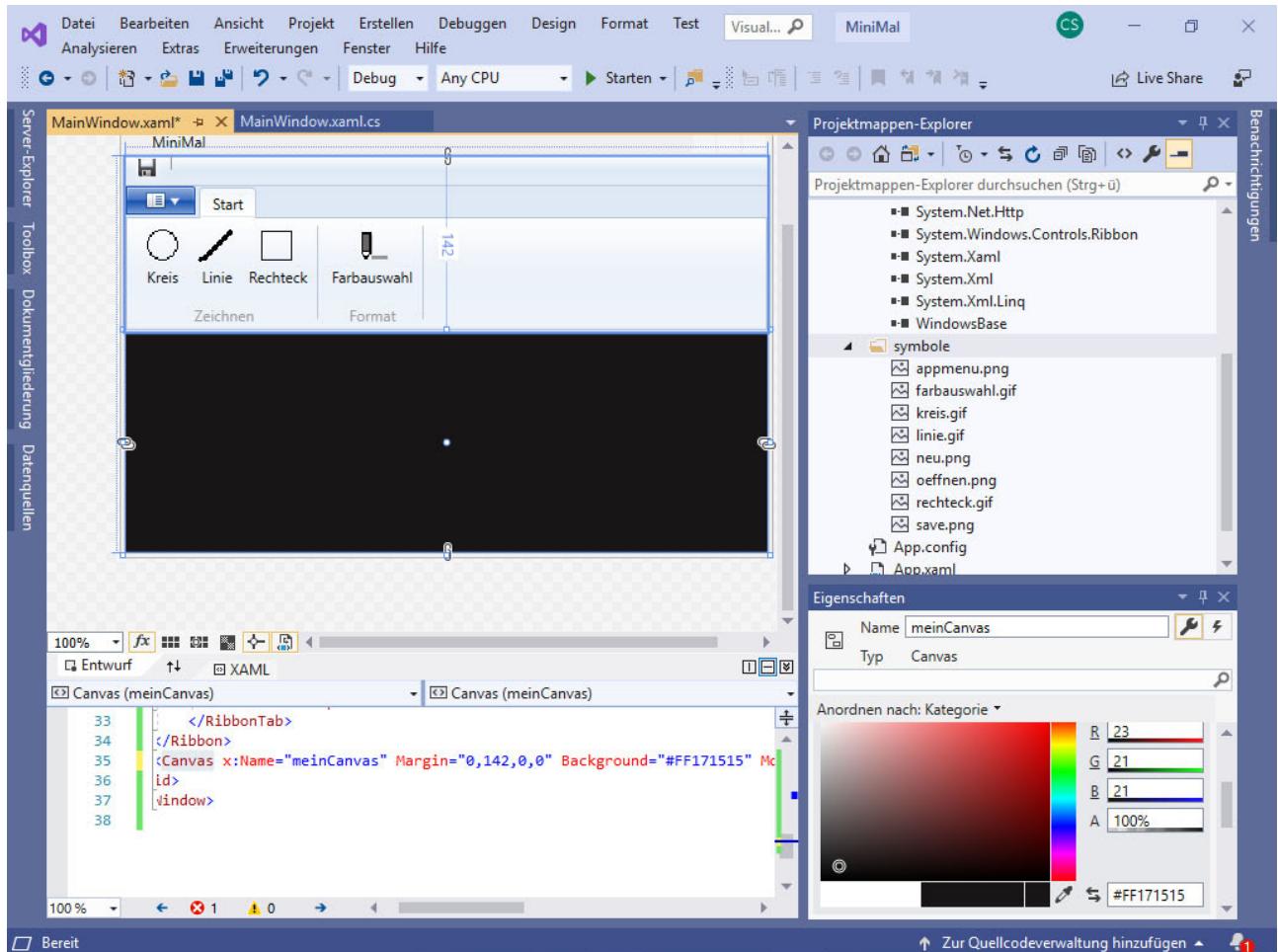


Abb. 2.3: Die Auswahl der Farbe

Am einfachsten geht das, wenn Sie mit der Maus in der Palette auf die gewünschte Farbe klicken. Sie können aber auch den RGB-Wert der Farbe in die drei Felder rechts neben der Palette eintragen.

Beim **RGB-Modell** wird eine Farbe über die drei Parameter **Rot**, **Grün** und **Blau** festgelegt. Die Werte geben dabei den Anteil der jeweiligen Farbe an und müssen im Bereich zwischen 0 und 255 liegen.



Wir benutzen in unserem Beispiel als Hintergrund Weiß. Sie können aber auch eine beliebige andere Farbe auswählen.

Damit Sie das Steuerelement gleich gezielt ansprechen können, sollten Sie ihm noch einen sprechenden Namen geben. Wir benutzen den Bezeichner `meinCanvas`.

2.3 Die Zeichenoperationen

Kommen wir jetzt zu den Zeichenoperationen. Hier erstellen wir die Methoden für das Drücken und Loslassen der linken Maustaste sowie für das Anklicken der Schaltflächen. Beim Drücken der linken Maustaste speichern wir nur die aktuelle Position in einem Feld. Beim Loslassen zeichnen wir dann abhängig vom Wert eines Feldes unterschiedliche Figuren. Den Wert dieses Feldes setzen wir beim Anklicken einer der Schaltflächen im Menüband.

Der Code für die Vereinbarung der Felder und die Methoden sieht dann so aus:

```
//für die Farbe
SolidColorBrush farbe;
//für die Position
Point start;
//für das Werkzeug
int werkzeug;

//die eigenen Methoden
//Sie erstellen die Objekte und fügen sie ein
//Breite und Höhe bzw. Endpunkt werden übergeben

//für das Rechteck
void Rechteck(int breite, int hoehe)
{
    try
    {
        //das Rechteck erzeugen
        Rectangle rechteck = new Rectangle();
        //die Eigenschaften setzen
        //die linke obere Ecke
        Canvas.SetLeft(rechteck, start.X);
        Canvas.SetTop(rechteck, start.Y);
        //Breite und Höhe
        rechteck.Width = breite;
        rechteck.Height = hoehe;
        rechteck.Stroke = farbe;
        //und hinzufügen
        meinCanvas.Children.Add(rechteck);
    }
    catch
    {
        MessageBox.Show("Es hat ein Problem gegeben");
    }
}

//die Methode zeichnet einen Kreis
//die Breite und Höhe des umgebenden
//Rechtecks werden als Parameter übergeben
void Kreis(int breite, int hoehe)
{
    try
    {
        //den Kreis erzeugen
        Ellipse kreis = new Ellipse();
```

```
//die Eigenschaften setzen
//die linke obere Ecke
Canvas.SetLeft(kreis, start.X);
Canvas.SetTop(kreis, start.Y);
//Breite und Höhe
kreis.Width = breite;
kreis.Height = hoehe;
kreis.Stroke = farbe;
//und hinzufügen
meinCanvas.Children.Add(kreis);
}
catch
{
    MessageBox.Show("Es hat ein Problem gegeben");
}
}

//die Methode zeichnet eine Linie
//Der Endpunkt wird als Parameter übergeben
void Linie(int x2, int y2)
{
    //eine neue Linie erzeugen
    Line linie = new Line();
    //die Eigenschaften setzen
    linie.X1 = start.X;
    linie.Y1 = start.Y;
    linie.X2 = x2;
    linie.Y2 = y2;
    linie.Stroke = farbe;
    //und hinzufügen
    meinCanvas.Children.Add(linie);
}

public MainWindow()
{
    InitializeComponent();
    //die Standardfarbe ist Schwarz
    farbe = Brushes.Black;
    //das Standardwerkzeug ist die Linie
    werkzeug = 1;
}

//die Methoden setzen jeweils den Wert für das entsprechende
//Werkzeug
private void ButtonLinie_Click(object sender,
RoutedEventArgs e)
{
    werkzeug = 1;
}

private void ButtonKreis_Click(object sender,
RoutedEventArgs e)
{
    werkzeug = 2;
}
```

```

private void ButtonRechteck_Click(object sender,
RoutedEventArgs e)
{
    werkzeug = 3;
}

private void MeinCanvas_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{
    //die Koordinate als Startpunkt speichern
    start = e.GetPosition(meinCanvas);
}

private void MeinCanvas_MouseLeftButtonUp(object sender,
MouseButtonEventArgs e)
{
    //wenn die Taste losgelassen wird, zeichnen wir von der
    //alten zur aktuellen Position
    //je nach Werkzeug wird ein anderes Objekt gezeichnet
    //die Linie
    if (werkzeug == 1)
        //bitte in einer Zeile eingeben
        Linie((int)e.GetPosition(meinCanvas).X,
              (int)e.GetPosition(meinCanvas).Y);
    //ein Kreis
    if (werkzeug == 2)
        //bitte in einer Zeile eingeben
        Kreis((int)(e.GetPosition(meinCanvas).X - start.X),
              (int)(e.GetPosition(meinCanvas).Y - start.Y));
    //das Rechteck
    if (werkzeug == 3)
        //bitte in einer Zeile eingeben
        Rechteck((int)(e.GetPosition(meinCanvas).X - start.X),
                  (int)(e.GetPosition(meinCanvas).Y - start.Y));
}
}

```

Code 2.6: Die Zeichenoperationen

Zunächst einmal vereinbaren wir Felder für die Position, das aktuelle Werkzeug und die aktuelle Farbe. Für die Farbe verwenden wir dabei den Typ `SolidColorBrush`.

Im Konstruktor setzen wir die Standardfarbe und das Standardwerkzeug. Auch das ist noch nicht besonders spannend.

Dann folgen drei Methoden für das Anklicken der Schaltflächen mit den Zeichenwerkzeugen. Sie setzen lediglich den Wert des Feldes `werkzeug`.

In den Methoden `MeinCanvas_MouseLeftButtonDown()` und `MeinCanvas_MouseLeftButtonUp()` verarbeiten wir die Mausereignisse. Beim Drücken der linken Maustaste speichern wir die aktuelle Position im Feld `start`. Beim Loslassen wird gezeichnet. Hier rufen wir abhängig vom ausgewählten Werkzeug entweder die Methode `Rechteck()`, die Methode `Kreis()` oder die Methode `Linie()` auf.

Als Argumente übergeben wir dabei die Breite und Höhe beziehungsweise die Endposition. Die Breite und Höhe berechnen wir aus der Differenz zwischen der Startposition und der aktuellen Position.

In den eigentlichen Methoden zum Zeichnen rufen wir zunächst die Startposition aus dem Feld `start` ab. Die Werte setzen wir ja beim Drücken der Maustaste. Beim Rechteck und beim Kreis können wir diese Positionen allerdings nicht direkt zuweisen. Wir müssen einen kleinen Umweg über die Methoden `SetLeft()` und `SetTop()` der Klasse `Canvas` gehen. Diese Methoden erwarten das Objekt, das positioniert werden soll, und die Position.

Hinweis:

Bei den Kreisen und Rechtecken funktioniert das Zeichnen nur, wenn Sie die Figur von oben nach unten beziehungsweise von links nach rechts malen. Andernfalls ist die Höhe beziehungsweise Breite negativ und das Programm löst eine Ausnahme aus. Daher haben wir die Anweisungen in einer einfachen `try ... catch`-Konstruktion verpackt. Wenn Sie das stört, müssten Sie die Anweisungen noch so erweitern, dass gegebenenfalls die Start- und Endpunkte getauscht werden.

Jetzt fehlt noch die Methode zum Auswählen einer neuen Farbe. Da die WPF keinen eigenen Dialog zur Farbauswahl kennt, benutzen wir den Standarddialog von Windows. Dazu müssen wir allerdings zunächst einmal Verweise auf die Dateien `System.Windows.Forms.dll` und `System.Drawing.dll` zum Projekt hinzufügen. Das erfolgt wie bei den anderen Verweisen auch über die Funktion **Projekt/Verweis hinzufügen** ...

Danach können Sie dann über die Klasse `System.Windows.Forms.ColorDialog` einen neuen Dialog zur Farbauswahl erzeugen und anzeigen lassen. Leider passt die Farbe, die der Dialog liefert, nicht direkt zum Typ `SolidColorBrush`. Wir müssen den Wert erst mit der Methode `FromArgb()` der Klasse `Color` in die einzelnen Bestandteile für das RGB-Modell zerlegen und in einer Variablen vom Typ `Color` zwischenspeichern. Diese Farbe können wir dann dem Feld `farbe` zuweisen.

Ausprogrammiert sehen die entsprechenden Anweisungen so aus:

```
private void ButtonFarbAuswahl_Click(object sender,
RoutedEventArgs e)
{
    //einen neuen Standarddialog von Windows zur Farbauswahl
    //erzeugen
    //bitte jeweils in einer Zeile eingeben
    System.Windows.Forms.ColorDialog farbAuswahl = new
    System.Windows.Forms.ColorDialog();
    //wurde der Dialog über OK geschlossen
    if (farbAuswahl.ShowDialog() ==
        System.Windows.Forms.DialogResult.OK)
    {
        //dann die Farbe zuweisen
        //das geht nur über einen Umweg
        //bitte in einer Zeile eingeben
        Color farbeWPF = Color.FromArgb(farbAuswahl.Color.A,
            farbAuswahl.Color.R, farbAuswahl.Color.G,
            farbAuswahl.Color.B);
```

```

        farbe = new SolidColorBrush(farbeWPF) ;
    }
}

```

Code 2.7: Die Farbauswahl

Übernehmen Sie jetzt die Anweisungen aus den vorigen Codes, speichern Sie die Änderungen und testen Sie das Programm.

2.4 Die Dateioperationen

Kommen wir nun zu den Dateioperationen.

Beginnen wir mit dem Anlegen einer neuen Datei. Hier löschen wir einfach die Zeichenfläche – also den Inhalt des Containers – mit der Anweisung `meinCanvas.Children.Clear()`.

Für das Laden und Speichern verwenden wir die Klassen `XamlWriter` und `XamlReader` aus dem Namensraum `System.Windows.Markup`. Mit ihnen können wir die Daten der Objekte auf der Zeichenfläche einfach im XAML-Format speichern und auch wieder zurücklesen. Für die Dateiauswahl verwenden wir beim Öffnen und Speichern wieder die Standarddialoge von Windows. Als Erweiterung benutzen wir `.mml` als Abkürzung für MiniMal.

Die Methode zum Speichern ist schnell erstellt. Sie sieht so aus:

```

void Speichern()
{
    //einen Speicherndialog erzeugen
    //bitte in einer Zeile eingeben
    Microsoft.Win32.SaveFileDialog speichernDialog = new
    Microsoft.Win32.SaveFileDialog();
    //die Eigenschaften für den Dialog setzen
    speichernDialog.Filter = "MiniMal-Dateien|*.mml";
    speichernDialog.FileName = string.Empty;
    //wurde der Dialog über Speichern geschlossen?
    //dann die Datei unter dem angegebenen Namen speichern
    if (speichernDialog.ShowDialog() == true)
    {
        //bitte jeweils in einer Zeile eingeben
        using (System.IO.FileStream dateiStream = new
            System.IO.FileStream(speichernDialog.FileName,
            System.IO.FileMode.Create))
        {
            System.Windows.Markup.XamlWriter.Save(meinCanvas,
                dateiStream);
        }
    }
}

```

Code 2.8: Das Speichern der Datei

Zur Erinnerung:

Mit der `using`-Anweisung sorgen wir dafür, dass die Datei wieder geschlossen wird und die Ressourcen freigegeben werden.



Wirklich neu in dem vorigen Code ist nur die Anweisung

```
System.Windows.Markup.XamlWriter.Save(meinCanvas, dateiStream);
```

Hier speichern wir den kompletten Container `meinCanvas` mitsamt allen untergeordneten Elementen über die Methode `Save()` der Klasse `System.Windows.Markup.XamlWriter`.

Das Zurücklesen ist leider nicht ganz so einfach. Da wir den gesamten Container gespeichert haben, müssen wir den alten Container vor dem Laden der Datei entfernen und durch den Container mit den Daten aus der Datei ersetzen.

Vergeben Sie dazu bitte im ersten Schritt einen Namen an das Grid im Fenster der Anwendung. Sonst können wir den Container gleich nicht entfernen. Wir benutzen in unserem Beispiel den Namen `meinGrid`. Die Anweisungen zum Laden sehen dann erst einmal so aus:

```
void Laden()
{
    //einen Öffnendialog erzeugen
    //bitte in einer Zeile eingeben
    Microsoft.Win32.OpenFileDialog oeffnenDialog = new
    Microsoft.Win32.OpenFileDialog();
    //die Eigenschaften für den Dialog setzen
    oeffnenDialog.Filter = "MiniMal-Dateien|*.mml";
    oeffnenDialog.FileName = string.Empty;
    //wurde der Dialog über Öffnen geschlossen?
    //dann die Datei öffnen
    if (oeffnenDialog.ShowDialog() == true)
    {
        //bitte in einer Zeile eingeben
        using (System.IO.FileStream dateiStream = new
            System.IO.FileStream(oeffnenDialog.FileName,
            System.IO FileMode.Open))
        {
            //die alte Zeichenfläche entfernen
            meinGrid.Children.Remove(meinCanvas);
            //und laden und wieder einfügen
            //bitte in einer Zeile eingeben
            meinCanvas = System.Windows.Markup.XamlReader.Load
                (dateiStream) as Canvas;
            meinGrid.Children.Add(meinCanvas);
        }
    }
}
```

Code 2.9: Das Laden der Datei

Mit der Anweisung

```
meinGrid.Children.Remove(meinCanvas);
```

entfernen wir zunächst den alten Container mit der Methode `Children.Remove()`.

Mit der Anweisung

```
meinCanvas = System.Windows.Markup.XamlReader.Load(dateistream)
as Canvas;
```

laden wir dann den Inhalt aus der Datei, konvertieren ihn mit dem Operator `as` – wenn möglich – in den Typ `Canvas` und weisen ihn `meinCanvas` zu.



Der Operator `as` prüft, ob die Umwandlung in den angegebenen Typ möglich ist, und liefert das Ergebnis der Umwandlung – wenn möglich – auch sofort zurück.

Damit der Container wieder angezeigt wird, fügen wir ihn mit der Anweisung

```
meinGrid.Children.Add(meinCanvas);
```

wieder zum Grid hinzu.

Lassen Sie die Methoden zum Speichern und Laden jetzt beim Anklicken der entsprechenden Schaltflächen ausführen und testen Sie die Erweiterungen. Dabei werden Sie allerdings eine unangenehme Überraschung erleben. Zwar werden die Shapes wieder geladen, aber Sie können nicht mehr zeichnen. Das liegt daran, dass die Eventhandler leider nicht mitgespeichert werden. Sie müssen die Methoden für das Drücken und Loslassen der Maustaste also wieder neu zuweisen. Ergänzen Sie dazu nach dem Hinzufügen des Containers die folgenden Anweisungen:

```
meinCanvas.MouseLeftButtonDown += new
MouseButtonEventHandler(meinCanvas_MouseLeftButtonDown);
meinCanvas.MouseLeftButtonUp += new
MouseButtonEventHandler(meinCanvas_MouseLeftButtonUp);
```

Lassen Sie abschließend noch beim Anklicken des Menüeintrags **Beenden** das Fenster der Anwendung über `Close()` schließen.

Setzen wir jetzt auch noch auf diese Anwendung ein kleines i-Tüpfelchen: Wir lassen den Mauszeiger bei den Zeichenoperationen als Pfeil darstellen. Dazu weisen Sie der Eigenschaft `Cursor` von `meinCanvas` beim Drücken der linken Maustaste den Wert `Cursors.Pen` zu. Die entsprechende Anweisung sieht so aus:

```
meinCanvas.Cursor = Cursors.Pen;
```

Um beim Loslassen der Maustaste wieder den Pfeil als Mauscursor zu verwenden, benutzen Sie die Anweisung:

```
meinCanvas.Cursor = Cursors.Arrow;
```

Damit wollen wir die Arbeit an dem kleinen Malprogramm beenden. Wenn Sie Lust haben, können Sie es ja noch selbst erweitern – zum Beispiel um Sicherheitsabfragen bei den Dateioperationen oder um weitere Einstellungen für die Zeichenwerkzeuge.

Zusammenfassung

Die WPF kennt verschiedene Mausereignisse. Neben dem Klicken können Sie auch auf das Drücken und Loslassen einer Maustaste sowie auf Bewegungen der Maus reagieren.

Die Klassen für die Mausereignisse kennen Methoden und Eigenschaften, die Ihnen weitere Informationen zum Ereignis liefern – zum Beispiel die gedrückte Maustaste oder die Position der Maus.

Shapes, die Sie im Code erstellen, müssen Sie selbst als untergeordnetes Element zu einem Steuerelement oder Container hinzufügen.

Die Hintergrundfarbe für ein Steuerelement legen Sie über den Pinsel fest.

Für die Farbauswahl kennt die WPF kein eigenes Steuerelement. Sie müssen den Standarddialog von Windows benutzen und entsprechende Verweise zum Projekt hinzufügen.

Mit den Klassen `XamlWriter` und `XamlReader` können Sie Objekte im XAML-Format in einer Datei speichern und wieder zurücklesen. Die Verarbeitung erfolgt über einen Stream. Es werden allerdings nicht alle Eigenschaften gespeichert. Eventhandler gehen zum Beispiel verloren und müssen wieder neu zugewiesen werden.

Über die Eigenschaft `Cursor` können Sie den Mauszeiger verändern.

Aufgaben zur Selbstüberprüfung

- 2.1 Welches Ereignis wird beim Loslassen der linken Maustaste ausgelöst?

- 2.2 Wie können Sie im Ereignis **MouseDown** feststellen, ob die linke Maustaste gedrückt wurde? Formulieren Sie bitte eine entsprechende Anweisung.

- 2.3 Mit welcher Methode können Sie in einem Mausereignis die aktuelle Position der Maus ermitteln? Welchen Typ liefert Ihnen die Methode? Was müssen Sie bei der Methode als Argument übergeben?

- 2.4 Sie haben im Code ein Shape mit dem Namen `kreis` erstellt. Es wird allerdings beim Ausführen des Programms nicht angezeigt. Was haben Sie wahrscheinlich vergessen?

- 2.5 Mit welchen Anweisungen können Sie die Position eines Shapes im Code setzen? Welche Argumente müssen Sie angeben?

- 2.6 Sie benutzen in einer WPF-Anwendung den Standarddialog von Windows zur Farbauswahl. Können Sie die ausgewählte Farbe direkt einem Shape zuweisen? Wenn nicht: Welche Schritte sind erforderlich?

- 2.7 Mit welcher Methode löschen Sie alle untergeordneten Elemente in einem Container vom Typ `Canvas`?

- 2.8 Sie wollen ein Steuerelement von Typ `Canvas` mit allen untergeordneten Elementen in einer XAML-Datei speichern. Einen entsprechenden Stream haben Sie bereits angelegt und einer Variablen `meinStream` zugewiesen. Formulieren Sie die Anweisung für das Speichern. Für das Steuerelement können Sie einen beliebigen Namen verwenden.

3 Animationen und 3D-Grafiken

In diesem Kapitel beschäftigen wir uns mit Animationen. Außerdem zeigen wir Ihnen kurz, wie Sie mit der WPF 3D-Grafiken erstellen können.

3.1 Animationen

Anders als zum Beispiel bei einer Windows-Forms-Anwendung mit GDI+ brauchen Sie bei einer WPF-Anwendung für eine Animation keinen Timer. Sie können die Werte von bestimmten Eigenschaften eines Objekts in einer bestimmten Zeitspanne gezielt verändern und so zum Beispiel dafür sorgen, dass ein Kreis immer größer wird.

Die Animationen erfolgen dabei allerdings nicht direkt über die Eigenschaften des Objekts, sondern über die Abhängigkeitseigenschaften. Diese Eigenschaften haben häufig denselben Namen wie die „normale“ Eigenschaft, gefolgt von dem Zusatz `Property`. Die Abhängigkeitseigenschaft für die Breite eines Rechtecks lautet dann zum Beispiel `Rectangle.WidthProperty`.



Bei einer Abhängigkeitseigenschaft – im Fachjargon auch *dependency property* genannt – hängt der Wert von anderen Quellen ab. Außerdem kann eine Abhängigkeitseigenschaft melden, dass sich ihr Wert verändert hat.

Hinweis:

Neben der Basisanimation über das Ändern von Eigenschaften über einen bestimmten Zeitraum unterstützt die WPF auch noch andere Animationsformen. Damit wollen wir uns hier aber nicht weiter beschäftigen. Mehr zu dem Thema finden Sie zum Beispiel in der Hilfe.

Abhängig von den Eigenschaften, die Sie in der Animation verändern wollen, müssen Sie den passenden Typ verwenden. Wenn Sie zum Beispiel eine Eigenschaft vom Typ `double` verändern, benutzen Sie die Klasse `DoubleAnimation`. Für eine Eigenschaft vom Typ `int` dagegen verwenden Sie die Klasse `Int32Animation`.

Hinweis:

Die Namen der Klassen orientieren sich an den Namen der .NET Framework-Typen.

Die eigentliche Bewegung steuern Sie dann über die Eigenschaften `From` und `To` der Animation. Die Eigenschaft `From` legt dabei den Ausgangspunkt der Animation fest und die Eigenschaft `To` den Endpunkt.

Gestartet wird die Animation mit der Methode `BeginAnimation()` für das Objekt. Als Argumente übergeben Sie dabei die Abhängigkeitseigenschaft, die verändert werden soll, und die Instanz der Klasse für die Animation.

Das folgende Fragment verdoppelt in einer Animation zum Beispiel die Größe eines Kreises:

```
DoubleAnimation animation = new DoubleAnimation();
animation.From = kreis.Width;
animation.To = kreis.Width * 2;
kreis.BeginAnimation(WidthProperty, animation);
kreis.BeginAnimation(HeightProperty, animation);
```

Code 3.1: Eine erste einfache Animation

Hinweis:

Die Klassen für die Animationen liegen im Namensraum `System.Windows.Media.Animation`. Bitte binden Sie diesen Namensraum für den vorigen Code und auch allen weiteren Beispiele für die Animationen über eine entsprechende `using`-Anweisung ein.

Schauen wir uns die einzelnen Anweisungen noch einmal der Reihe nach an:

Mit der Anweisung

```
DoubleAnimation animation = new DoubleAnimation();
```

wird eine Instanz `animation` der Klasse `DoubleAnimation` erzeugt.

Mit den beiden Anweisungen

```
animation.From = kreis.Width;
animation.To = kreis.Width * 2;
```

setzen wir die Eigenschaften `From` und `To` für die Animation. Ausgangspunkt ist die aktuelle Breite des umgebenden Rechtecks für den Kreis. Endpunkt soll die doppelte Größe sein.

Hinweis:

Der Bezeichner `kreis` steht für eine Ellipse, die wir direkt aus der Toolbox in das Fenster gesetzt haben.

Die beiden Anweisungen

```
kreis.BeginAnimation(WidthProperty, animation);
kreis.BeginAnimation(HeightProperty, animation);
```

schließlich starten die Animation für die Breite und die Höhe.

Hinweis:

Die Eigenschaften heißen vollständig `Ellipse.WidthProperty` beziehungsweise `Ellipse.HeightProperty`. Visual Studio schlägt aber selbst vor, den Zugriff zu vereinfachen und den Namen der Klasse zu entfernen.

Wir animieren den Kreis in dem Beispiel ausgehend von der oberen linken Ecke des Kreises. Er wird daher auch nicht um den Mittelpunkt vergrößert, sondern wandert immer weiter nach unten rechts.

Die Eigenschaften `From` und `To` müssen Sie nicht unbedingt setzen. In unserem Beispiel könnten wir auf das ausdrückliche Setzen von `From` auch verzichten. Dann wird automatisch der aktuelle Wert der entsprechenden Eigenschaft verwendet – in unserem Beispiel also die Breite des Kreises. Das gilt entsprechend, wenn Sie die Eigenschaft `To` nicht setzen. Dann wird ebenfalls der aktuelle Wert der entsprechenden Eigenschaft benutzt.



Nicht jedes Objekt besitzt Standardwerte für Eigenschaften. Eine Schaltfläche, die Sie aus der Toolbox einfügen, hat zum Beispiel keinen gesetzten Wert für die Eigenschaft `Height`. Wenn Sie dann eine Animation für diese Eigenschaft durchführen wollen, wird eine Ausnahme ausgelöst. Sie können die Werte aber jederzeit per Hand zuweisen.

Das Tempo der Animation wird in unserem Beispiel durch das Animationssystem selbst gesteuert. Sie können über die Eigenschaft `Duration` aber auch selbst eine Dauer vorgeben. Das geht am einfachsten, wenn Sie mit der Methode `Parse()` der Struktur `TimeSpan`⁵ aus einer Zeichenkette eine Zeitangabe vom Typ `Duration` erzeugen. Diese Zeichenfolge enthält erst die Anzahl der Tage, dann die Stunden, die Minuten, die Sekunden und abschließend die Millisekunden. Die Tage und die Millisekunden werden dabei durch einen Punkt von den anderen Angaben getrennt, die Stunden, Minuten und Sekunden durch einen Doppelpunkt. Die Zeichenkette `4.5:0:10.20` steht dann für 4 Tage, 5 Stunden, 10 Sekunden und 20 Millisekunden.

Die Angaben der Tage und Millisekunden können Sie auch weglassen. Bei den anderen Werten müssen Sie aber den Wert 0 ausdrücklich setzen – auch wenn Sie eigentlich keine Angaben benötigen. Die Zeichenkette `0:0:30` steht zum Beispiel für 30 Sekunden.

Sie können die einzelnen Werte auch direkt an den Konstruktor der Struktur `TimeSpan` übergeben. Für ein Zeitintervall, das 30 Sekunden umfasst, lautet die Anweisung dann `new TimeSpan(0, 0, 30)`. Hier werden erst die Stunden, dann die Minuten und anschließend die Sekunden angegeben. Sie können aber auch alle fünf Werte an den Konstruktor übergeben – also die Tage, die Stunden, die Minuten, die Sekunden und die Millisekunden.

Die Anweisung

```
animation.Duration = new Duration(TimeSpan.Parse("0:0:30"));
```

setzt also die Dauer einer Animation auf 30 Sekunden.

Über die Eigenschaft `BeginTime` können Sie bei Bedarf auch die Startzeit angeben. Dazu weisen Sie ebenfalls einen Wert vom Typ `TimeSpan` zu. Die Anweisung

```
animation.BeginTime = TimeSpan.Parse("0:0:1");
```

setzt also die Startzeit auf eine Sekunde.

5. `TimeSpan` lässt sich mit „Zeitspanne“ übersetzen.

Wenn Sie die Dauer und die Startzeit angeben, wird erst abgewartet und dann die Animation in der Zeitspanne ausgeführt, die Sie bei der Dauer angegeben haben. Bei einer Startzeit von einer Sekunde und einer Dauer von zwei Sekunden beträgt die Gesamtdauer also drei Sekunden.



Um eine Animation automatisch wieder rückwärts laufen zu lassen, setzen Sie die Eigenschaft `AutoReverse`⁶ auf `true`. Über die Eigenschaft `RepeatBehavior`⁷ können Sie auch die Anzahl der Wiederholungen setzen. Dabei geben Sie entweder die Anzahl der Wiederholungen als Zahl an oder eine Zeitspanne vom Typ `TimeSpan`. Die Angaben müssen Sie dabei beim Konstruktor der Struktur `RepeatBehavior` machen.

Die beiden folgenden Anweisungen sorgen zum Beispiel dafür, dass die Animation rückwärts läuft und viermal wiederholt wird:

```
//die Animation soll rückwärts laufen
animation.AutoReverse = true;
//und insgesamt viermal wiederholt werden
animation.RepeatBehavior = new RepeatBehavior(4);
```

Code 3.2: Zurücklaufen und Wiederholung einer Animation

Wenn Sie die Anzahl der Wiederholungen über einen numerischen Wert setzen, gilt die Dauer für einen Durchlauf der Animation in eine Richtung.



Interessant sind auch noch die Eigenschaften `AccelerationRatio` und `DecelerationRatio`⁸ einer Animation. Sie legen fest, wie stark zu Beginn der Animation beschleunigt werden soll beziehungsweise wie stark am Ende der Animation abgebremst werden soll. In der Standardeinstellung ist der Wert der beiden Eigenschaften 0. Das entspricht einer konstanten Geschwindigkeit über die gesamte Animation. Um die Geschwindigkeit anzupassen, vergeben Sie einen Wert zwischen 0 und 1.

Bitte beachten Sie:



Die Summe der beiden Werte darf nicht größer werden als 1. Wenn Sie also die Eigenschaft `AccelerationRatio` auf 0.5 setzen, kann der Wert für die Eigenschaft `DecelerationRatio` maximal 0.5 betragen.

Wenn die Summe der beiden Werte größer ist als 1, wird beim Ausführen der Animation eine Ausnahme ausgelöst.

Im praktischen Einsatz finden Sie die beiden Eigenschaften im folgenden Code. Hier wird ein Kreis von links nach rechts und wieder zurück bewegt. Die Animation wird dabei sowohl beschleunigt als auch abgebremst.

6. `AutoReverse` lässt sich mit „automatisch umdrehen“ übersetzen.

7. `Repeat behavior` bedeutet übersetzt „Verhalten bei Wiederholung“.

8. `Acceleration` bedeutet übersetzt „Beschleunigung“ und `Deceleration` „Verzögerung, Verlangsamung“. `Ratio` lässt sich mit „Verhältnis“ übersetzen.

```
DoubleAnimation animation = new DoubleAnimation();
//Start und Ende setzen
animation.From = Canvas.GetLeft(kreis);
animation.To = Canvas.GetLeft(kreis) + 350;
//es soll zwei Sekunden dauern und wieder zurücklaufen
animation.Duration = new Duration(TimeSpan.Parse("0:0:2"));
animation.AutoReverse = true;
//Beschleunigen und Abbremsen
animation.AccelerationRatio = 0.5;
animation.DecelerationRatio = 0.5;
//die Eigenschaft animieren
kreis.BeginAnimation(LeftProperty, animation);
```

Code 3.3: Eine Animation mit Beschleunigung und Abbremsen

Hinweis:

Damit Sie die Position eines Objekts animieren können, müssen Sie es in einen Container vom Typ `Canvas` setzen. Sonst haben Sie keinen Zugriff auf Eigenschaften wie `LeftProperty` zum Verändern der Position.



Die vorigen Codes finden Sie auch im Projekt **EinfacheAnimation** im Download-Bereich Ihrer Online-Lernplattform.

3.2 3D-Grafiken

Neben zweidimensionalen Grafiken können Sie mit der WPF auch dreidimensionale Grafiken erstellen. Hier verfügen die Objekte nicht nur über X- und Y-Koordinaten, sondern auch über eine Z-Koordinate.

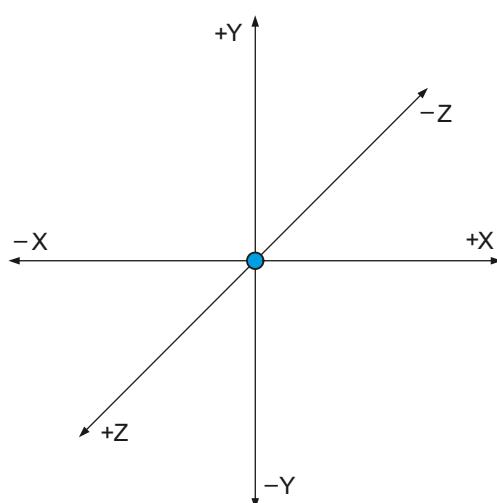


Abb. 3.1: Ein dreidimensionales Koordinatensystem (der markierte Punkt ist der 0-Punkt)



Die Werte für die einzelnen Achsen können auch negativ werden. Dann liegt das Objekt auf der entsprechenden Achse hinter dem Punkt 0, 0, 0.

Um dreidimensionale Grafiken zu erstellen, legen Sie im ersten Schritt ein Objekt vom Typ `Viewport3D` an. Dieses Objekt dient als eine Art Fenster für die dreidimensionale Ansicht und setzt die Objekte in der richtigen Form um. Die Mitte des Viewports bildet auch den Mittelpunkt des Koordinatensystems.

In das Fenster setzen Sie eine Kamera, eine Lichtquelle und die Objekte selbst. Damit die Objekte das Licht reflektieren, müssen Sie ihnen noch ein Material zuweisen. Andernfalls bleiben sie unsichtbar.

In der Toolbox suchen Sie die Elemente für eine 3D-Grafik allerdings vergebens. Sie müssen die Szene entweder im XAML-Code erstellen oder über C#-Anweisungen.

Die folgenden XAML-Anweisungen definieren zum Beispiel einen Viewport mit einer Kamera und einer Lichtquelle.

```
<Viewport3D>
    <Viewport3D.Camera>
        <!-- bitte in einer Zeile eingeben -->
        <PerspectiveCamera Position="-100, 100, 100"
            LookDirection="100, -100, -100" />
    </Viewport3D.Camera>
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <Model3DGroup>
                <DirectionalLight Color="White" Direction="0, -2, -1" />
            </Model3DGroup>
        </ModelVisual3D.Content>
    </ModelVisual3D>
</Viewport3D>
```

Code 3.4: Ein Viewport mit Kamera und Licht

Der Viewport wird dabei über das Tag `<Viewport3D>` beschrieben. Es bildet das äußere Element. Über das Tag `<Viewport3D.Camera>` wird anschließend die Kamera für den Viewport erstellt. In unserem Beispiel benutzen wir eine perspektivische Kamera vom Typ `PerspectiveCamera`. Sie befindet sich an der Position $-100, 100, 100$ – also links vorn oberhalb des Nullpunkts. Die Blickrichtung der Kamera geht auf den Punkt $100, -100, -100$ – also schräg von vorne nach hinten und von oben nach unten.

Die Koordinaten werden in der Reihenfolge X, Y, Z angegeben.



Mit dem Tag `<ModelVisual3D>` wird dann das Modell eingeleitet. Es steht – wenn Sie so wollen – für den Inhalt der Szene. Was das Modell genau enthält, legen Sie im Tag `<ModelVisual3D.Content>` fest. In unserem Beispiel wird mit dem Tag `<Model3DGroup>` eine Gruppe beschrieben, die im vorigen Code aber nur aus einem Element besteht – nämlich dem Licht. Wir verwenden hier ein gerichtetes Licht vom Typ `DirectionalLight` als Lichtquelle. Es hat die Farbe Weiß und scheint in die Richtung $0, -2, -1$ – also von vorn nach hinten und von oben nach unten.

Probieren Sie einmal aus, welche Wirkung die Anweisungen haben. Legen Sie ein neues WPF-Projekt an und übernehmen Sie die XAML-Anweisungen aus dem obigen Code. Es wird allerdings nicht zu sehen sein. Denn das Wichtigste fehlt in unserer Szene noch – ein grafisches Objekt.

Und um das zu erstellen, müssen Sie recht viel Aufwand betreiben. Denn jedes grafische dreidimensionale Objekt wird aus Dreiecken erstellt. Für ein Quadrat müssen Sie also zwei Dreiecke beschreiben, die aneinander liegen. Ein kompletter Würfel besteht dann aus insgesamt 12 Dreiecken – nämlich jeweils zwei für die sechs Seiten.

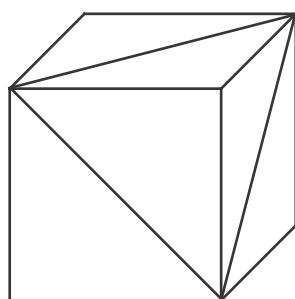


Abb. 3.2: Ein Würfel aus Dreiecken

Die Beschreibung der Dreiecke erfolgt dabei über die drei Punkte, die das Dreieck bilden, und die Reihenfolge, in der diese Punkte verbunden werden sollen. Schauen wir uns dazu ein Beispiel an.

Die Punkte für zwei Dreiecke, die ein Quadrat bilden, können Sie so beschreiben:

Punkt 1: 0, 0, 30

Punkt 2: 30, 0, 30

Punkt 3: 30, 30, 30

Punkt 4: 0, 30, 30

Da die Dreiecke aneinanderstoßen, teilen sie sich jeweils zwei Punkte. Wir benötigen also nicht sechs Punkte, sondern kommen mit vier aus.

Um die beiden Dreiecke zu erstellen, müssen die Punkte in der folgenden Reihenfolge verbunden werden:

Dreieck 1: von 1 nach 2 und dann nach 3

Dreieck 2: von 3 nach 4 und dann nach 1

Das Ergebnis sieht so aus:

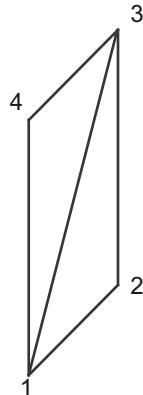


Abb. 3.3: Ein Quadrat aus Dreiecken (das Quadrat bildet das Seitenteil eines Würfels)

Die vollständigen XAML-Anweisungen für das Erstellen eines Würfels mit einem blau-violetten Material finden Sie im folgenden Code:

```
<GeometryModel3D>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions= "0,0,30 30,0,30 30,30,30
                           0,30,30 30,0,0 0,0,0
                           0,30,0 30,30,0 30,0,30
                           30,0,0 30,30,0 30,30,30
                           0,0,0 0,0,30 0,30,30
                           0,30,0 0,30,30 30,30,30
                           30,30,0 0,30,0 30,0,30
                           0,0,30 0,0,0 30,0,0"
      TriangleIndices= "0 1 2 2 3 0
                         4 5 6 6 7 4
                         8 9 10 10 11 8
                         12 13 14 14 15 12
                         16 17 18 18 19 16
                         20 21 22 22 23 20"/>
  </GeometryModel3D.Geometry>
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="BlueViolet"/>
  </GeometryModel3D.Material>
</GeometryModel3D>
```

Eingeleitet wird die Beschreibung mit dem Tag `<GeometryModel3D>`. Es steht für ein geometrisches 3D-Modell. Dann folgt im Tag `<GeometryModel3D.Geometry>` die Beschreibung der Geometrie – also der Form. Die Positionen werden dabei über die Eigenschaft `Positions` im Tag `<MeshGeometry3D>` gesetzt. Die Reihenfolge wird bei der Eigenschaft `TriangleIndices` angegeben.

Bitte beachten Sie:

Der erste Punkt in der Auflistung hat immer den Index 0.



Das Material schließlich wird im Tag `<GeometryModel3D.Material>` beschrieben. Wir wählen hier ein einfaches Material in der Farbe Blauviolett.

Übernehmen Sie die Anweisungen aus dem vorigen Code und setzen Sie sie in das Tag `<Model3DGroup>`. Der Lohn der ganzen Arbeit sieht dann so aus:

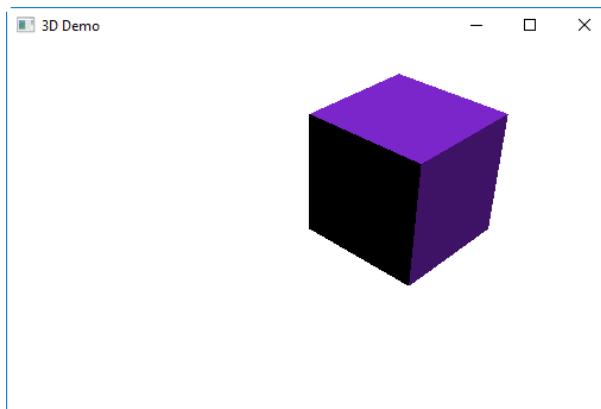


Abb. 3.4: Ein dreidimensionaler Würfel

Experimentieren Sie ruhig auch ein wenig mit den Werten. Verschieben Sie zum Beispiel einmal die Kamera und das Licht. Dann könnte der Würfel auch so aussehen:

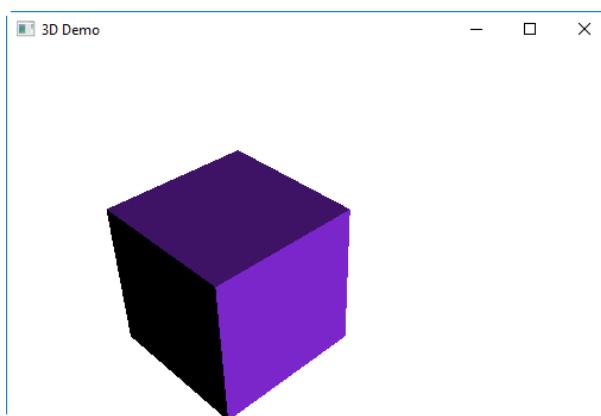


Abb. 3.5: Ein dreidimensionaler Würfel mit geänderten Kamera- und Lichteinstellungen



Das vollständige Beispiel finden Sie im Projekt **DreiDDemo** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Damit wollen wir unseren Ausflug in die 3D-Grafik auch schon wieder beenden.

Hinweis:

Es gibt externe Programme, mit denen Sie 3D-Grafiken komfortabel erstellen und als XAML-Code exportieren können. Damit ersparen Sie sich das mühselige Erstellen der Objekte per Hand.

3.3 Grafische Effekte

Abschließend wollen wir uns noch an einem kleinen Beispiel ansehen, welche grafischen Effekte Sie mit der WPF mit recht geringem Aufwand erzielen können. Der Code 3.5 projiziert ein Video auf einen Text. Dazu wird über die Klassen MediaPlayer und VideoDrawing ein Pinsel mit dem Video erzeugt, der dem Vordergrund des Textes zugewiesen wird.

```
//einen Medioplayer erzeugen
MediaPlayer meinPlayer = new MediaPlayer();
//und eine Instanz der Klasse VideoDrawing
VideoDrawing meinVideoDrawing = new VideoDrawing();
//das Video laden
//den Pfad müssen Sie gegebenenfalls anpassen
//bitte in einer Zeile eingeben
meinPlayer.Open(new Uri(@"c:/beispiele/demo.mp4",
UriKind.Absolute));
//dem Player des Videodrawings zuweisen
meinVideoDrawing.Player = meinPlayer;
//den Wiedergabebereich festlegen
meinVideoDrawing.Rect = new Rect(0, 0, Width, Height);
//einen neuen Drawing-Pinsel aus dem VideoDrawing erzeugen
DrawingBrush meinPinsel = new DrawingBrush(meinVideoDrawing);
//dem Vordergrund der Textbox zuweisen
textBox.Foreground = meinPinsel;
//und abspielen
meinPlayer.Play();
```

Code 3.5: Ein Video auf einem Text

Das Ergebnis sieht so aus:

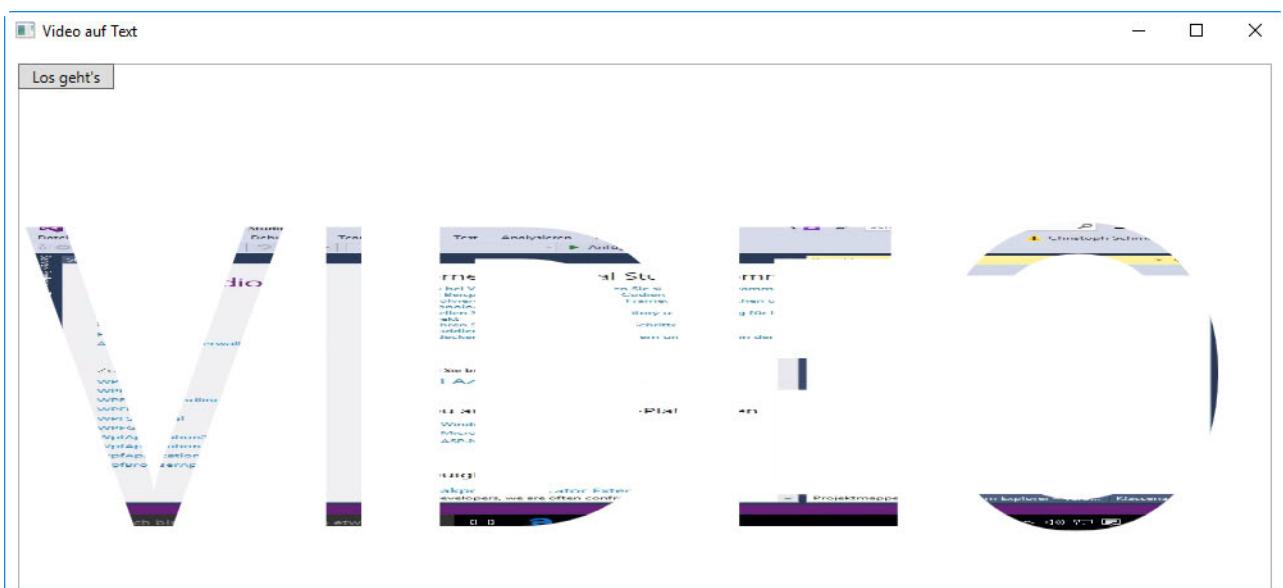


Abb. 3.6: Ein Video auf einem Text

Hinweis:

Den kompletten Code finden Sie im Projekt **VideoAufText** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Den Pfad zum Video müssen Sie gegebenenfalls anpassen.

Zusammenfassung

Animationen in der WPF können Sie über Abhängigkeitseigenschaften eines Objekts durchführen.

Bei einer Abhängigkeitseigenschaft hängt der Wert von anderen Quellen ab. Außerdem kann eine Abhängigkeitseigenschaft melden, dass sich ihr Wert verändert hat.

Sie müssen für jeden Typ die passende Animationsklasse verwenden.

Sie können auch die Dauer, die Wiederholung und den automatischen Rücklauf einer Animation steuern. Außerdem lassen sich Beschleunigung und Abbremsen festlegen.

Mit der WPF können Sie auch dreidimensionale Grafiken erstellen. Die Objekte werden dabei durch Dreiecke beschrieben.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie haben für eine Animation die Eigenschaften `From` und `To` nicht gesetzt. Was geschieht beim Starten der Animation? Begründen Sie bitte kurz Ihre Antwort.

- 3.2 Sie wollen eine Abhängigkeitseigenschaft vom Typ `double` animieren. Welche Klasse müssen Sie für die Animation verwenden?

- 3.3 Sie wollen die Höhe eines Shapes animieren. Wie heißt die entsprechende Eigenschaft?

- 3.4 Welchen Typ müssen Sie bei der Eigenschaft `Duration` einer Animation verwenden?

- 3.5 Wie groß darf der Wert für die Summe der beiden Eigenschaften `AccelerationRatio` und `DecelerationRatio` einer Animation maximal sein?

- 3.6 Welche Elemente müssen Sie in einen Viewport setzen?

- 3.7 Sie haben Objekte in eine dreidimensionale Szene eingefügt, die aber nicht zu sehen sind. Was könnte die Ursache sein?

4 Datenbindung

In diesem Kapitel beschäftigen wir uns mit der Datenbindung der WPF.

Sehen wir uns dazu erst einmal einige Grundlagen an.

4.1 Grundlagen

Bisher haben wir auf Änderungen an einem Steuerelement vor allem mit Ereignissen und entsprechenden Methoden reagiert. Im folgenden Code aktualisieren sich zum Beispiel ein Eingabefeld und ein Schieberegler wechselseitig.

```
private void TextBox1_TextChanged(object sender,
TextChangedEventArgs e)
{
    slider1.Value = Convert.ToDouble(textBox1.Text);
}

private void Slider1_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
{
    textBox1.Text = slider1.Value.ToString();
}
```

Code 4.1: Wechselseitige Aktualisierung von Steuerelementen

Hinweis:

Wir fangen in dem vorigen Code keine Fehler ab. Wenn zum Beispiel der Inhalt des Eingabefelds gelöscht wird oder wenn ein ungültiger Wert eingegeben wird, löst die Konvertierung eine Ausnahme aus.

Diese wechselseitige Aktualisierung lässt sich auch über die Datenbindung der WPF erreichen. Hier können Sie den Wert einer Eigenschaft vom Wert einer anderen Eigenschaft abhängig machen.



Bei der Datenbindung kann eine Eigenschaft den Wert einer anderen Eigenschaft bestimmen.

Um eine Eigenschaft eines Steuerelements an die Eigenschaft eines anderen Steuerelements zu binden, reicht eine kurze Anweisung im XAML-Code. Die folgende Anweisung hat zum Beispiel dieselbe Wirkung wie der Code oben:

```
<TextBox x:Name="textBox" HorizontalAlignment="Left" Height="23"
Margin="0,10,0,0" TextWrapping="Wrap" VerticalAlignment="Top"
Width="150" Text="{Binding ElementName=slider, Path=Value}">
```

Interessant ist vor allem der letzte Teil mit der Zuweisung auf die Eigenschaft `Text`. Hier wird über ein Binding-Objekt die Bindung hergestellt. Über die Eigenschaft `ElementName` wird das Element angesprochen und über die Eigenschaft `Path` die Eigenschaft. In unserem Beispiel erhält also die Eigenschaft `Text` von `textBox` den Wert der Eigenschaft `Value` von `slider`.

Der komplette Ausdruck für das Erstellen der Bindung muss in geschweifte Klammern gesetzt werden.



Die Eigenschaften `ElementName` und `Path` müssen außerdem durch ein Komma getrennt werden.

Die Bindung besteht in unserem Beispiel automatisch in beide Richtungen. Über die Eigenschaft `Mode` können Sie die Bindung aber auch gezielt steuern. Der Wert `OneWay` aktualisiert zum Beispiel nur das Ziel bei Änderungen der Quelle, aber nicht die Quelle bei Änderungen am Ziel. Der Wert `OneWayToSource` dagegen aktualisiert nur die Quelle bei Änderungen am Ziel, aber nicht das Ziel bei Änderungen an der Quelle.

Ob die Bindung in beide Richtungen besteht oder nur in eine, hängt auch von den Eigenschaften ab.



Und besonders interessant: Um die Konvertierung müssen Sie sich in unserem Beispiel nicht weiter kümmern. Obwohl der numerische Wert des Schiebereglers und der Text des Eingabefelds eigentlich nicht zueinander passen, erfolgt die Umsetzung in unserem Fall automatisch. Auch Fehler werden dabei abgefangen.

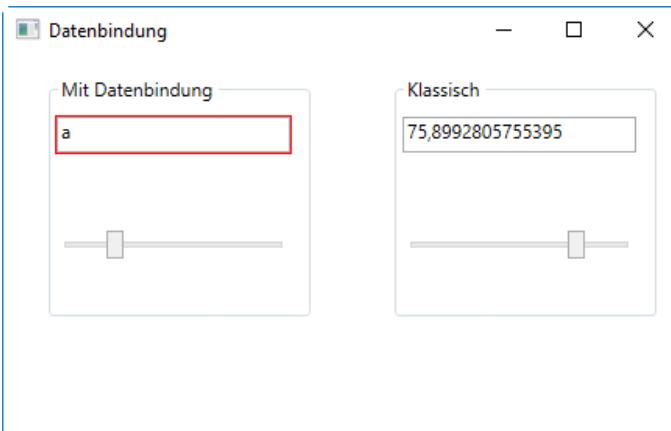


Abb. 4.1: Ein abgefangener Fehler bei der Datenbindung
(im Feld links steht ein ungültiger Wert)

Für die Quelle wird bei der Datenbindung häufig der Begriff **Source** verwendet. Das Ziel wird **Target** genannt.



Die Datenbindung lässt sich auch mit C#-Anweisungen herstellen. Allerdings ist dazu ein wenig mehr Aufwand erforderlich – wie der folgende Code 4.2 zeigt.

```
//eine neue Instanz der Klasse Binding erstellen
Binding meinBinding = new Binding();
//die Quelle setzen
meinBinding.Source = slider;
```

```
//den Pfad  
meinBinding.Path = new PropertyPath("Value");  
//und die Verbindung herstellen  
textBox.SetBinding(textBox.TextProperty, meinBinding);
```

Code 4.2: Eine Datenbindung mit C#-Anweisungen

Mit der Anweisung

```
Binding meinBinding = new Binding();
```

wird zunächst eine neue Instanz der Klasse `Binding` erstellt. Danach werden die Werte für die Eigenschaften `Source` und `Path` gesetzt. Die Eigenschaft `Path` erwartet dabei den Typ `PropertyPath`. Die Eigenschaft, zu der Sie die Bindung herstellen wollen, müssen Sie dabei als Zeichenkette übergeben.

Mit der Anweisung

```
textBox.SetBinding(textBox.TextProperty, meinBinding);
```

schließlich wird die Bindung hergestellt. Bitte achten Sie hier darauf, dass Sie bei der C#-Anweisung den Namen der Abhängigkeitseigenschaft benutzen müssen. Im XAML-Code dagegen können Sie auch den Namen der eigentlichen Eigenschaft verwenden.



Praktisch umgesetzt finden Sie die vorigen Codes im Projekt **DatenbindungEinfach** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Allerdings sind der Datenbindung auch Grenzen gesetzt. Sie können zwar eine beliebige Eigenschaft bei der Quelle benutzen, aber nur eine Abhängigkeitseigenschaft beim Ziel.

**Zur Auffrischung:**

Eine Abhängigkeitseigenschaft – auch *dependency property* genannt – erhält ihren Wert von anderen Quellen – hier eben von einer Eigenschaft eines anderen Objekts. Eine Abhängigkeitseigenschaft kann melden, dass sich ihr Wert verändert hat.

Außerdem funktioniert die Datenbindung bei vielen Steuerelementen nicht automatisch in beide Richtungen. Dann kann nur die Quelle direkt das Ziel verändern, aber nicht umgekehrt.

Auch der automatischen Konvertierung sind Grenzen gesetzt. So können Sie zum Beispiel nicht aus dem Wert von drei Schieberegbern automatisch eine Farbe nach dem RGB-Modell ableiten. In solchen Fällen müssen Sie selbst für die Konvertierung sorgen. Wie das geht, werden wir uns gleich noch ansehen.

Sie müssen bei der Datenbindung auch sorgfältig darauf achten, dass Sie das richtige Objekt und die richtige Eigenschaft verwenden. Wenn Sie sich zum Beispiel bei einem Bezeichner vertippen oder eine Eigenschaft falsch schreiben, wird das Programm trotzdem ohne Schwierigkeiten übersetzt und ausgeführt. Die Datenbindung kann aber natürlich nicht funktionieren.

Mögliche Probleme können Sie aber im Ausgabefenster des Debuggers überprüfen. Starten Sie dazu das Programm mit der Funktion **Debuggen/Debugging starten** oder mit der Funktionstaste **F5**. Lassen Sie dann mit der Funktion **Ansicht/Ausgabe** das Ausgabefenster anzeigen. Unten in der Liste werden dann auch Fehler bei der Datenbindung angezeigt – zum Beispiel eine nicht vorhandene Quelle.

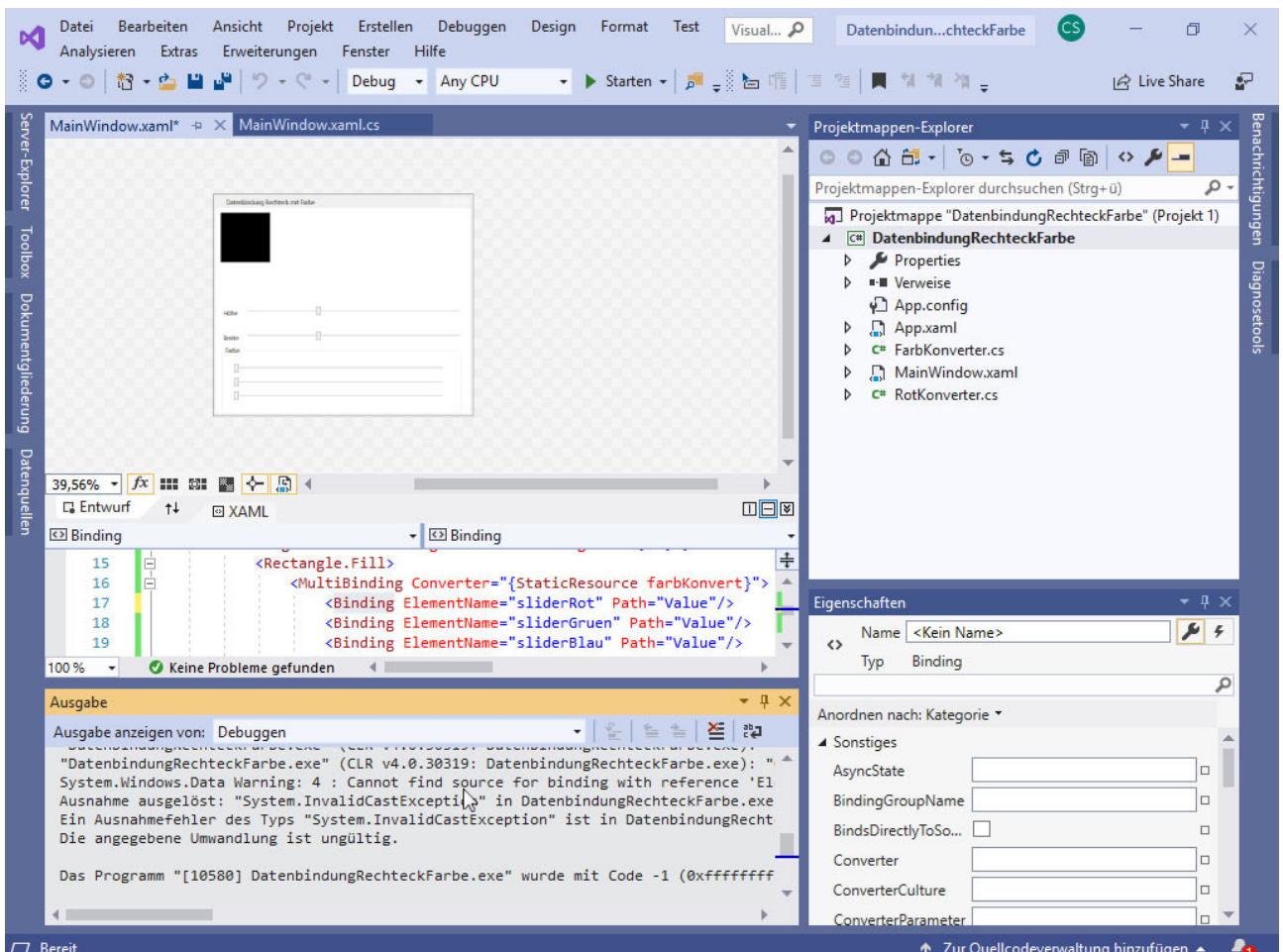


Abb. 4.2: Das Ausgabefenster des Debuggers mit einer Meldung zu einer fehlenden Quelle für die Datenbindung (unten links am Mauszeiger)

Hinweis:

Auch Windows-Forms-Anwendungen kennen das Konzept der Datenbindung. Sie können es allerdings nur für einige wenige Steuerelemente benutzen. Wir werden uns später beim Erstellen einer Datenbankanwendung noch einmal mit diesem Thema beschäftigen.

4.2 Einige einfache Beispiele

Schauen wir uns den Einsatz der Datenbindung jetzt an einigen praktischen Beispielen an. Beginnen wir mit einer Veränderung unseres dreidimensionalen Würfels. Er soll durch einen Schieberegler um eine Achse gedreht werden können. Das hört sich aufwendig an, ist aber schnell umgesetzt. Denn die WPF kennt einige Klassen für 3D-Transformationen.



Eine Transformation ist – ganz allgemein ausgedrückt – eine Umformung oder Umwandlung. Zu den Transformationen bei einer 3D-Grafik gehören Skalierungen zum Verändern der Größe, Rotationen für Drehungen und Verschiebungen zum Verändern der Position.

Über die Eigenschaft `Transform` des Modells können Sie dann die gewünschte Transformation zuweisen – zum Beispiel über die Klasse `RotateTransform3D`. Sie führt eine Drehung durch. Wie die Drehung erfolgt, wird über die Eigenschaft `Rotation` festgelegt. Sie können hier zum Beispiel über eine Instanz der Klasse `AxisAngleRotation3D` eine Rotation mit einem bestimmten Winkel um eine bestimmte Achse festlegen. Und genau diesen Winkel für die Drehung wollen wir jetzt über eine Datenbindung an den aktuellen Wert eines Schiebereglers verändern.

Laden Sie das Projekt mit dem dreidimensionalen Würfel noch einmal. Ergänzen Sie im ersten Schritt bitte im XAML-Code hinter dem schließenden Tag für den Inhalt des Modells die folgenden fett gedruckten Anweisungen.

```
...
    <DiffuseMaterial Brush="BlueViolet"/>
    </GeometryModel3D.Material>
  </GeometryModel3D>
</Model3DGroup>
</ModelVisual3D.Content>
<ModelVisual3D.Transform>
  <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <!-- bitte in einer Zeile eingeben -->
      <AxisAngleRotation3D Angle="{Binding
        ElementName=meinSlider, Path=Value}" Axis = "1 1 1"/>
    </RotateTransform3D.Rotation>
  </RotateTransform3D>
</ModelVisual3D.Transform>
</ModelVisual3D>
```

Code 4.3: Die Transformation für den Würfel

Hier legen wir fest, dass die Eigenschaft `Transform` für unser Modell eine Rotation um eine Achse erhalten soll. Den Winkel legen wir über die Eigenschaft `Angle` fest. Sie erhält ihren Wert über eine Datenbindung vom Steuerelement mit dem Namen `meinSlider`. Die Angabe `Axis = "1 1 1"` legt fest, dass die Rotation um alle drei Achsen erfolgen soll. Wenn Sie das nicht möchten, setzen Sie den Wert der Achse, um die nicht rotiert werden soll, auf 0. Die Achsen werden dabei wieder in der Reihenfolge X, Y, Z angegeben.

Jetzt fehlt noch der Schieberegler zum Verändern der Werte. Er ist mit der folgenden Anweisung schnell erstellt:

```
<Slider x:Name="meinSlider" Height="23" Margin="10,271,10,10"
  Maximum="360" />
```

Setzen Sie diese Anweisung bitte unten in das Grid außerhalb des Viewports. Testen Sie das Programm dann. Der Würfel sollte sich bei einer Veränderung des Schiebereglers drehen.

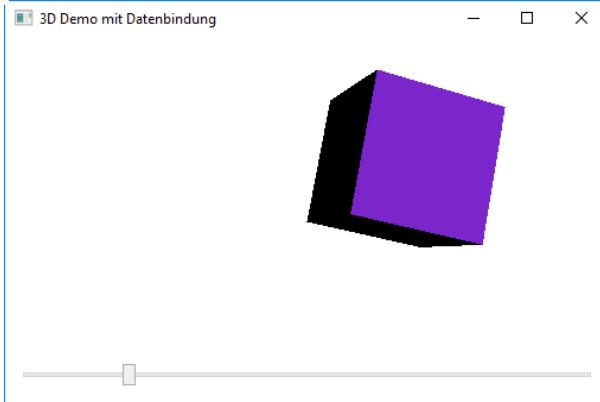


Abb. 4.3: Der gedrehte Würfel

Das komplette Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **DreiDDemoDatenbindung**.



Schauen wir uns jetzt noch ein anderes einfaches Beispiel an. In einem Programm sollen die Breite und die Höhe eines Rechtecks über Schieberegler verändert werden können. Die entsprechenden XAML-Anweisungen sehen so aus:

```
<!-- bitte jeweils in einer Zeile eingeben -->
<Rectangle Fill="#FFF4F4F5" HorizontalAlignment="Left"
Margin="10,10,0,0" Stroke="Black" VerticalAlignment="Top"
Width="{Binding ElementName=sliderBreite, Path=Value}"
Height="{Binding ElementName=sliderHoehe, Path=Value}"/>
<Label x:Name="label1" Content="Höhe"
HorizontalAlignment="Left" Margin="10,200,0,0"
VerticalAlignment="Top"/>
<Slider x:Name="sliderHoehe" HorizontalAlignment="Left"
Margin="60,200,0,0" VerticalAlignment="Top" Maximum="300"
Width="440" Value="100"/>
<Label x:Name="label" Content="Breite"
HorizontalAlignment="Left" Margin="10,250,0,0"
VerticalAlignment="Top"/>
<Slider x:Name="sliderBreite" HorizontalAlignment="Left"
Margin="60,250,0,0" VerticalAlignment="Top" Maximum="300"
Width="440" Value="100"/>
```

Code 4.4: Ein Rechteck mit Datenbindung

Die Eigenschaft `Width` des Rechtecks verbinden wir mit dem aktuellen Wert des Schiebereglers `sliderBreite` und die Eigenschaft `Height` verbinden wir mit dem aktuellen Wert des Schiebereglers `sliderHoehe`. Damit lassen sich die Breite und die Höhe des Rechtecks über die Schieberegler verändern.

Den kompletten Code finden Sie im Projekt **DatenbindungRechteck** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.



4.3 Ein komplexeres Beispiel

Mit ähnlichen Techniken lassen sich jetzt auch andere Eigenschaften des Rechtecks anpassen – zum Beispiel die Rahmenbreite. Andere Eigenschaften, wie zum Beispiel die Füllfarbe, lassen sich allerdings nicht so einfach über die Datenbindung ändern. Denn die Eigenschaft `Fill` eines Shapes setzt die Farbe über den Typ `Brush`. Und er lässt sich nicht ohne Weiteres aus dem numerischen Wert eines Schiebereglers erstellen. Sie müssen daher selbst für die Konvertierung sorgen.

Hinweis:

Sie können einen Pinsel vom Typ `Brush` aus einer Zeichenkette erstellen. Damit ließe sich die Füllfarbe zum Beispiel über eine Datenbindung an ein Eingabefeld setzen. Das ist allerdings alles andere als komfortabel, da der Anwender dann zum Beispiel den Namen der Farbe oder den hexadezimalen Farbcode eingeben müsste.

Wenn Sie nur einen Wert konvertieren müssen, erstellen Sie dazu eine Klasse, die die Schnittstelle `System.Windows.Data.IValueConverter`⁹ implementiert. In dieser Klasse erstellen Sie die Methoden `Convert()` und `ConvertBack()`. Die beiden Methoden müssen öffentlich sein und den Typ `object` zurückgeben. Sie erwarten folgende Argumente:

- `value` vom Typ `object` für den Wert, der umgewandelt werden soll,
- `targetType` vom Typ `Type` für den Typ des Ziels,
- `parameter` vom Typ `object` für einen Parameter für die Konvertierung und
- `culture` vom Typ `CultureInfo` für die Kultur.



Die Kultur beschreibt im .NET Framework bestimmte Eigenschaften wie zum Beispiel den Kalender, die Darstellung von Datums- und Zeitangaben und auch die Formatierung von Zahlen.

Die Methode `Convert()` ist für das Umwandeln des Werts aus der Quelle in das passende Format für das Ziel zuständig. Die Methode `ConvertBack()` dagegen baut die Daten in die andere Richtung um. Wenn Sie nur eine Umwandlung von der Quelle zum Ziel benötigen, können Sie die Methode auch leer lassen.

Schauen wir uns den Einsatz der Schnittstelle `IValueConverter` jetzt an einem praktischen Beispiel an. Wir benutzen den Wert eines Schiebereglers, um den Rotanteil einer Farbe zu verändern.

Falls Sie das Programm zur Veränderung der Breite und Höhe des Rechtecks nicht geöffnet haben, laden Sie es bitte noch einmal. Ergänzen Sie dann unten drei weitere Schieberegler für die Farben. Sie können sie zum Beispiel in eine Gruppe mit der Überschrift **Farben** setzen. Legen Sie das Maximum für die drei Schieberegler jeweils auf 255 fest. Das ist der maximale Wert, den ein Farbanteil im RGB-Modell haben kann.

9. *Value converter* lässt sich mit „Wertumwandler“ übersetzen.

Erstellen Sie anschließend mit der Funktion **Projekt/Neues Element hinzufügen ...** eine neue Klasse für das Projekt. Wählen Sie dazu im Fenster **Neues Element hinzufügen** den Eintrag **Klasse** für den Zweig **Visual C#** aus. Als Namen können Sie zum Beispiel **RotKonverter** verwenden.

Im Quelltext binden Sie zuerst über entsprechende `using`-Anweisungen die Namensräume `System.Windows.Data` und `System.Windows.Media` ein. Den Namensraum `System.Windows.Media` brauchen wir für die Umwandlung der Farben.

Sorgen Sie danach dafür, dass die Klasse die Schnittstelle `IValueConverter` implementiert. Ergänzen Sie dazu hinter dem Namen der Klasse einen Doppelpunkt und die Angabe `IValueConverter`.

Die beiden Methoden können Sie nun wie beim Implementieren der Schnittstelle `IComparable` in der Klasse `Score` für unsere Bestenliste per Hand erstellen. Gerade bei umfangreicherer Schnittstellen mit komplizierten Methoden ist das allerdings recht fehleranfällig. Sie sollten die Arbeit daher Visual Studio überlassen.

Klicken Sie dazu mit der Maus in den Namen `IValueConverter` im Quelltext. Danach können Sie über die Livecodeanalyse die Methode erstellen lassen. Probieren Sie das einmal aus. Öffnen Sie die Vorschläge der Livecodeanalyse für die Zeile mit der Vereinbarung der Klasse. Wählen Sie dann den Vorschlag **Schnittstelle implementieren**.

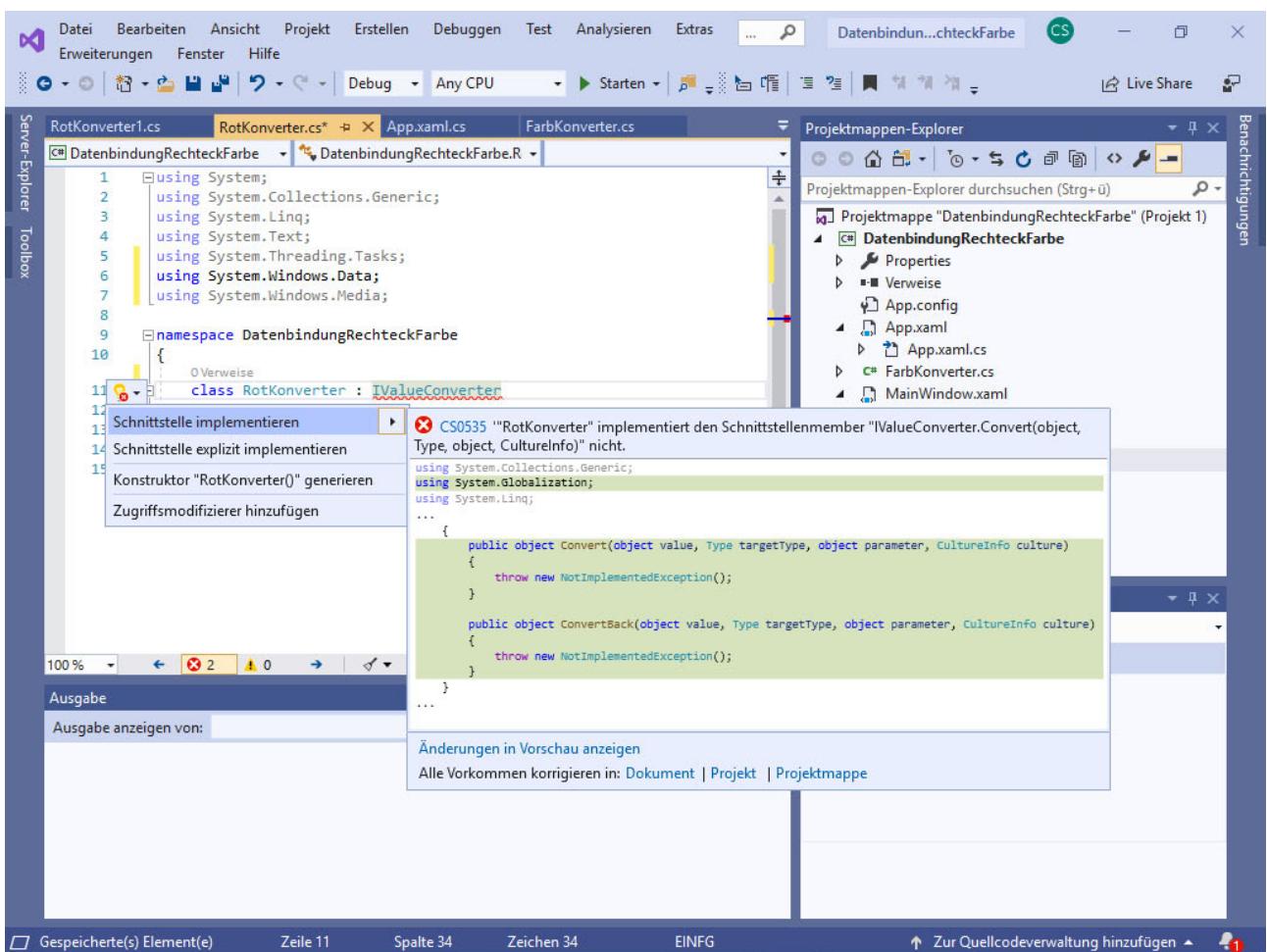


Abb. 4.4: Die Vorschläge der Livecodeanalyse für das Implementieren der Schnittstelle

Visual Studio fügt die Methoden automatisch ein und ergänzt in jeder Methode die Anweisung

```
throw new NotImplementedException();
```

Sie sorgt dafür, dass eine Ausnahme vom Typ `NotImplementedException` ausgelöst wird.

In der Methode `Convert()` lassen wir jetzt den Wert vom Schieberegler für die rote Farbe in den Typ `byte` umwandeln und erstellen dann mit der Methode `FromRgb()` der Struktur `Color` eine Farbe. Die Anteile der beiden anderen Farben setzen wir auf 0. Mit der Farbe erzeugen wir anschließend einen Pinsel, den wir aus der Methode zurückgeben. Die vollständige Methode sieht dann so aus:

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
{
    //den Wert in den Typ byte umwandeln
    byte rot = (byte)(double)value;
    //eine neue Farbe erstellen
    //Grün und Blau sind 0
    Color neueFarbe = Color.FromRgb(rot, 0, 0);
    //einen neuen Pinsel in der Farbe erstellen
    SolidColorBrush pinsel = new SolidColorBrush(neueFarbe);
    //und den Pinsel zurückgeben
    return pinsel;
}
```

Code 4.5: Die Methode `Convert()`

Hinweis:

Denken Sie bitte an das Einbinden des Namensraums `System.Windows.Media`. Sonst ist die `Color`-Struktur nicht bekannt.

Sie müssen den übergebenen Wert erst in einen `double` umbauen und diesen `double` dann in einen `byte`. Ein direktes Umwandeln in den Typ `byte` löst eine Ausnahme aus.

Die Methode `ConvertBack()` lassen wir unverändert. Eine Umwandlung aus der Farbe wollen wir ja nicht vornehmen. Löschen können wir die Methode allerdings nicht. Denn die Schnittstelle schreibt diese Methode vor. Sie muss also in jedem Fall vorhanden sein – auch wenn sie leer ist oder nur die Anweisung zum Erzeugen der Ausnahme enthält.



Auch wenn Sie eine Methode einer Schnittstelle nicht benutzen, müssen Sie sie erstellen. Andernfalls können Sie den Quelltext nicht übersetzen.

Bei einer Klasse, die die Schnittstelle `IValueConverter` implementiert, hat es sich als guter Stil etabliert, den Quell- und Zieltyp über ein Attribut mit dem Namen `ValueConversion` zu beschreiben. Dadurch können zum Beispiel externe Werkzeuge die Typen abfragen.

Die Attributbeschreibung wird in eckigen Klammern vor der Vereinbarung der Klasse angegeben. Für unser Beispiel könnte sie so aussehen:

```
[ValueConversion(typeof(double), typeof(SolidColorBrush))]
```

Hier wird erst der Typ der Quelle angegeben und dann der Typ des Ziels. Die beiden Typen können Sie dabei aber nicht direkt angeben. Sie müssen sie über den Operator `typeof()` ermitteln.

Ob Sie das Attribut beschreiben oder nicht, ist vor allem eine Stilfrage. Für den internen Einsatz der Klasse spielt die Angabe keine Rolle.

Bitte beachten Sie:



Attribute, die Sie über solche Attributbeschreibungen erstellen, dienen vor allem zur Information. Sie haben nichts mit den Attributen einer Klasse zu tun. Sie werden bei C# über Felder abgebildet.

Im nächsten Schritt müssen Sie noch dafür sorgen, dass auf die Klasse für die Konvertierung im XAML-Code zugegriffen werden kann und dass die Methode für die Konvertierung aufgerufen wird.

Dazu können Sie zum Beispiel bei den Ressourcen des Fensters eine Instanz der Klasse erstellen. Das erfolgt mit den XAML-Anweisungen

```
<Window.Resources>
    <local:RotKonverter x:Key="rotKonvert"/>
</Window.Resources>
```

Code 4.6: Das Erzeugen einer Instanz im XAML-Code

Das Erzeugen der Instanz übernimmt dabei die Anweisung

```
<local:RotKonverter x:Key="rotKonvert"/>
```

Hinter `local:` geben Sie den Namen der Klasse an, dann folgt hinter `x:Key` der Name der Instanz.

Beim Rechteck geben Sie dann für die Eigenschaften `Fill` an, dass eine Bindung zum Schieberegler für die rote Farbe über die Eigenschaft `Value` besteht und die Konvertierung über die Instanz `rotKonvert` der Klasse `RotKonverter` erfolgen soll. Die entsprechende Anweisung sieht so aus (die neuen Teile sind fett markiert):

```
<Rectangle Fill="{Binding ElementName=sliderRot, Path=Value,
    Converter={StaticResource rotKonvert}}"
    HorizontalAlignment="Left" Margin="10,10,0,0" Stroke="Black"
    VerticalAlignment="Top" Width="{Binding
        ElementName=sliderBreite, Path=Value}" Height="{Binding
        ElementName=sliderHoehe, Path=Value}"/>
```

Neu ist hier nur die Eigenschaft `Converter`. Sie legt fest, dass die Konvertierung über unsere Instanz `rotKonvert` erfolgen soll, die wir als Ressource vereinbart haben.

Übernehmen Sie diese Änderungen jetzt bitte und testen Sie das Programm. Sie sollten nur über einen Schieberegler den Rotanteil in der Füllfarbe verändern können.

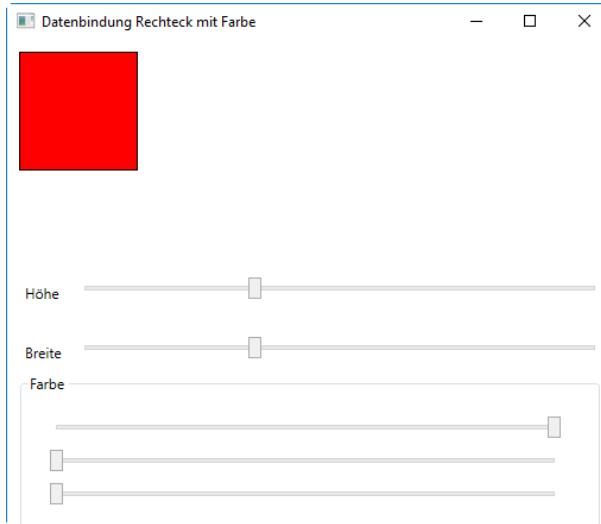


Abb. 4.5: Der Schieberegler im Einsatz

Um nun auch die anderen beiden Schieberegler für die Änderung der Füllfarbe benutzen zu können, könnten wir die Werte der Farbanteile über Felder in der Klasse für die Konvertierung zwischenspeichern und die zwischengespeicherten Werte beim Erzeugen der Farbe benutzen. Dazu müssten wir aber recht umständlich aus der Klasse zum Konvertieren auf die Schieberegler zugreifen, um uns die Werte zu beschaffen.

Außerdem gibt es eine sehr viel einfachere Alternative: Wir binden den Wert der Füllfarbe nicht mehr nur an eine Quelle, sondern an mehrere – eben an unsere drei Schieberegler.

Dazu vereinbaren Sie für die Eigenschaft `MultiBinding` der Füllfarbe eine Liste mit den einzelnen Bindungen. Das könnte zum Beispiel so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<Rectangle HorizontalAlignment="Left" Margin="10,10,0,0"
Stroke="Black" VerticalAlignment="Top" Width="{Binding
ElementName=sliderBreite, Path=Value}" Height="{Binding
ElementName=sliderHoehe, Path=Value}">
    <Rectangle.Fill>
        <MultiBinding>
            <Binding ElementName="sliderRot" Path="Value"/>
            <Binding ElementName="sliderGruen" Path="Value"/>
            <Binding ElementName="sliderBlau" Path="Value"/>
        </MultiBinding>
    </Rectangle.Fill>
</Rectangle>
```

Code 4.7: Das MultiBinding für das Rechteck

Wenn Sie diese Anweisungen so übernehmen, beschwert sich Visual Studio allerdings, dass ein **MultiValueConverter** angegeben werden muss. Denn für eine mehrfache Bindung müssen Sie die Konvertierung mit einer Klasse vornehmen, die die Schnittstelle `System.Windows.Data.IMultiValueConverter` implementiert. Das erfolgt fast genauso wie das Implementieren der Schnittstelle

`System.Windows.Data.IValueConverter`. Die Werte werden lediglich als Array übergeben und nicht mehr als einzelner Wert.

Erstellen wir diese Klasse. Legen Sie mit der Funktion **Projekt/Neues Element hinzufügen ...** eine neue Klasse für das Projekt an. Als Namen können Sie zum Beispiel **FarbKonverter** verwenden. Binden Sie über entsprechende `using`-Anweisungen die Namensräume `System.Windows.Data` und `System.Windows.Media` ein. Sorgen Sie danach dafür, dass die Klasse die Schnittstelle `IMultiValueConverter` implementiert. Lassen Sie Visual Studio dann über die Livecodeanalyse die beiden Methoden erstellen.

Für die Methode `Convert()` übernehmen Sie bitte den Quelltext aus dem folgenden Code 4.8.

```
public object Convert(object[] values, Type targetType, object parameter, CultureInfo culture)
{
    //die Werte in den Typ byte umwandeln
    byte rot = (byte)(double)values[0];
    byte gruen = (byte)(double)values[1];
    byte blau = (byte)(double)values[2];
    //ein neue Farbe erstellen
    Color neueFarbe = Color.FromRgb(rot, gruen, blau);
    //einen neuen Pinsel in der Farbe erstellen
    SolidColorBrush pinsel = new SolidColorBrush(neueFarbe);
    //und den Pinsel zurückgeben
    return pinsel;
}
```

Code 4.8: Die Methode Convert() für die Klasse FarbKonverter

Neu sind hier vor allem die ersten drei Anweisungen. Hier wandeln wir die Werte aus dem Array `values` um. Die Werte in dem Array entsprechen dabei der Reihenfolge der Bindungen. Der Wert `values[0]` steht also für den Wert der ersten Bindung, der Wert `values[1]` für den Wert der zweiten Bindung und so weiter.

Bitte beachten Sie:

Der Parameter heißt `values` und nicht `value` wie bei der Schnittstelle `IValueConverter`.



Aus den drei Werten erstellen wir unsere Farbe und den Pinsel. Diesen Pinsel liefern wir zurück. Das kennen Sie ja bereits.

Die Methode `ConvertBack()` können Sie wieder unverändert lassen.

Erstellen Sie dann bei den Ressourcen für das Fenster eine Instanz der Klasse für die Farbkonvertierung. Die Anweisungen könnten so aussehen:

```
<Window.Resources>
    <local:FarbKonverter x:Key="farbKonvert" />
</Window.Resources>
```

Code 4.9: Das Erstellen einer Instanz für den Konverter

Im letzten Schritt müssen Sie die Instanz wieder bei der Eigenschaft `Converter` für das Multibinding angeben. Das erfolgt wie auch bei der einfachen Bindung mit Konvertierung. Die Anweisung könnte also so aussehen:

```
...
<Rectangle.Fill>
    <MultiBinding Converter="{StaticResource farbKonvert}">
    ...

```

Code 4.10: Das Setzen der Eigenschaften Converter

Hinweis:

Unter Umständen erscheint beim Zugriff auf die Klassen im XAML-Code eine Fehlermeldung, dass die Klassen nicht bekannt sind. Überprüfen Sie dann noch einmal sorgfältig, ob Sie sich nicht vertippt haben. Wenn der Name stimmt, können Sie das Projekt in der Regel trotz der Meldung ausführen lassen.



Das vollständige Projekt mit den beiden Klassen zur Umwandlung finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **DatenbindungRechteckFarbe**. Die Farbe wird in dem Projekt durch die Bindung aller drei Schieberegler gesteuert.

Abschließend noch ein Hinweis:

Sie können die Methoden zum Konvertieren auch für andere Zwecke verwenden. In der Methode `Convert()` der Schnittstelle `IMultiValueConverter` könnten Sie zum Beispiel Werte aus mehreren Quellen mit arithmetischen Operationen verarbeiten und dann das Ergebnis zurückgeben.

Zusammenfassung

Bei der Datenbindung kann eine Eigenschaft den Wert einer anderen Eigenschaft bestimmen. Die Datenbindung können Sie im XAML-Code oder in den Code-Behind-Daten festlegen.

Die Bindung besteht zwischen einer Quelle – Source genannt – und einem Ziel – Target genannt.

Die Datenbindung kann in unterschiedliche Richtungen erfolgen.

Fehler bei der Datenbindung werden im Ausgabefenster des Debuggers angezeigt.

Wenn keine automatische Konvertierung möglich ist, müssen Sie entsprechende Schnittstellen implementieren.

Beim Implementieren einer Schnittstelle müssen Sie immer alle Methoden erstellen – auch dann, wenn Sie eine Methode gar nicht benötigen.

Instanzen einer Klasse aus Code-Behind-Dateien können Sie auch im XAML-Code erstellen.

Über die Eigenschaft `Multibinding` können Sie eine Eigenschaft an mehrere Quellen binden.

Aufgaben zur Selbstüberprüfung

- 4.1 Sie wollen die Eigenschaft `Text` von der Eigenschaft `Value` eines Steuerelements mit dem Namen `quelle` abhängig machen. Formulieren Sie eine entsprechende XAML-Anweisung. Sie müssen nur die Zuweisung der Eigenschaft notieren.

- 4.2 Über welche Eigenschaft steuern Sie die Richtung der Datenbindung?

- 4.3 Können Sie eine beliebige Eigenschaft beim Ziel der Datenbindung benutzen?

- 4.4 Sie haben eine einfache Datenbindung erstellt, bei der die Werte nicht automatisch konvertiert werden. Welche Schnittstelle müssen Sie für die Konvertierung implementieren?

- 4.5 Sie wollen im XAML-Code bei den Ressourcen des Fensters eine Instanz `test` einer Klasse `Testklasse` erstellen. Wie müssen die Anweisungen lauten?

- 4.6 Welche Schnittstelle müssen Sie implementieren, wenn Sie Werte bei einer mehrfachen Datenbindung konvertieren müssen?

Schlussbetrachtung

In diesem Studienheft haben Sie einige weitere interessante Bereiche der WPF kennengelernt. Sie wissen jetzt, wie Sie mit Medien wie Audios und Videos umgehen und wie Sie Zeichenoperationen durchführen. Außerdem haben Sie einen kleinen Ausflug in die Welt der Animationen und der 3D-Grafik unternommen. Zum Abschluss haben wir uns noch die Datenbindung angesehen. Hier können Sie die Eigenschaften eines Steuerelements von Eigenschaften anderer Steuerelemente abhängig machen – ohne eine einzige Zeile C#-Code zu schreiben.

Experimentieren Sie auch mit den Beispielen aus diesem Studienheft. Einige Erweiterungen zu den Programmen warten wie gewohnt bei den Einsendaufgaben auf Sie.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Sie können die Klasse `MediaElement` nur dann einsetzen, wenn der Windows Media Player ab der Version 10 installiert ist. Außerdem können Sie mit der Klasse keine Dateien wiedergeben, die Sie als Ressource in ein Projekt einbetten.
- 1.2 Die Lautstärke wird im Bereich 0,0 bis 1,0 festgelegt.
- 1.3 Die Eigenschaft `Source` erwartet den Typ `Uri`.
- 1.4 Wenn Sie aus dem Code die Steuerung der Medienwiedergabe übernehmen wollen, müssen Sie die Eigenschaft `LoadedBehaviour` für das `MediaElement` auf `Manual` setzen.
- 1.5 Die Anweisung kann so aussehen:

```
image.Source = new BitmapImage(new Uri("bilder/bild.png",
UriKind.Relative));
```

Der Bezeichner für das Steuerelement kann auch abweichen.

- 1.6 In welche Richtung das Mausrad gedreht wurde, können Sie über die Eigenschaft `e.Delta` in der Methode für das Mausereignis abfragen. Wenn sie einen positiven Wert liefert, wurde das Rad nach vorn gedreht, andernfalls nach hinten.

Kapitel 2

- 2.1 Beim Loslassen der linken Maustaste wird das Ereignis `MouseLeftButtonUp` ausgelöst.
- 2.2 Ob die linke Maustaste gedrückt wurde, können Sie über die Eigenschaft `e.LeftButton` herausfinden. Die Anweisung könnte so aussehen:

```
if (e.LeftButton == MouseButtonState.Pressed)
```
- 2.3 Die Position können Sie mit der Methode `GetPosition()` ermitteln. Sie liefert Ihnen den Typ `Point`. Als Argument übergeben Sie das Steuerelement, das als relativer Ausgangspunkt für das Ermitteln der Position benutzt wird.
- 2.4 Sie haben wahrscheinlich vergessen, das Shape zu einem übergeordneten Element hinzufügen.
- 2.5 Die Position eines Shapes setzen Sie mit den Methoden `SetLeft()` und `SetTop()` der Klasse `Canvas`. Die Methoden erwarten das Objekt, dessen Position Sie ändern wollen, und die Position als Argumente.
- 2.6 Sie können die Farbe nicht direkt zuweisen. Sie müssen sie zum Beispiel mit der Methode `FromArgb()` der Klasse `Color` in das passende Format umwandeln.

- 2.7 Um alle untergeordneten Elemente in einem Container vom Typ `Canvas` zu löschen, benutzen Sie die Methode `Children.Clear()`.

- 2.8 Die Anweisung könnte so aussehen:

```
System.Windows.Markup.XamlWriter.Save(meinCanvas,  
meinStream);
```

Der Bezeichner für das zweite Argument kann auch abweichen.

Kapitel 3

- 3.1 Beim Starten der Animation geschieht nichts Sichtbares. Es werden die aktuellen Werte benutzt.
- 3.2 Für die Animation einer Abhängigkeitseigenschaft vom Typ `double` verwenden Sie die Klasse `DoubleAnimation`.
- 3.3 Die Eigenschaft für die Animation der Höhe eines Shapes heißt `HeightProperty`.
- 3.4 Für die Eigenschaft `Duration` verwenden Sie den Typ `TimeSpan`.
- 3.5 Die Summe der beiden Werte darf maximal 1 betragen.
- 3.6 In einen Viewport setzen Sie eine Kamera, eine Lichtquelle und die Objekte.
- 3.7 Wahrscheinlich haben Sie den Objekten kein Material zugewiesen.

Kapitel 4

- 4.1 Die Anweisung könnte so aussehen:

```
Text="{Binding ElementName=quelle, Path=Value}"
```

- 4.2 Die Richtung der Datenbindung steuern Sie über die Eigenschaft `Mode`.

- 4.3 Nein. Als Ziel können nur Abhängigkeitseigenschaften genutzt werden.

- 4.4 Sie müssen für die Konvertierung die Schnittstelle `System.Windows.Data.IValueConverter` implementieren.

- 4.5 Die Anweisungen könnten so aussehen:

```
<Window.Resources>  
  <local:Testklasse x:Key="test"/>  
</Window.Resources>
```

- 4.6 Sie müssen die Schnittstelle `System.Windows.Data.IMultiValueConverter` implementieren.

B. Glossar

Attribut	Attribute werden in Attributbeschreibungen eingesetzt. Sie dienen vor allem zur Beschreibung und liefern zum Beispiel Informationen nach außen. Der Begriff Attribut findet sich auch bei Klassen. Dort beschreibt es die Eigenschaften und den Zustand eines Objekts. Diese Attribute werden in C# über Felder abgebildet.
Ausnahmebehandlung	Durch die Ausnahmebehandlung können Sie in einem Programm auf Laufzeitfehler reagieren. Die Ausnahmebehandlung wird auch <i>Exception Handling</i> genannt.
Canvas	Das Steuerelement Canvas ist ein Layout-Container, in dem untergeordnete Elemente über absolute Koordinaten positioniert werden können.
Casting	Siehe <i>Typecasting</i>
Datenbindung	Bei der Datenbindung bestimmt der Wert einer Eigenschaft den Wert einer anderen Eigenschaft.
Eventhandler	Der <i>Eventhandler</i> stellt eine Verbindung zwischen einem Ereignis und einer Methode her, die beim Eintreten des Ereignisses ausgeführt werden soll.
Extensible Application Markup Language	Siehe XAML
GDI+	GDI+ ist eine geräteunabhängige Programmierschnittstelle für die Ausgabe von Informationen auf grafikfähigen Geräten.
Kultur	Die Kultur beschreibt im .NET Framework bestimmte Eigenschaften, wie zum Beispiel den Kalender, die Darstellung von Datums- und Zeitangaben und auch die Formatierung von Zahlen.
Material	Ein Material beschreibt die Oberfläche eines Objekts. Ohne Material ist ein Objekt nicht sichtbar, da es kein Licht reflektiert.
Media Player	Der Media Player ist ein Programm von Windows zur Wiedergabe von Audio- und Videodateien.
Quickinfos	Quickinfos sind kurze Hinwestexte. Sie werden normalerweise angezeigt, wenn der Mauszeiger einen kurzen Moment auf einem bestimmten Element stehen bleibt.
Registrierung (Eventhandler)	Bei der Registrierung eines Eventhandlers wird ein Ereignis mit einer Methode verbunden.

RGB-Modell	Beim RGB-Modell wird eine Farbe über die drei Parameter Rot, Grün und Blau festgelegt. Die Werte geben dabei den Anteil der jeweiligen Farbe an und müssen im Bereich zwischen 0 und 255 liegen.
Schnittstelle	Eine Schnittstelle ist – etwas vereinfacht ausgedrückt – eine Klasse, die die Signatur von bestimmten Methoden enthält. Die Methoden selbst – also die eigentliche Funktionalität – müssen in eigenen Klassen im Detail implementiert werden – und zwar sämtliche Methoden, die die Schnittstelle vorgibt.
Signatur	Eine Signatur beschreibt den Namen und die Typen der Parameterliste einer Methode.
Source	Source ist die englische Bezeichnung für Quelle.
Target	Target ist die englische Bezeichnung für Ziel.
Timer	Ein <i>Timer</i> löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis aus, in dem Sie Anweisungen verarbeiten lassen können. Ein <i>Timer</i> ermöglicht damit die zeitgesteuerte Verarbeitung von Anweisungen.
Tooltipp	Siehe Quickinfos
Transformation	Eine Transformation ist – ganz allgemein ausgedrückt – eine Umformung oder Umwandlung.
Typecasting	<i>Typecasting</i> bezeichnet das Umwandeln eines Werts in einen anderen Datentyp.
URI	URI steht für <i>Uniform Resource Identifier</i> . Es handelt sich um eine eindeutige Identifizierung einer Ressource – zum Beispiel einer Internet-Seite.
Viewport	Ein Viewport dient als eine Art Fenster für eine dreidimensionale Ansicht und setzt die Objekte in der richtigen Form um. Die Mitte des Viewports bildet auch den Mittelpunkt des Koordinatensystems. Ein Viewport enthält in der Regel die Kamera, eine Lichtquelle und die Objekte selbst.
XAML	XAML steht für <i>eXtensible Application Markup Language</i> . Es handelt sich um die Beschreibungssprache für die Oberfläche von WPF-Anwendungen. XAML basiert auf XML und stellt die Informationen ebenfalls strukturiert in Textform dar.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 6. Aufl., Bonn: Rheinwerk.

Speziell mit dem Thema WPF beschäftigt sich das folgende Buch:

Huber, T.C. (2019). *Windows Presentation Foundation*. Das umfassende Handbuch. 5. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Die Vorschau eines Videos im Steuerelement	4
Abb. 1.2	Die Wiedergabe eines Videos	5
Abb. 1.3	Der Mediaplayer	6
Abb. 1.4	Der Sammlungs-Editor	7
Abb. 1.5	Die Symbolleiste mit den drei Symbolen	8
Abb. 1.6	Das Fenster in der Rohform	9
Abb. 2.1	Das Malprogramm	21
Abb. 2.2	Das Menüband für das Malprogramm	24
Abb. 2.3	Die Auswahl der Farbe	25
Abb. 3.1	Ein dreidimensionales Koordinatensystem	40
Abb. 3.2	Ein Würfel aus Dreiecken	42
Abb. 3.3	Ein Quadrat aus Dreiecken	43
Abb. 3.4	Ein dreidimensionaler Würfel	44
Abb. 3.5	Ein dreidimensionaler Würfel mit geänderten Kamera- und Lichteinstellungen	44
Abb. 3.6	Ein Video auf einem Text	45
Abb. 4.1	Ein abgefangener Fehler bei der Datenbindung	49
Abb. 4.2	Das Ausgabefenster des Debuggers mit einer Meldung zu einer fehlenden Quelle für die Datenbindung	51
Abb. 4.3	Der gedrehte Würfel	53
Abb. 4.4	Die Vorschläge der Livecodeanalyse für das Implementieren der Schnittstelle	55
Abb. 4.5	Der Schieberegler im Einsatz	58

E. Tabellenverzeichnis

Tab. 1.1	Eigenschaften für die Schaltflächen	8
Tab. 2.1	Mausereignisse der WPF	17

F. Codeverzeichnis

Code 1.1	Die XAML-Anweisungen für die Symbolleiste	7
Code 1.2	Die Methode für das Öffnen	10
Code 1.3	Die Methoden für das Anklicken der Schaltflächen.....	11
Code 1.4	Die geänderte Methode für das Symbol	12
Code 1.5	Die Methode zum Verändern der Lautstärke	13
Code 1.6	Das Ändern der Lautstärke über das Mausrad	14
Code 2.1	Die Verarbeitung von Mausereignissen	18
Code 2.2	Ein erstes, sehr einfaches Malprogramm	19
Code 2.3	Die XAML-Anweisungen für das Symbol in der Symbolleiste für den Schnellzugriff.....	22
Code 2.4	Die XAML-Anweisungen für das Anwendungsmenü	23
Code 2.5	Die XAML-Anweisungen für das Register Start mit den Gruppen Zeichnen und Format	23
Code 2.6	Die Zeichenoperationen	28
Code 2.7	Die Farbauswahl	30
Code 2.8	Das Speichern der Datei	30
Code 2.9	Das Laden der Datei	31
Code 3.1	Eine erste einfache Animation.....	37
Code 3.2	Zurücklaufen und Wiederholung einer Animation.....	39
Code 3.3	Eine Animation mit Beschleunigung und Abbremsen.....	40
Code 3.4	Ein Viewport mit Kamera und Licht	41
Code 3.5	Ein Video auf einem Text	45
Code 4.1	Wechselseitige Aktualisierung von Steuerelementen	48
Code 4.2	Eine Datenbindung mit C#-Anweisungen.....	50
Code 4.3	Die Transformation für den Würfel	52
Code 4.4	Ein Rechteck mit Datenbindung	53
Code 4.5	Die Methode Convert()	56
Code 4.6	Das Erzeugen einer Instanz im XAML-Code	57
Code 4.7	Das Multibinding für das Rechteck	58
Code 4.8	Die Methode Convert() für die Klasse FarbKonverter	59
Code 4.9	Das Erstellen einer Instanz für den Konverter	59
Code 4.10	Das Setzen der Eigenschaften Converter	60

G. Sachwortverzeichnis

#	
3D-Modell	43
A	
Abhängigkeitseigenschaft	36
Animation	36
Tempo der	38
Attribut	56
Attributbeschreibung	57
B	
Binding-Objekt	48
C	
Canvas	18
D	
Datenbindung	48
Richtung der	49
Dialog	
zur Farbauswahl	29
G	
Grafik	
dreidimensionale	40
H	
Hintergrund	
festlegen	24
K	
Kamera	41
Koordinatensystem	
dreidimensionales	40
Kultur	54
L	
Lautstärke	
ändern	13
Lichtquelle	41
M	
Material	41
Mausereignis	
der WPF	17
Mausklicks	
Anzahl der, ermitteln	18
Mauskoordinaten	
abfragen	17
Mausrad	
..... 14	
Maustaste	
abfragen	18
Mauszeiger verändern	
..... 32	
Mediadei	
abspielen	11
Menüband	
..... 21	
Multibinding	
..... 58	
MultiValueConverter	
..... 58	
R	
RGB-Modell	
..... 25	
S	
Schaltfläche	
Eigenschaften für die	8
Schieberegler	
..... 9	
Schnittstelle	
automatisch implementieren	55
IValueConverter	54
Source	
..... 49	
Statusleiste	
..... 9	
Symbolleiste	
..... 6	
T	
Target	
..... 49	
Transformation	
..... 52	
V	
Viewport	
..... 41	
W	
Wiedergabe	
abbrechen	11
unterbrechen	11
von Audio- und Videodateien	3

X

XAML-Code

Instanz einer Klasse im, erstellen .	57
XAMLReader	30
XAMLWriter	30

Z

Zeichnen

mit der Maus	19
--------------------	----

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP15D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Schicken Sie für die Lösungen bitte jeweils das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

Beschreiben Sie bei allen Aufgaben zusätzlich, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Änderungen Sie vornehmen beziehungsweise welche Steuerelemente Sie verwenden.

- Ergänzen Sie den Mediaplayer aus diesem Studienheft um Funktionen zum Vor- und Zurückspulen. Beim Vorspulen soll 10 Sekunden nach vorn gesprungen werden, beim Zurückspulen 10 Sekunden nach hinten.

Fügen Sie dazu im ersten Schritt entsprechende Symbole in das Menüband ein. Bilder für die Symbole finden Sie bei den Beispielen im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Ordner **\cshp15d_icons** unter den Namen **vor.png** und **zurueck.png**.

Die aktuelle Position im Medium erhalten Sie über die Eigenschaft `Position` der Klasse `MediaElement`. Über diese Eigenschaft können Sie die Position auch verändern. Bitte beachten Sie aber, dass die Eigenschaft einen Wert vom Typ `TimeSpan` liefert. Um diesen Wert zu verändern, können Sie Methoden wie `Add()` und `Subtract()` benutzen. Die Methoden erwarten allerdings ebenfalls wieder eine Angabe von Typ `TimeSpan` als Argument.

Sie können aber über die Methode `TimeSpan.FromSeconds()` sehr einfach Sekunden in das passende Format umrechnen lassen. Dazu übergeben Sie die Anzahl der Sekunden an die Methode. Weitere Informationen zum Umgang mit der Struktur `TimeSpan` finden Sie in der Hilfe.

Stellen Sie sicher, dass die Funktionen zum Vor- und Zurückspulen nur dann aufgerufen werden können, wenn aktuell ein Medium geladen ist.

35 Pkt.

2. Erweitern Sie das Malprogramm aus diesem Studienheft um eine Funktion zum Zeichnen eines gefüllten Kreises. Der Kreis soll dabei mit der Farbe gefüllt werden, die aktuell auch für den Stift ausgewählt ist.

Diese Funktion soll ebenfalls über das Menüband aufgerufen werden. Eine Grafik für das Symbol finden Sie bei den Beispielen auf Ihrer Online-Lernplattform unter dem **kreisgefüllt.gif**.

15 Pkt.

3. Erstellen Sie ein Programm, das einen Kreis und ein Quadrat animiert. Der Kreis soll dabei immer größer werden und das Quadrat immer kleiner. Die Animationen sollen jeweils 10 Mal wiederholt werden und automatisch wieder zurücklaufen.

Alle weiteren Einstellungen für die Animation wie die Größe oder die Geschwindigkeit können Sie selbst festlegen.

15 Pkt.

4. Erstellen Sie ein Programm, in dem die Werte aus zwei Eingabefeldern verkettet und in einem Label angezeigt werden. Der Wert in dem Label soll dabei über eine Datenbindung beschafft werden.

35 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Ein Webbrowser

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1019N01

CSHP16D

Objektorientierte Software-Entwicklung mit C#

Ein Webbrowser

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Ein Webbrowser

Inhaltsverzeichnis

Einleitung	1
1 Das Grundgerüst des Browsers	3
1.1 Vorüberlegungen	3
1.2 Der Rohbau des Fensters	4
Zusammenfassung	13
2 Die Navigationsfunktionen	15
2.1 Erstellen der Menüleiste	15
2.2 Das Öffnen einer neuen Seite	16
2.3 Das Menü Navigation	22
Zusammenfassung	24
3 Die Symbolleisten	26
3.1 Die Symbolleiste Navigation	26
3.2 Die Symbolleiste Adresse	28
3.3 Der Feinschliff	31
Zusammenfassung	35
4 Ein paar Extras	36
4.1 Das Menü Ansicht	36
4.2 Sonderfunktionen für die Liste	39
4.3 Die Statusleiste	40
4.4 Ein paar Kleinigkeiten	43
Zusammenfassung	43
5 Der Einsatz von Commands	46
Zusammenfassung	48
Schlussbetrachtung	49

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	50
B.	Glossar	52
C.	Literaturverzeichnis	53
D.	Abbildungsverzeichnis	54
E.	Tabellenverzeichnis	55
F.	Codeverzeichnis	56
G.	Sachwortverzeichnis	57
H.	Einsendeaufgabe	59

Einleitung

In diesem Studienheft werden wir einen kleinen Webbrowser als WPF-Anwendung erstellen, der in einer Liste alle besuchten Seiten einer Sitzung mitschreibt. Die Bedienung des Programms soll über Menüs und verschiedene Symbolleisten erfolgen.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie ein komplexes Layout über ein Grid umsetzen,
- wie Sie Bereiche in einem Grid über einen GridSplitter verändern lassen können,
- wie Sie Internet-Seiten in Ihren eigenen Programmen anzeigen lassen,
- wie Sie Menüleisten in einer WPF-Anwendung erstellen,
- wie Sie in eigenen Programmen zwischen Internet-Seiten navigieren,
- wie Sie gezielt auf eine bestimmte Taste in einem Eingabefeld reagieren,
- wie Sie Markierungen in einem Menü setzen,
- wie Sie Symbolleisten zur Laufzeit ein- und ausblenden,
- wie Sie eine Fortschrittsanzeige umsetzen,
- wie Sie Ihre Anwendungen mit einem eigenen Symbol versehen und
- wie Sie Commands implementieren.

Christoph Siebeck

1 Das Grundgerüst des Browsers

In diesem Kapitel erstellen wir das Grundgerüst unseres Browsers.

1.1 Vorüberlegungen

Beginnen wir mit einigen Vorüberlegungen, wie der Browser grundsätzlich aussehen und arbeiten soll.

- Das Fenster soll aus zwei Bereichen bestehen: Links soll die Liste der bisher besuchten Seiten erscheinen und rechts die Anzeige der Webseite. Die beiden Bereiche sollen gleichzeitig angezeigt werden und beliebig in der Breite verändert werden können.
- Die Liste der besuchten Seiten erstellen wir mit einem Steuerelement **ListBox**. Bei jedem Wechsel zu einer Seite soll die URL der Seite an die bereits vorhandenen Einträge in der Liste angehängt werden.

Zur Auffrischung:

Eine URL ist eine eindeutige Adresse einer Ressource im Internet. URL steht für *Uniform Resource Locator* – übersetzt etwa „einheitlicher Ressourcenbezeichner“.



- Die Anzeige der Webseiten im rechten Bereich des Fensters erfolgt durch das Steuerelement **WebBrowser**. Dieses Steuerelement bietet alle grundlegenden Funktionen für die Anzeige von Internet-Seiten und auch für die Navigation an.
- Die Bedienung des Programms soll über eine Menüleiste, zwei Symbolleisten und ein Kontext-Menü erfolgen. Außerdem erstellen wir noch eine Statusleiste, die den Fortschritt beim Laden einer Seite anzeigt.
- Neben den Grundfunktionen wie Öffnen und Navigieren soll der Anwender zusätzlich die Möglichkeit haben, bestimmte Bereiche im Fenster nach Wunsch einzublenden.

Die Endversion des Webbrowsers soll ungefähr so aussehen:

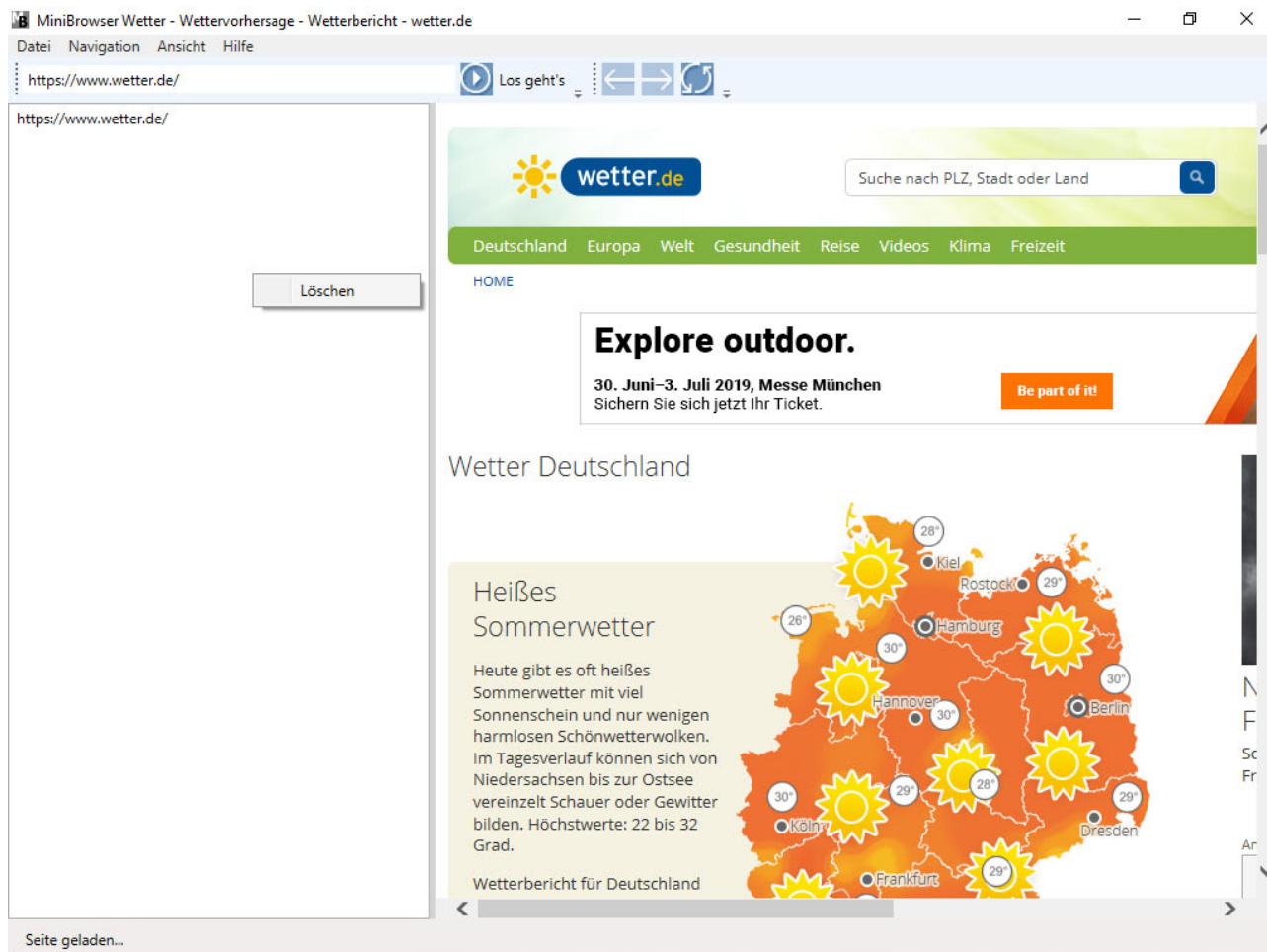


Abb. 1.1: Die Endversion des Webbrowsers



Den fertigen Webbrowser finden Sie im Projekt **Minibrowser** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

In dem anderen Ordner befindet sich ebenfalls ein Projekt mit dem Namen **Minibrowser**. Es enthält eine Variante des Programms mit Commands. Mehr zu dieser Variante erfahren Sie am Ende des Studienhefts.

1.2 Der Rohbau des Fensters

Beginnen wir jetzt mit der praktischen Umsetzung.

Legen Sie bitte ein neues Projekt für eine WPF-Anwendung an. Als Projektnamen können Sie zum Beispiel **Minibrowser** verwenden.

Setzen Sie dann die Eigenschaft **WindowState** in der Gruppe **Allgemein** des Fensters auf **Maximized**, damit das Fenster direkt nach dem Start als Vollbild angezeigt wird. Den Text in der Titelleiste lassen Sie bitte unverändert stehen. Wir werden ihn gleich dynamisch nach dem Laden einer Webseite setzen.

Beim Layout werden wir in diesem Projekt einen anderen Weg gehen als bisher. Wir fügen die Steuerelemente nicht mehr gemeinsam in eine Zelle des Grids ein und setzen dann die Abstände vom Rand sowie weitere Eigenschaften für die Anzeige. Stattdessen erstellen wir ein Grid mit mehreren Zeilen und Spalten. Für jedes einzelne Steuerelement verwenden wir dann eine oder mehrere Zellen.

Das hat den großen Vorteil, das sich auch bei komplexeren Layouts Größenänderungen an den Steuerelementen sehr einfach umsetzen lassen. Denn die Größe eines Steuerelements orientiert sich bei dieser Variante an der Größe der Zelle, in der es sich befindet. Und die Größe einer Zelle können wir automatisch setzen lassen.

Außerdem können wir so sehr einfach die Bereiche umsetzen, die gleichzeitig angezeigt werden sollen und sich beliebig in der Breite verändern lassen. Wir erstellen dazu zwei Spalten, die wir über einen **GridSplitter** anpassen können. Über diesen Splitter kann ein Anwender die Breite beziehungsweise die Höhe von Zellen mit der Maus verändern. Beim Vergrößern einer Zelle werden die anderen Zellen entsprechend verkleinert – und umgekehrt.

In unserem Beispiel benötigen wir für das Layout insgesamt vier Zeilen und drei Spalten. In die erste Zeile setzen wir die Menüleiste. Die zweite Zeile enthält die Symbolleisten. Die dritte Zeile soll das Listenfeld und die Anzeige für den Webbrowser aufnehmen. Die vierte Zeile schließlich benutzen wir für die Statusleiste.

Die drei Spalten brauchen wir eigentlich nur für die Anzeige der dritten Zeile. In die linke Spalte setzen wir hier das Listenfeld, in die mittlere den Splitter zum Verändern der Größe und in die rechte die Anzeige für den Webbrowser. In den anderen Zeilen lassen wir die Steuerelemente spaltenübergreifend darstellen – also nur in einer einzigen Spalte.

Der schematische Aufbau des Grids sieht also so aus:

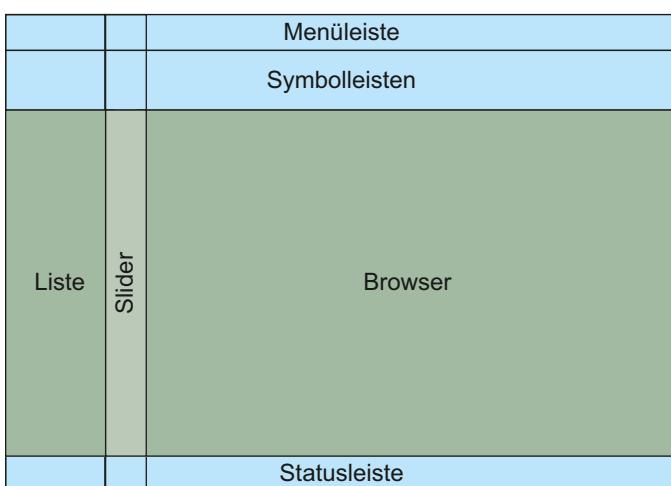


Abb. 1.2: Der schematische Aufbau des Grids

Beginnen wir jetzt mit der praktischen Umsetzung. Dazu erstellen wir die Zeilen und Spalten direkt im XAML-Code. Die entsprechenden Anweisungen finden Sie im folgenden Code 1.1:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" MinWidth="100"/>
    <ColumnDefinition Width="5"/>
    <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
```

Code 1.1: Das Grid

Zuerst werden in unserem Beispiel die Definitionen für die Zeilen im Tag `<Grid.RowDefinitions>`¹ angegeben. Jede einzelne Zeile wird dabei durch ein Tag `<RowDefinition>` beschrieben. Die Höhe können Sie dabei entweder durch einen absoluten Wert, durch ein Sternchen * oder den Text Auto angegeben. Das Sternchen * steht dabei für einen proportionalen Wert und der Text Auto für eine automatische Berechnung abhängig von der Größe der Steuerelemente in der Zeile.



Bitte beachten Sie:

Alle drei Angaben müssen in Anführungszeichen gesetzt werden.

In unserem Beispiel sollen sich also die erste, die zweite und die vierte Zeile an der Größe der Steuerelemente in der Zeile orientieren. Die dritte Zeile soll den verbleibenden Platz einnehmen. Daher geben wir hier eine proportionale Größe an.

Danach folgen im Tag `<Grid.ColumnDefinitions>`² die Spalten. Hier wird jede Spalte durch ein Tag `<ColumnDefinition>` beschrieben. Für die Breite können Sie dabei ebenfalls wieder einen festen Wert, das Sternchen * für eine proportionale Breite und den Text Auto für eine automatische Breite verwenden.

Die mittlere Spalte soll in unserem Beispiel fünf Pixel breit sein. Die erste und die dritte Spalte teilen sich den verbleibenden Rest proportional auf. Die erste Spalte soll dabei einen Teil bekommen und die zweite Spalte zwei Teile. Außerdem legen wir für die erste Spalte eine Mindestbreite von 100 Pixeln fest.

1. Row bedeutet übersetzt „Zeile“.
2. Column bedeutet übersetzt „Spalte“.

Geben Sie die Anweisungen aus dem vorigen Code jetzt im XAML-Editor zwischen den Tags für das Grid ein. Das Fenster könnte dann so aussehen:

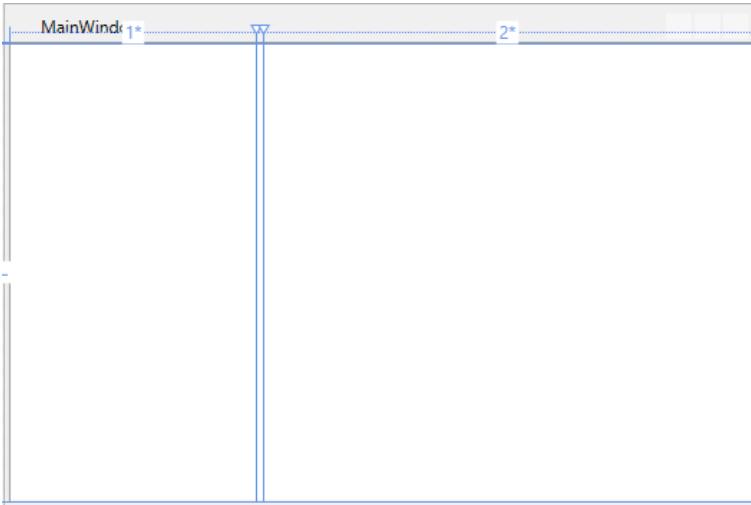


Abb. 1.3: Das Fenster im ersten Rohbau

Hinweis:

Sie können die Definitionen für die Spalten und Zeilen eines Grids auch über das Eigenschaftenfenster festlegen. Dazu bearbeiten Sie die Eigenschaften **ColumnDefinitions** und **RowDefinitions** mit dem Auflistungs-Editor. Sie finden die Eigenschaften bei den erweiterten Eigenschaften in der Gruppe **Layout** für das Grid. Das Bearbeiten über den Sammlungs-Editor ist allerdings recht umständlich.

Fügen wir jetzt die Steuerelemente ein. Für die Menü-, Symbol- und Statusleisten erstellen wir dabei zunächst nur leere Gerüste. Diese Gerüste werden wir im weiteren Verlauf mit Leben füllen.

Übernehmen Sie bitte die XAML-Anweisungen aus dem folgenden Code und setzen Sie sie hinter die Definitionen der Spalten. Was sich genau hinter den Anweisungen verbirgt, erklären wir Ihnen im Anschluss.

```
<!-- Die Menüleiste mit einem Eintrag -->
<Menu Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3">
    <MenuItem Header="_Datei">
        </MenuItem>
</Menu>
<!-- Die Symbolleisten -->
<ToolBarTray Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="3">
    <ToolBar>
        <Button>
            <Image>
                </Image>
            </Image>
        </Button>
    </ToolBar>
    <ToolBar>
        <Button>
            <Image>
                </Image>
            </Image>
        </Button>
    </ToolBar>
```

```

        </Button>
    </ToolBar>
</ToolBarTray>
<!-- Das Listenfeld mit einem Eintrag -->
<ListBox Grid.Column="0" Grid.Row="2">
    <ListBoxItem Content="Eintrag"></ListBoxItem>
</ListBox>
<!-- Der Splitter -->
<!-- bitte in einer Zeile eingeben -->
<GridSplitter Grid.Column="1" Grid.Row="2" Width ="5"
Background="LightGray"/>
<!-- Das Steuerelement für den WebBrowser-->
<WebBrowser Grid.Column="3" Grid.Row="2">
</WebBrowser>
<!-- Die Statusleiste mit einem Label -->
<StatusBar Grid.Column="0" Grid.Row="3" Grid.ColumnSpan="3">
    <Label Content="Eintrag"></Label>
</StatusBar>

```

Code 1.2: Die Gerüste für die Bedienelemente

Schauen wir uns die Anweisungen der Reihe nach an. Mit den ersten drei Anweisungen

```

<Menu Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3">
    <MenuItem Header="Datei">
    </MenuItem>
</Menu>

```

erzeugen wir eine Menüleiste mit einem Eintrag **Datei**. Die Menüleiste wird dabei im Tag `<Menu>` beschrieben und der Eintrag im Tag `<MenuItem>`. Über die Eigenschaften **Grid.Column** und **Grid.Row** legen wir die Position im Grid fest. In unserem Beispiel soll das Menü in der ersten Spalte in der ersten Zeile angezeigt werden und außerdem drei Spalten umfassen. Das legen wir über die Eigenschaft **Grid.ColumnSpan³** fest.



Bitte beachten Sie:

Die Nummerierung der Spalten und Zeilen beginnt jeweils bei 0.

Für die erste Zeile beziehungsweise die erste Spalte können Sie die Angaben auch weglassen. Dann wird der Standardwert 0 benutzt.

Danach folgt die Beschreibung der Symbolleisten. Sie liegen in einem Steuerelement vom Typ **ToolBarTray**. Es wird in der ersten Spalte der zweiten Zeile angezeigt und deckt ebenfalls drei Spalten ab.

Die Anweisungen

```

<ListBox Grid.Column="0" Grid.Row="2">
    <ListBoxItem Content="Eintrag"></ListBoxItem>
</ListBox>
<GridSplitter Grid.Column="1" Grid.Row="2" Width ="5"
Background="LightGray"/>

```

3. *Span* lässt sich mit „Bereich“ übersetzen.

```
<WebBrowser Grid.Column="3" Grid.Row="2">  
</WebBrowser>
```

erzeugen die dritte Zeile. Links befindet sich das Listenfeld. Es enthält zunächst einmal einen Eintrag vom Typ **ListBoxItem** als Platzhalter. Dann folgt der Splitter. Er ist fünf Pixel breit und bekommt einen hellgrauen Hintergrund.

Wenn Sie keine Breite oder Höhe für einen GridSplitter angeben, wird er nicht angezeigt.



In die dritte Spalte setzen wir das Steuerelement **WebBrowser**.

Die Anweisungen

```
<StatusBar Grid.Column="0" Grid.Row="3" Grid.ColumnSpan="3">  
  <Label Content="Eintrag"></Label>  
</StatusBar>
```

schließlich erzeugen die Statusleiste. Hier lassen wir ebenfalls erst einmal einen Platzhalter über ein Label anzeigen.

Das Fenster könnte jetzt so aussehen:

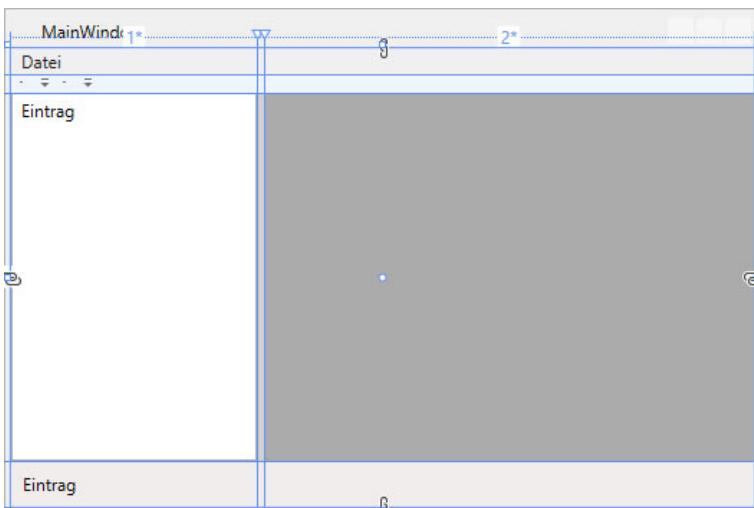


Abb. 1.4: Das Fenster mit den Steuerelementen

Hinweis:

Sie können die Steuerelemente auch aus der Toolbox in das Fenster einfügen und dann mit der Maus oder über die Eigenschaften **Row** und **Column** in der Gruppe **Layout** im Eigenschaftenfenster positionieren. Allerdings werden beim Einfügen aus der Toolbox zum Teil andere Standardeinstellungen benutzt, die sich auch auf das Layout auswirken können.

Nun wird es Zeit für einen ersten Test des Programms. Geben Sie bitte bei der Eigenschaft **Source** in der Gruppe **Allgemein** des WebBrowser-Steuerelements die Adresse einer beliebigen Internet-Seite an – zum Beispiel <https://www.wetter.de>.



Sie müssen auch das Protokoll selbst angeben. Nur die Angabe www.wetter.de ohne Protokoll führt zu einem Fehler.

Speichern Sie dann die Änderungen und lassen Sie das Programm ausführen.



Abb. 1.5: Der Webbrowser in Aktion

Hinweis:

Die Seite kann nur dann angezeigt werden, wenn eine Verbindung zum Internet besteht. Falls diese Verbindung bei Ihnen nicht automatisch hergestellt wird, bauen Sie die Verbindung bitte vor dem Test auf.

Die Seite ändert ständig ihr Aussehen. Bei Ihnen wird sie daher anders aussehen als in der Abbildung.

Bei der Anzeige von einigen Seiten erscheinen unter Umständen Meldungen zu Scriptfehlern. Diese Meldungen können Sie einfach ignorieren.

Überprüfen Sie beim Probelauf auch, wie die Steuerelemente auf Größenänderungen des Fensters reagieren und ob sich die Breite der Spalten über den Splitter verändert lässt. Bei den Größenänderungen sollten sich die Steuerelemente dank des Grids automatisch

anpassen. Die Veränderungen über den Splitter funktionieren allerdings noch nicht wie gewohnt. Wenn Sie den Splitter nach links ziehen, geschieht nichts. Beim Ziehen nach rechts verkleinern sich beide Spalten.

Dieses Verhalten lässt sich aber leicht anpassen. Ändern Sie die Eigenschaft **ResizeBehavoir⁴** des Splitters in der Gruppe **Allgemein** in `PreviousAndNext`. Damit wirken Änderungen am Splitter auf die vorige und die nächste Spalte. Sie können die Eigenschaft natürlich auch direkt im XAML-Code setzen.

Zur Erinnerung:

Unser Splitter befindet sich in einer eigenen Spalte. Diese Spalte wollen wir über den Splitter aber nicht verändern. Daher müssen wir die Änderungen für die vorige Spalte mit dem Listenfeld und die folgende Spalte mit dem WebBrowser-Steuerelement durchführen.



Im letzten Schritt des Rohbaus sorgen wir jetzt noch dafür, dass die URL der Seite in dem Listenfeld im linken Panel angezeigt wird und dass der Titel der Seite in der Titelleiste des Fensters erscheint.

Dazu verwenden wir das Ereignis **Navigated⁵** des WebBrowser-Steuerelements. Es tritt ein, wenn eine Seite gefunden und das Laden begonnen wurde. Um die URL der Seite in die Liste aufzunehmen, hängen wir einfach mit der Methode `Items.Add()` den aktuellen Wert der Eigenschaft **Source** des WebBrowser-Steuerelements an den Inhalt der Liste an.

Den Text in der Titelleiste bauen wir aus der Zeichenkette „MiniBrowser“ und der Eigenschaft **Document.Title** des WebBrowser-Steuerelements zusammen. Damit sorgen wir dafür, dass nach jedem Laden der Seite sowohl der Name unseres Programms als auch der Titel der aktuellen Seite in der Titelleiste des Fensters angezeigt werden.

Damit wir aus dem Code auf die Steuerelemente zugreifen können, vergeben Sie bitte zunächst Namen an das Listenfeld und das WebBrowser-Steuerelement. Wir benutzen in unserem Beispiel die Bezeichner `liste` und `browser`. Wenn Sie andere Bezeichner verwenden, müssen Sie die Codes entsprechend anpassen.

Die Anweisungen für die Methode `Browser_Navigated()` finden Sie im folgenden Code 1.3.

```
//die Adresse in das Listenfeld schreiben
liste.Items.Add(browser.Source);
//den Text in der Titelleiste zusammenbauen
Title = "MiniBrowser " + ((dynamic)browser.Document).Title;
```

Code 1.3: Die Anweisungen für die Methode `Browser_Navigated()`

Über die Angabe `dynamic` wird das Objekt in den sehr allgemeinen Typ `dynamic` gecastet. Er steht für nahezu beliebige Inhalte und wird auch beim Übersetzen nicht geprüft. Ohne diese Angabe ist kein Zugriff auf die Eigenschaft `Title` möglich.



4. `ResizeBehavoir` lässt sich mit „Verhalten bei Größenänderung“ übersetzen.

5. `Navigated` bedeutet so viel wie „navigiert“.

Löschen Sie jetzt noch den Platzhalter aus dem Listenfeld. Übernehmen Sie dann die Anweisungen aus dem vorigen Code und testen Sie das Programm noch einmal. Die Anzeige sollte jetzt so aussehen:



Abb. 1.6: Die Anzeige im Listenfeld und in der Titelleiste



Bitte beachten Sie:

Der Eintrag im Listenfeld und auch der Text in der Titelleiste erscheinen erst dann, wenn die Seite vollständig geladen wurde. Bei einer langsamen Verbindung kann es daher durchaus einen Moment dauern, bis die Anzeige aktualisiert wird.

Unter Umständen erscheinen beim Laden einer Seite mehrere identische Einträge im Listenfeld links im Fenster. Das liegt in der Regel daran, dass von der aufgerufenen Seite mehrere andere Seiten nachgeladen werden.

Probieren Sie auch einmal aus, ob die Anzeige beim Aufruf einer anderen Seite korrekt funktioniert. Klicken Sie dazu einfach auf einen Link in der angezeigten Webseite. Nach dem Laden der Seite sollten sich die Einträge im Listenfeld und gegebenenfalls auch in der Titelleiste entsprechend ändern.

Damit ist der Rohbau unserer Anwendung zunächst einmal fertig. Im nächsten Kapitel werden wir eine Menüleiste mit Funktionen zum Öffnen anderer Seiten und zum Navigieren erstellen.

Zusammenfassung

In einem Grid können Sie Steuerelemente in Spalten und Zeilen darstellen.

Über einen GridSplitter kann ein Anwender die Breite und Höhe einer Spalte beziehungsweise Zeile verändern.

Mit dem Steuerelement **WebBrowser** können Sie Internetseiten in einer Anwendung anzeigen lassen. Die zuerst angezeigte Seite legen Sie dabei über die Eigenschaft **Source** fest.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Sie wollen ein Steuerelement in die erste Spalte in der zweiten Zeile eines Grids setzen. Welche Werte müssen Sie für die Eigenschaften **Grid.Column** und **Grid.Row** angeben?

- 1.2 Sie haben einen GridSplitter in eine eigene Zeile eines Grids eingefügt. Auf welche Zeilen wirkt der Splitter in der Standardeinstellung? Über welche Eigenschaft können Sie das Verhalten verändern?

- 1.3 Welches Ereignis tritt beim Steuerelement **WebBrowser** ein, wenn eine Seite vollständig geladen wurde?

- 1.4 Mit welcher Eigenschaft können Sie den Titel einer Seite ermitteln, die aktuell in einem WebBrowser-Steuerelement angezeigt wird? Was ist beim Zugriff auf die Eigenschaft besonders wichtig? Bitte beachten Sie, dass der Titel ermittelt werden soll und nicht die URL.

2 Die Navigationsfunktionen

In diesem Kapitel werden wir die wichtigsten Navigationsfunktionen für unseren Browser erstellen. Wir sorgen unter anderem dafür, dass eine neue Seite über einen Dialog geöffnet werden kann, und stellen dem Anwender Funktionen zum Navigieren durch bereits angezeigte Seiten zur Verfügung. Der Aufruf der entsprechenden Funktionen soll zunächst über eine Menüleiste erfolgen.

Hinweis:

Die Symbolleisten mit den Funktionen erstellen wir im nächsten Kapitel.

2.1 Erstellen der Menüleiste

Beginnen wir jetzt mit dem Erstellen der Menüleiste. Einen ersten groben Entwurf haben wir ja bereits mit den Zeilen

```
<Menu Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3">
    <MenuItem Header="_Datei">
        </MenuItem>
    </Menu>
```

erstellt. Das Tag `<Menu>` beschreibt die Menüleiste und das Tag `<MenuItem>` einen Menüeintrag. Den Text des Eintrags legen Sie über die Eigenschaft **Header** fest. Der Unterstrich `_` markiert dabei den Buchstaben, über den das Menü später auch mit der Tastatur geöffnet werden kann.

Hinweis:

Wie bei einer Windows Forms-Anwendung auch müssen Sie immer auch die Taste `Alt` drücken, um ein Menü zu öffnen. Die Taste alleine öffnet das Menü nur dann, wenn ein Eintrag in der Menüleiste markiert ist und sämtliche Menüs geschlossen sind.

Einträge in einem Menü legen Sie durch ein Steuerelement **MenuItem** an, das sich innerhalb des Tags für das Menü befindet. Unsere Menüleiste mit einem Eintrag **Beenden** für das Menü **Datei** sieht im XAML-Code dann so aus:

```
<Menu Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3">
    <MenuItem Header="_Datei">
        <MenuItem Header="_Beenden" />
    </MenuItem>
</Menu>
```

Hinweis:

Sie können die Einträge auch über die Eigenschaft **Items** für die Menüleiste beziehungsweise ein Menü erstellen. Im Sammlungs-Editor wählen Sie dann im Kombinationsfeld unten im Fenster das gewünschte Element aus – zum Beispiel eben **MenuItem** für einen Menüeintrag.

In der Menüleiste im Fenster sollte nun der Eintrag **Datei** erscheinen. Der Buchstabe D für die Auswahl des Menüs über die Tastatur wird dabei unterstrichen dargestellt. Durch einen Mausklick auf den Menüeintrag im Designer können Sie jetzt auch das Menü aufklappen lassen.

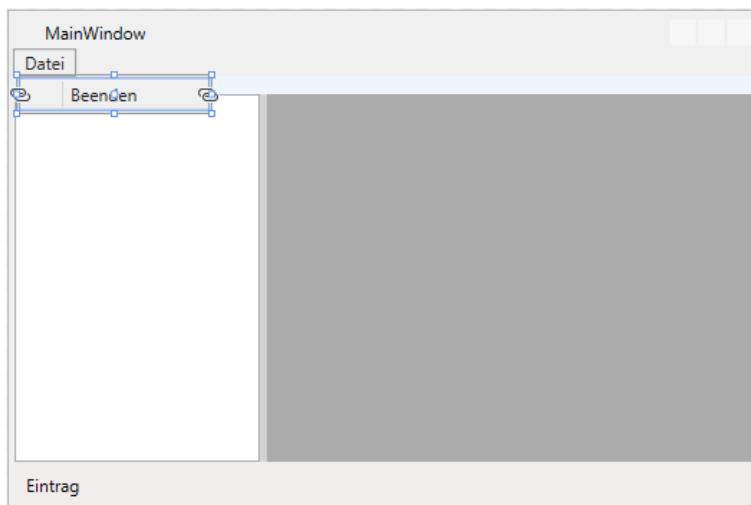


Abb. 2.1: Das geöffnete Menü Datei im Designer

Damit wir die verschiedenen Menüeinträge später eindeutig unterscheiden können, vergeben Sie bitte noch einen Namen an den neuen Eintrag – zum Beispiel `dateiBeenden`.

Nun müssen wir noch dafür sorgen, dass beim Anklicken des Eintrags **Beenden** im Menü **Datei** die Anwendung geschlossen wird. Das erfolgt wie gewohnt über das Ereignis **Click** des Steuerelements.

Markieren Sie bitte den Menüeintrag **Beenden**. Erstellen Sie dann über das Eigenschaftenfenster eine Methode für das Ereignis **Click** und schließen Sie in dieser Methode die Anwendung über die Anweisung `Close()`.

Speichern Sie anschließend die Änderungen und testen Sie das gerade erstellte Menü. Probieren Sie dabei auch die Auswahl über die Tastatur aus.

2.2 Das Öffnen einer neuen Seite

Als nächste Funktion wollen wir jetzt das Öffnen einer neuen Seite über das Menü **Datei** umsetzen. Die Eingabe der Adresse soll dabei in einem eigenen Fenster über ein Eingabefeld erfolgen.

Das Erstellen dieses Fensters und auch das Anzeigen sind nicht weiter schwierig. Wir setzen dabei im Wesentlichen genau dieselben Techniken ein, die wir zum Beispiel bereits bei der Bestenliste in unserem Pong-Spiel umgesetzt haben.

Erstellen Sie bitte zunächst mit der Funktion **Projekt/Fenster hinzufügen ...** ein neues Fenster mit dem Namen **UrlOeffnenDialog**. Fügen Sie in dieses Fenster ein Label, ein Eingabefeld und zwei Schaltflächen ein. Setzen Sie den Text im Label bitte auf **Öffnen**.

Den Text im Eingabefeld können Sie löschen. Die linke Schaltfläche soll den Text **Öffnen** erhalten und die rechte den Text **Abbrechen**. Positionieren Sie die Steuerelemente ungefähr so wie in der folgenden Abbildung und verkleinern Sie das Fenster ein wenig.

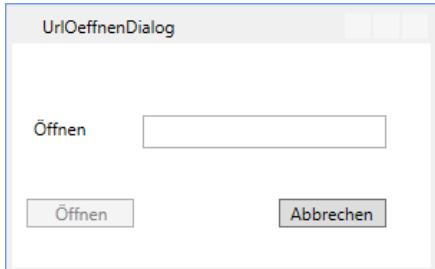


Abb. 2.2: Das Fenster zum Öffnen einer URL

Setzen Sie dann die Eigenschaften für das Fenster wie in der folgenden Tab. 2.1.

Tab. 2.1: Die Eigenschaften für das zweite Fenster

Eigenschaft	Wert
WindowState	ToolWindow
Title	Öffnen...
Topmost	Ja
ResizeMode	NoResize

Das Setzen der Eigenschaft **DialogResult** erfolgt bei einer WPF-Anwendung – anders als bei einer Windows-Forms-Anwendung – nicht mehr direkt über die Eigenschaften einer Schaltfläche. Stattdessen setzen Sie beim Anklicken einer Schaltfläche die Eigenschaft **DialogResult** für das Fenster entweder auf `true` oder auf `false`. In unserem Fall soll die Eigenschaft den Wert `true` bekommen, wenn auf die Schaltfläche **Öffnen** geklickt wird. Beim Anklicken von **Abbrechen** setzen wir die Eigenschaft auf `false`.

Außerdem müssen wir noch eine Methode erstellen, die den Wert des Eingabefelds als Zeichenkette liefert. Die drei Methoden finden Sie im Code 2.1:

```
private void ButtonOeffnen_Click(object sender,
RoutedEventArgs e)
{
    DialogResult = true;
}

private void ButtonAbbrechen_Click(object sender,
RoutedEventArgs e)
{
    DialogResult = false;
}
```

```
//die Methode liefert die eingegebene Adresse zurück
public string LiefereAdresse()
{
    return textBox.Text;
}
```

Code 2.1: Die Methoden für den Dialog

Hinweis:

Die Namen der Methoden für die Schaltflächen in dem Code hängen von den Bezeichnern ab, die Sie vergeben haben.

Übernehmen Sie die Methoden aus dem Code bitte. Denken Sie bei der Methode `LiefereAdresse()` daran, dass Sie die Sichtbarkeit `public` verwenden müssen. Andernfalls ist die Methode nach außen nicht sichtbar und kann nicht aus dem anderen Fenster aufgerufen werden.

Speichern Sie dann alle Änderungen und wechseln Sie wieder in das Fenster unseres Browsers.

Legen Sie anschließend im Menü **Datei** einen neuen Eintrag zum Anzeigen des gerade erstellten Fensters an. Als Text verwenden Sie bitte `Öffnen...` und für den Namen zum Beispiel `dateioeffnen`.

Um die optische Gestaltung des Menüs werden wir uns gleich kümmern. Jetzt sorgen wir erst einmal dafür, dass beim Anklicken des neuen Eintrags das Fenster zum Öffnen erscheint und die Eingaben ausgewertet werden.

Übernehmen Sie dazu bitte die Anweisungen aus dem folgenden Code 2.2 für das Ereignis **Click** des neuen Menüeintrags. Was die einzelnen Zeilen genau bewirken, erklären wir Ihnen im Anschluss.

```
string adresse;
//den eigenen Öffnendialog modal öffnen
UrlOeffnenDialog oeffnenDialog = new UrlOeffnenDialog();
oeffnenDialog.ShowDialog();
//den "Eigentümer" setzen
oeffnenDialog.Owner = this;
//wurde die Schaltfläche Öffnen in dem Dialog angeklickt?
if (oeffnenDialog.DialogResult == true)
{
    //die Adresse über die Methode im Öffnendialog beschaffen
    adresse = oeffnenDialog.LiefereAdresse();
    //wenn die Adresse nicht leer ist, die Seite anzeigen
    if (adresse != string.Empty)
    {
        //lässt sich die Adresse umwandeln
        try
        {
            browser.Navigate(new Uri(adresse));
        }
    }
}
```

```

        catch (UriFormatException)
    {
        //bitte in einer Zeile eingeben
        MessageBox.Show("Das Format der Adresse " +
            adresse + " ist nicht gültig.", "Fehler");
    }
}
}

```

Code 2.2: Die Anweisungen für das Ereignis Click

Zunächst einmal vereinbaren wir eine lokale Variable `adresse`, die die eingegebene Adresse aufnehmen soll. Damit erleichtern wir uns in den folgenden Anweisungen den Zugriff auf die Adresse.

Anschließend erzeugen wir eine neue Instanz des zweiten Fensters und zeigen es modal an. Das erledigen die beiden Anweisungen

```
UrlOeffnenDialog oeffnenDialog = new UrlOeffnenDialog();
oeffnenDialog.ShowDialog();
```

Mit der Anweisung

```
oeffnenDialog.Owner = this;
```

setzen wir ausdrücklich den Eigentümer des modalen Fensters auf das Fenster, aus dem der Aufruf erfolgt. Ohne dieses Setzen des Eigentümers kann es bei der WPF zu seltsamen Verhalten kommen – zum Beispiel, dass das untergeordnete Fenster im Hintergrund angezeigt wird.

Mit der Anweisung

```
if (oeffnenDialog.DialogResult == true)
```

überprüfen wir, ob die Eigenschaft **DialogResult** des zweiten Fensters den Wert `true` hat. Wenn der Vergleich zutrifft – der Anwender also den Dialog über die Schaltfläche **Öffnen** verlassen hat – rufen wir die Methode `LiefereAdresse()` im zweiten Fenster auf und weisen das Ergebnis unserer Variablen `adresse` zu.

Wenn die Adresse nicht leer ist, lassen wir die Seite mit der Methode `Navigate()`⁶ des WebBrowser-Steuerelements anzeigen. Dazu bauen wir die Zeichenkette in den Typ `Uri` um – auch wenn wir an die Methode direkt eine Zeichenkette übergeben könnten. Dann haben wir allerdings keine Möglichkeit, auf Eingaben in einem ungültigen Format zu reagieren. Und genau das macht unsere `try ... catch`-Konstruktion im vorigen Code.

Bei einer WPF-Anwendung müssen Sie das modale Fenster nicht ausdrücklich schließen. Es wird beim Einsatz der Eigenschaft **DialogResult** automatisch geschlossen.



6. *Navigate* bedeutet übersetzt so viel wie „Navigiere“.

Abschließend nehmen wir jetzt noch zwei kosmetische Korrekturen im Menü **Datei** vor.

Verschieben Sie zunächst gegebenenfalls den Eintrag **Öffnen...** so, dass er ganz oben im Menü angezeigt wird. Das geht am einfachsten, wenn Sie die XAML-Anweisungen an die passende Stelle kopieren. Sie können den Eintrag aber auch im Sammlungs-Editor über das Symbol **Element nach oben verschieben** links neben dem Kombinationsfeld für die Auswahl nach oben schieben.

Fügen Sie abschließend noch eine Trennlinie in das Menü ein. Fügen Sie dazu einfach ein Tag `<Separator />` im XAML-Code zwischen den beiden Einträgen ein. Sie können die Trennlinie aber auch über den Sammlungs-Editor einfügen. Wählen Sie dazu im Kombinationsfeld für die Art des Menüeintrags den Eintrag **Separator** und verschieben Sie die Linie dann so, dass sie zwischen den beiden Menüeinträgen steht.

Das geänderte Menü sollte dann ungefähr so aussehen wie in der Abb. 2.3.

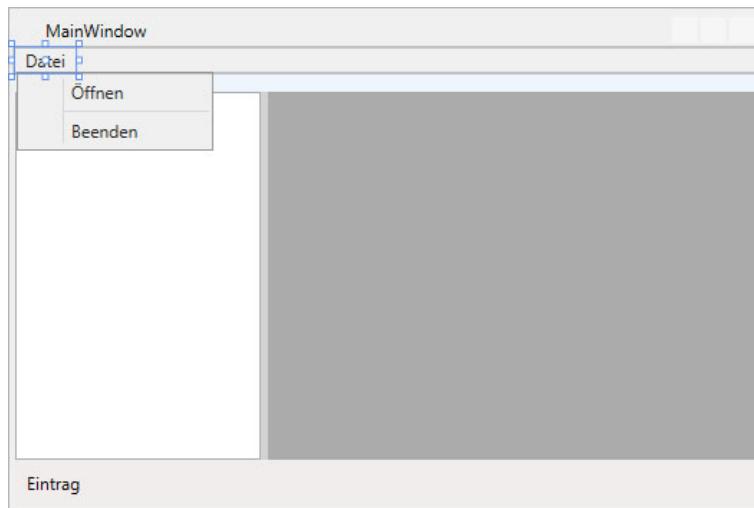


Abb. 2.3: Das geänderte Menü Datei

Speichern Sie alle Änderungen und testen Sie die Erweiterungen.

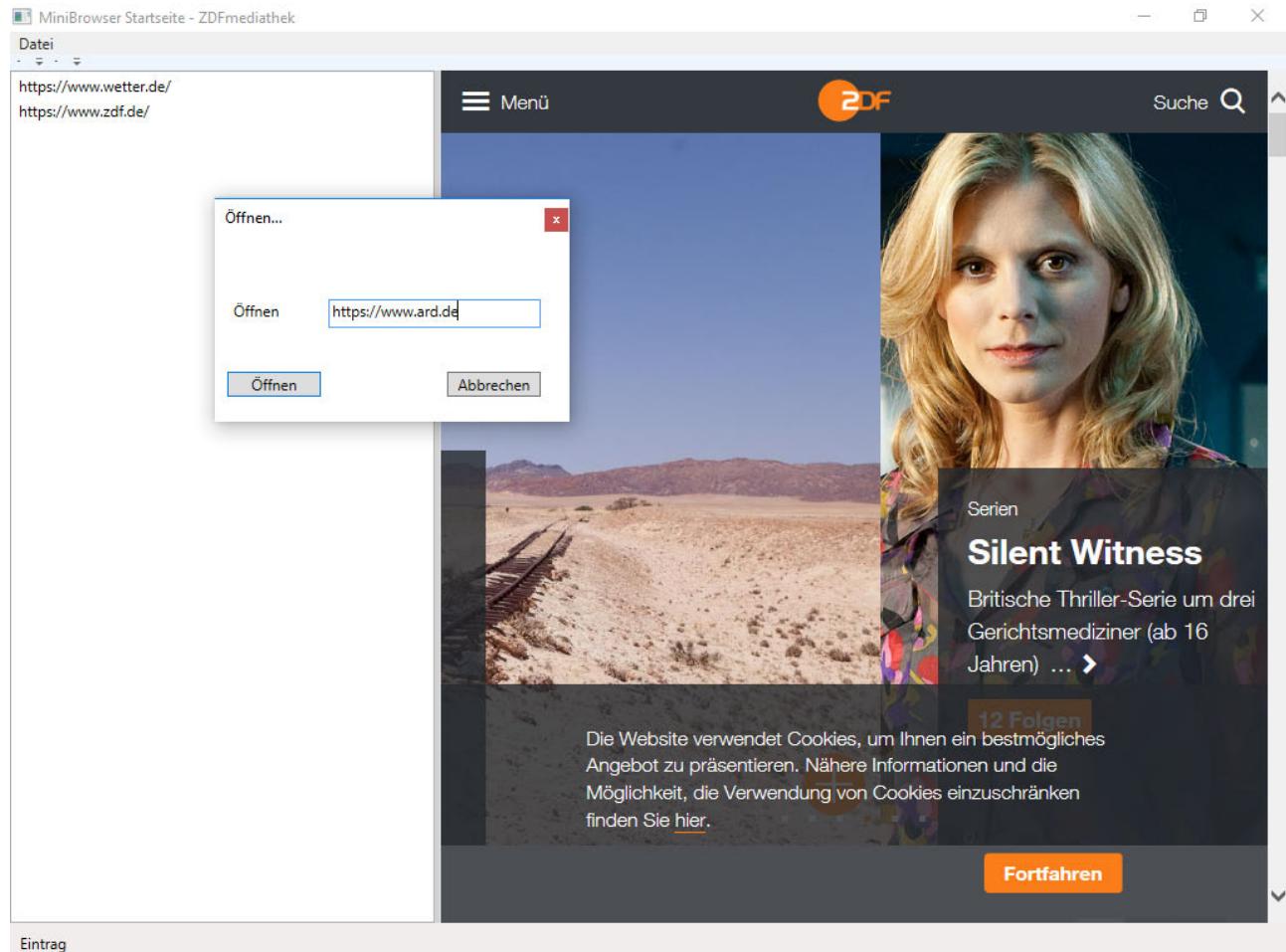


Abb. 2.4: Der Dialog Öffnen im praktischen Einsatz

Sie können den Dialog zum Öffnen einer Webseite auch noch ein wenig eleganter gestalten. Über die Eigenschaft **IsDefault** für eine Schaltfläche können Sie diese Schaltfläche zum Beispiel zur Standardschaltfläche für den Dialog machen. Die Aktion lässt sich dann auch mit der Eingabetaste aufrufen. Die Eigenschaft **IsCancel** sorgt entsprechend dafür, dass Sie die Aktion der Schaltfläche auch über die Taste **Esc** aufrufen können.

Außerdem können Sie die Tabulatorreihenfolge für die Steuerelemente so setzen, dass sie von links oben nach rechts unten durchlaufen werden. Dazu verwenden Sie die Eigenschaft **TabIndex** in der Gruppe **Allgemein**. Die Reihenfolge wird dabei durch fortlaufende Nummern angegeben. Das erste Element erhält die Nummer 0.

Allerdings erhält das Steuerelement mit dem **TabIndex** 0 nicht automatisch auch den Fokus. Sie können ihn aber mit der Methode `Focus()` im Code setzen. Die Anweisung

```
textBox.Focus();
```

setzt den Fokus zum Beispiel auf das Steuerelement mit dem Namen `textBox`. Sie können diese Anweisung zum Beispiel im Ereignis **Loaded** für das Fenster ausführen lassen.

Auch das Abfangen einer leeren Eingabe für die Adresse lässt sich etwas professioneller gestalten – nämlich, indem Sie die Schaltfläche **Öffnen** nur dann aktiv schalten, wenn eine Eingabe in dem Feld gemacht wurde. Schalten Sie dazu die Markierung für die Eigenschaft **IsEnabled** der Schaltfläche in der Gruppe **Allgemein** zunächst ab. Danach

überprüfen Sie dann, ob das Feld einen Inhalt hat, und aktivieren die Schaltfläche gegebenenfalls wieder. Dazu können Sie zum Beispiel das Ereignis **TextChanged**⁷ des Eingabefelds verwenden. Es tritt ein, wenn der Inhalt des Feldes geändert wurde. Die vollständige Methode für das Ereignis **TextChanged** könnte zum Beispiel so aussehen:

```
private void TextBox_TextChanged(object sender,
TextChangedEventArgs e)
{
    //die Schaltfläche Öffnen aktivieren, wenn das Feld nicht leer
    //ist
    if (textBox.Text != string.Empty)
        buttonOeffnen.IsEnabled = true;
    else
        buttonOeffnen.IsEnabled = false;
}
```

Code 2.3: Die Methode für das Ereignis **TextChanged**

Hinweis:

Der Bezeichner der Schaltfläche in dem Code hängt von dem Namen ab, den Sie an die Schaltfläche vergeben haben.

Mit der Anweisung im `else`-Zweig wird die Schaltfläche wieder deaktiviert, wenn das Feld nach einer Änderung leer ist. Denn der Anwender kann den Inhalt des Eingabefelds ja auch wieder komplett löschen.

Hinweis:

Da der Dialog jetzt nur noch dann über die Schaltfläche **Öffnen** verlassen werden kann, wenn das Eingabefeld nicht leer ist, können Sie die Abfrage auf eine leere Adresse im Quelltext für das Ereignis **Click** der Funktion **Öffnen...** im Menü **Datei** auch wieder entfernen.

2.3 Das Menü Navigation

Nachdem wir jetzt beliebige Internet-Seiten aufrufen können, wollen wir dem Anwender auch einige zusätzliche Standardfunktionen anbieten – zum Beispiel die Vorwärts- und Rückwärtsnavigation durch bisher angezeigte Seiten und das Neuladen einer Seite. Diese Funktionen stellen wir zunächst in einem eigenen Menü **Navigation** zur Verfügung.

Legen Sie bitte ein neues Menü **Navigation** an und erstellen Sie für dieses Menü die drei Einträge **Vorwärts**, **Rückwärts** und **Neu laden**. Vergeben Sie dabei bitte selbst sprechende Namen für die Einträge und wählen Sie geeignete Kürzel für die Tastaturbedienung aus.

Die Anweisungen für die Funktionen sind schnell erstellt, da wir auf vorgefertigte Methoden des WebBrowser-Steuerelements zurückgreifen können. Sie finden die kompletten Methoden für die **Click**-Ereignisse der neuen Menüeinträge zusammengefasst im Code 2.4.

7. **TextChanged** lässt sich mit „Text geändert“ übersetzen.

```
private void NavigationVorwaerts_Click(object sender,
RoutedEventArgs e)
{
    //wenn es einen Eintrag für Vorwärts gibt, dann aufrufen
    if (browser.CanGoForward)
        browser.GoForward();
}

private void NavigationRueckwaerts_Click(object sender,
RoutedEventArgs e)
{
    //wenn es einen Eintrag für Rückwärts gibt, dann aufrufen
    if (browser.CanGoBack)
        browser.GoBack();
}

private void NavigationNeuLaden_Click(object sender,
RoutedEventArgs e)
{
    browser.Refresh();
}
```

Code 2.4: Die Methoden für die neuen Menüeinträge

Hinweis:

Die Namen der Methoden hängen von den Bezeichnungen der Steuerelemente ab.
Sie können bei Ihnen daher auch abweichen.

Besonderheiten gibt es bei den Methoden eigentlich nicht. Je nachdem, welche Aktion ausgewählt wurde, rufen wir die dazugehörige Methode des WebBrowser-Steuerelements auf. Bei der Vorwärts- und Rückwärtssnavigation überprüfen wir zusätzlich über die Eigenschaften **CanGoForward** beziehungsweise **CanGoBack**⁸, ob die Navigation in die gewünschte Richtung überhaupt möglich ist.

Übernehmen Sie jetzt die Anweisungen für die Ereignisse aus dem Code 2.4 und testen Sie die Erweiterungen.

Hinweis:

Die Vorwärts- und Rückwärtssnavigation bezieht sich auf den Navigationsverlauf, den der Browser selbst mitschreibt, und nicht auf die Liste, die wir im Listenfeld erstellen.

In der Liste im Listenfeld werden Seiten, die Sie bereits einmal in der aktuellen Sitzung angezeigt haben, beim erneuten Aufruf noch einmal eingetragen. Eine Änderung der Listenfunktion, die jede Seite nur einmal aufführt, wartet als Einsendeaufgabe auf Sie.

Damit haben wir die wesentlichen Grundfunktionen für unseren Browser fertiggestellt. Im nächsten Kapitel werden wir einige Symbolisten einfügen.

8. Übersetzt bedeuten die beiden Eigenschaften so viel wie „Kann vorwärts gehen“ und „Kann rückwärts gehen“.

Zusammenfassung

Eine Menüleiste erstellen Sie über das Steuerelement **Menu**. Einen Menüeintrag erzeugen Sie über das Steuerelement **MenuItem**. Sie können die Menüs entweder im XAML-Code erstellen oder über den Auflistungs-Editor.

Das Anlegen eines neuen Fensters erfolgt in der WPF ähnlich wie bei einer Windows Forms-Anwendung.

Ein modales Fenster wird in der WPF automatisch geschlossen, wenn Sie die Eigenschaft **DialogResult** verwenden.

Über einen Separator können Sie Trennlinien in ein Menü einfügen.

Die Tabulatorreihenfolge setzen Sie über die Eigenschaft **TabIndex**.

Aufgaben zur Selbstüberprüfung

- 2.1 Für welches Element setzen Sie in einer WPF-Anwendung die Eigenschaft **DialogResult**? Welche Werte kann die Eigenschaft erhalten?

- 2.2 Welche Eigenschaft sollten Sie beim Öffnen eines modalen Fensters in der WPF in jedem Fall setzen, um Probleme zu vermeiden?

- 2.3 Mit welcher Eigenschaft machen Sie eine Schaltfläche zur Standardschaltfläche?

- 2.4 Welches Ereignis wird ausgelöst, wenn sich der Text in einem Eingabefeld ändert?

3 Die Symbolleisten

In diesem Kapitel werden wir unseren Browser um Symbolleisten erweitern.

Wir verwenden dabei zwei getrennte Leisten: In der einen Leiste stellen wir dem Anwender ein Eingabefeld und eine Schaltfläche zum direkten Aufruf einer Internetseite zur Verfügung, in der anderen Leiste bieten wir die Funktionen zum Vorwärts- und Rückwärtsnavigieren sowie zum erneuten Laden einer Seite an.

Ein erstes Gerüst für diese beiden Leisten haben wir ja bereits erstellt. Dabei haben wir auch einen Container vom Typ **ToolBarTray** benutzt, damit die Symbolleisten vom Anwender verschoben werden können. Machen wir uns jetzt an die endgültige Version. Da das Erstellen von Schaltflächen mit Symbolen über den Auflistungs-Editor nicht direkt möglich ist, erstellen wir die Symbolleisten im XAML-Code.

Beginnen wir mit der Symbolleiste für die Navigation.

3.1 Die Symbolleiste Navigation

Erstellen Sie bitte im ersten Schritt einen Ordner für die Symbole im Projekt. Kopieren Sie in diesen Ordner die Grafiken für die Navigationsfunktionen.

Hinweis:

Die entsprechenden Symbole finden Sie im Ordner \cshp16d bei den Beispielen im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Legen Sie dann in der rechten Symbolleiste insgesamt drei Schaltflächen mit Image-Komponenten an. Der komplette Container für die Leisten sollte so aussehen wie im folgenden Code 3.1:

```
<ToolBarTray Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="3">
    <!-- Die erste Leiste -->
    <ToolBar>
        <Button>
            <Image>
                </Image>
            </Button>
    </ToolBar>
    <!-- Die zweite Leiste -->
    <ToolBar>
        <Button>
            <Image>
                </Image>
            </Button>
        <Button>
            <Image>
                </Image>
            </Button>
        <Button>
            <Image>
                </Image>
            </Button>
    </ToolBar>
```

```
</ToolBar>
</ToolBarTray>
```

Code 3.1: Die XAML-Anweisungen für die Symbolleisten

Weisen Sie den Schaltflächen dann von links nach rechts die Eigenschaften aus der Tab. 3.1 zu:

Tab. 3.1: Eigenschaften für die Schaltflächen

Name	ToolTip	Bild
buttonZurueck	Zurück	browser_zurueck.bmp
buttonVorwaerts	Vorwärts	browser_vorwaerts.bmp
buttonNeuLaden	Neu laden	browser_neuladen.bmp

Zur Erinnerung:

Die Bilder für die Schaltflächen setzen Sie über die Eigenschaft **Source** der Image-Komponente.



Passen Sie anschließend die Breite und Höhe der Schaltflächen so an, dass sie jeweils 32 Pixel breit und 32 Pixel hoch sind. Die Symbolleiste sollte nun so aussehen:

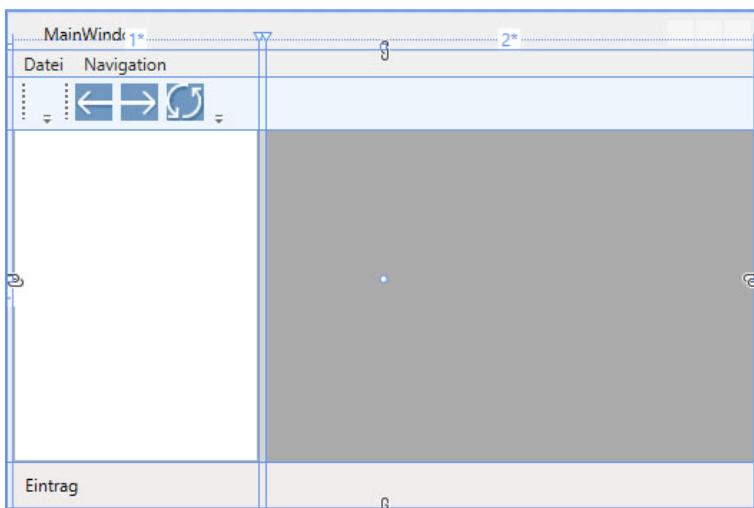


Abb. 3.1: Die Symbolleiste für die Navigation

Weisen Sie den drei Schaltflächen dann die Methoden zu, die auch beim Anklicken der entsprechenden Funktionen im Menü **Navigation** ausgeführt werden.

Speichern Sie anschließend die Änderungen und testen Sie die Symbole. Rufen Sie dazu nach dem Start des Programms eine andere Internetseite auf und navigieren Sie anschließend hin und her. Probieren Sie auch aus, ob sich die beiden Symbolleisten über die Anfasser vorn in der Leiste verschieben lassen.

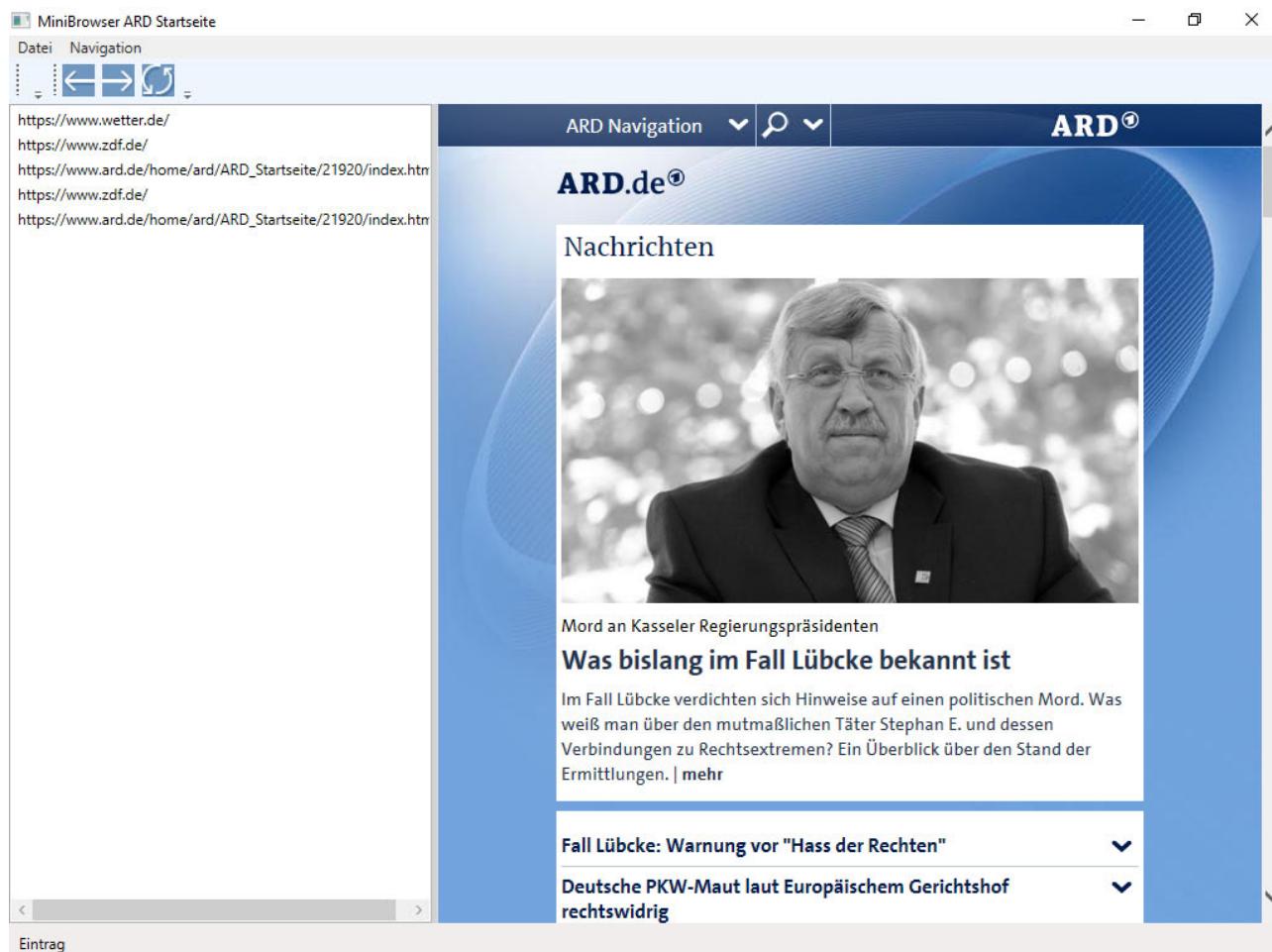


Abb. 3.2: Die Navigationssymbole im praktischen Einsatz

3.2 Die Symbolleiste Adresse

Damit ist die rechte Symbolleiste fertig und wir können uns um die linke Leiste kümmern. Sie soll ein Eingabefeld und ein Symbol enthalten.

Fügen Sie zunächst das Eingabefeld ein. Benutzen Sie dazu ein Steuerelement vom Typ **TextBox**. Verbreitern Sie das Eingabefeld auf ungefähr 350 Pixel und setzen Sie die Eigenschaft **ToolTip** für das Feld auf **Geben Sie eine Adresse ein**.

Hinweis:

Abhängig von der Breite des Fensters führt die Größenänderung beim Eingabefeld unter Umständen dazu, dass die Symbolleiste im Designer umgebrochen wird beziehungsweise dass eine Leiste scheinbar ausgeblendet wird. Diese Darstellung wirkt sich aber nur auf den Entwurf aus. In der eigentlichen Anwendung wird die Symbolleiste in der maximalen Breite des Fensters dargestellt. Wenn Sie die umgebrochene Darstellung stört, verbreitern Sie einfach das Fenster im Designer.

Weisen Sie anschließend der Schaltfläche in der Symbolleiste über die Eigenschaft **Source** der Image-Komponente die Grafik **browser_los.bmp** zu. Setzen Sie den Text für die Schaltfläche und auch die Eigenschaft **ToolTip** auf **Los geht's**.

Auf dieser Schaltfläche wollen wir aber nicht nur eine Grafik, sondern auch noch Text anzeigen lassen. Das geht leider nicht direkt über die Eigenschaft **Content**, da die bereits mit der Image-Komponente belegt ist. Wir setzen daher ein **StackPanel** in die Schaltfläche und ordnen in diesem Container ein Steuerelement vom Typ **TextBlock** und die Image-Komponente an.

Zur Auffrischung:

In einem **StackPanel** werden die Steuerelemente entweder von oben nach unten oder von links nach rechts gestapelt.



Der XAML-Code für die Schaltfläche sieht dann so aus:

```
<Button ToolTip="Los geht's" Height="32">
    <StackPanel Orientation="Horizontal">
        <Image Source="symbole/browser_los.bmp"/>
        <TextBlock Margin="5,5,0,0" Text="Los geht's"/>
    </StackPanel>
</Button>
```

Code 3.2: Der XAML-Code für die Schaltfläche

Die Anweisung

```
<StackPanel Orientation="Horizontal">
```

erzeugt ein **StackPanel** mit horizontaler Ausrichtung. In diesem **StackPanel** befindet sich links die Grafik und rechts der Text. Den Text positionieren wir dabei über die Eigenschaft **Margin** so, dass er mittig mit ein wenig Abstand nach links angezeigt wird.



Bitte beachten Sie, dass Sie für diese Schaltfläche die Breite nicht mehr fest vorgeben dürfen. Sonst verschwindet der Text.

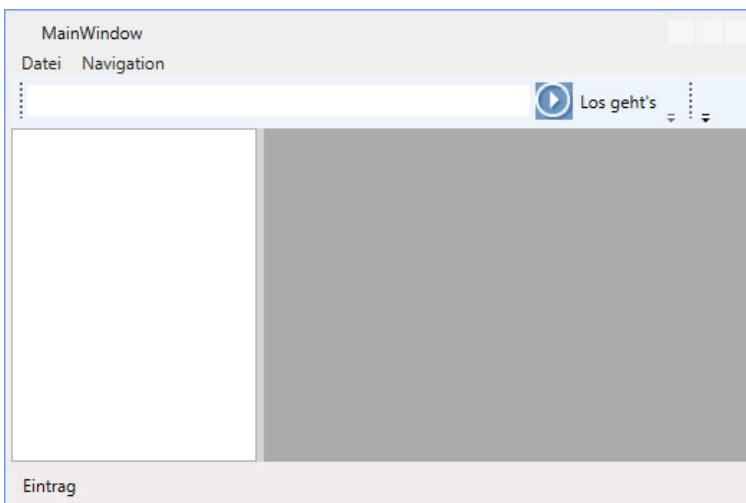


Abb. 3.3: Die Symbolleiste Adresse (die Symbolleiste für die Navigation befindet sich verkleinert ganz rechts im Fenster)

Da wir bisher keine Methode programmiert haben, die direkt eine Seite im Steuerelement **WebBrowser** anzeigt, müssen wir die Anweisungen für das Ereignis **Click** der Schaltfläche in der oberen Symbolleiste neu erstellen. Das ist aber nicht allzu schwierig. Wir überprüfen, ob ein Text im Eingabefeld steht, und übergeben diesen Text dann an die Methode `Navigate()` des WebBrowser-Steuerelements.

Vergeben Sie zunächst an das Eingabefeld und die neue Schaltfläche sprechende Namen. Wir verwenden die Bezeichner `eingabefeld` und `buttonLos`. Die vollständige Methode für das Ereignis **Click** der Schaltfläche sieht so aus:

```
private void ButtonLos_Click(object sender, RoutedEventArgs e)
{
    string adresse;
    //den Text aus dem Eingabefeld beschaffen
    adresse = eingabefeld.Text;
    //wenn Text im Eingabefeld steht, übergeben wir ihn an die
    //Methode Navigate()
    if (eingabefeld.Text != string.Empty)
    {
        //lässt sich die Adresse umwandeln
        try
        {
            browser.Navigate(new Uri(eingabefeld.Text));
        }
        catch (UriFormatException)
        {
            //bitte in einer Zeile eingeben
            MessageBox.Show("Das Format der Adresse " + adresse + " "
                "ist nicht gültig.", "Fehler");
        }
    }
}
```

Code 3.3: Die Methode `ButtonLos_Click()`

Übernehmen Sie bitte die Anweisungen aus dem Code und testen Sie das Programm. Jetzt sollte auch der Aufruf einer Seite über das Eingabefeld in der oberen Symbolleiste funktionieren. Denken Sie beim Test aber bitte daran, dass Sie auch das Protokoll der Seite eingeben müssen.

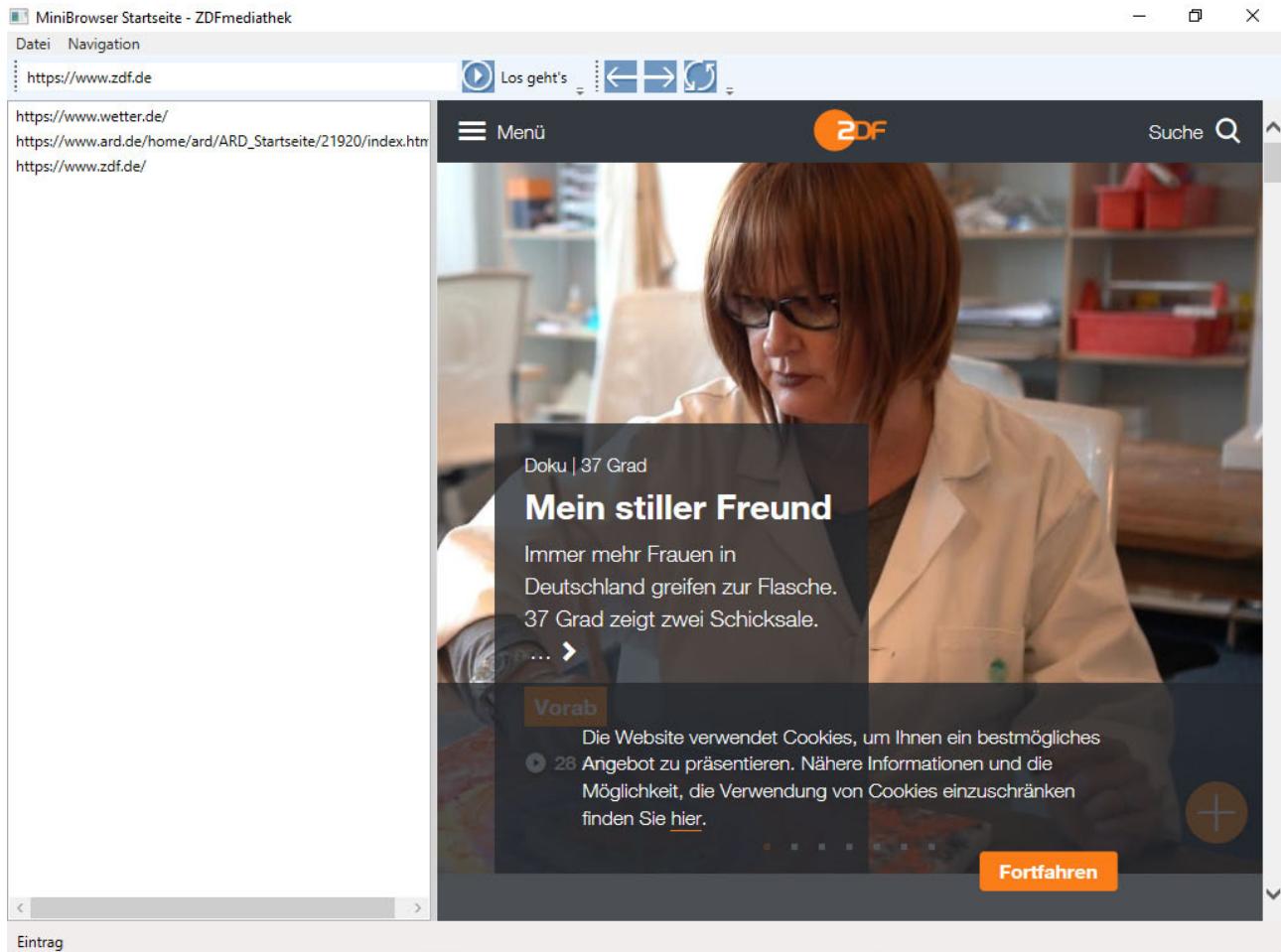


Abb. 3.4: Das Eingabefeld im Einsatz

3.3 Der Feinschliff

Abschließend wollen wir unseren Symbolleisten noch ein wenig Feinschliff verpassen.

Im ersten Schritt lassen wir die Symbole aus der Leiste **Navigation** auch bei den entsprechenden Einträgen im Menü **Navigation** anzeigen. Dazu weisen Sie der Eigenschaft **MenuItem.Icon** eine Image-Komponente mit dem entsprechenden Bild zu. Das geht am einfachsten direkt im XAML-Code. Für den Eintrag **Vorwärts** im Menü **Navigation** könnte dieser Code so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<MenuItem x:Name="navigationVorwaerts" Header="_Vorwärts"
Click="NavigationVorwaerts_Click">
<MenuItem.Icon>
    <Image Source="symbole/browser_vorwaerts.bmp" />
</MenuItem.Icon>
</MenuItem>
```

Code 3.4: Das Symbol für den Menüeintrag Vorwärts

Nehmen Sie die entsprechenden Änderungen jetzt bitte auch für die beiden anderen Einträge im Menü **Navigation** vor.

Damit der Aufruf einer Seite etwas einfacher wird, überprüfen wir auch noch, ob der Anwender im Eingabefeld die Eingabetaste gedrückt hat, und rufen dann die entsprechende Seite auf. Dazu erstellen wir eine Methode für das Ereignis **KeyDown** des Eingabefelds. Dieses Ereignis tritt jedes Mal ein, wenn das Eingabefeld den Fokus hat und eine Taste gedrückt wird.

Hinweis:

Es gibt zwei unterschiedliche Tastaturereignisse in der WPF: **KeyDown** und **KeyUp**. Das Ereignis **KeyDown** tritt ein, sobald eine Taste gedrückt wird. Das Ereignis **KeyUp** tritt ein, wenn eine Taste wieder losgelassen wird.

In welchem Ereignis Sie eine normale Tastatureingabe auswerten, ist eigentlich relativ beliebig. Denn das Ereignis **KeyUp** muss ja nach dem Drücken einer Taste irgendwann eintreten – nämlich spätestens dann, wenn der Anwender die Taste wieder loslässt. Wichtig sind die Unterschiede vor allem dann, wenn Sie gezielt auf das Festhalten beziehungsweise Loslassen einer Taste reagieren wollen.

Welche Taste gedrückt beziehungsweise losgelassen wurde, wird bei den Ereignissen **KeyUp** und **KeyDown** über die Eigenschaft **KeyCode** des Parameters **e** an die jeweilige Methode übergeben, und zwar als Wert der Aufzählung **Key**. Für die Eingabetaste steht beispielsweise der Wert **Key.Enter**.

Die Anweisungen für das Ereignis **KeyDown** des Eingabefelds sehen dann so aus:

```
private void Eingabefeld_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
        ButtonLos_Click(sender, e);
}
```

Code 3.5: Die Methode **Eingabefeld_KeyDown()**

Das Anzeigen der Seite erledigt hier unsere Methode **ButtonLos_Click()**. Sie überprüft ja auch, ob überhaupt eine Eingabe in dem Feld gemacht wurde.

Übernehmen Sie jetzt die Anweisungen aus dem vorigen Code für das Ereignis **KeyDown** des Eingabefelds.

Damit das Eingabefeld auch bei der Navigation über einen Link in einer Seite oder beim Einsatz unserer Vorwärts- und Rückwärtssnavigation die URL der neuen Seite anzeigt, lassen wir den Text in dem Feld beim Ereignis **Navigated** des WebBrowser-Steuerelements aktualisieren. Dazu wandeln wir die Eigenschaft **Source** des Steuerelements **browser** mit der Methode **ToString()** in eine Zeichenkette um und weisen diese Zeichenkette der Eigenschaft **Text** des Eingabefelds zu. Die entsprechende Anweisung sieht so aus:

```
eingabefeld.Text = browser.Source.ToString();
```

Ergänzen Sie diese Anweisung bitte am Ende der Methode **Browser_Navigated()**.

Im nächsten Schritt des Feinschliffs aktivieren wir die Schaltflächen für die Vorwärts- und Rückwärtsnavigation nur dann, wenn die entsprechenden Funktionen überhaupt zur Verfügung stehen. Dazu deaktivieren Sie die Eigenschaft **IsEnabled** der beiden Schaltflächen zunächst, überprüfen dann im Ereignis **Navigated**, ob eine Vorwärtsbeziehungsweise Rückwärtsnavigation überhaupt möglich ist, und aktivieren die Symbole gegebenenfalls wieder. Damit das Aktivieren beziehungsweise Deaktivieren auch bei späteren Navigationen korrekt erfolgt, müssen Sie die Schaltflächen zusätzlich wieder deaktivieren, wenn die erste beziehungsweise letzte Seite erreicht wurde.

Da sich Menüeinträge mit einem sehr ähnlichen Verfahren aktivieren beziehungsweise deaktivieren lassen, erledigen wir das ebenfalls in der Methode `Browser_Navigated()`. Deaktivieren Sie jetzt bitte die Eigenschaft **IsEnabled** für die Schaltflächen und die Menüeinträge der Vorwärts- beziehungsweise Rückwärtsnavigation. Sie finden diese Eigenschaft jeweils in der Gruppe **Allgemein**. Setzen Sie dann auch noch die Eigenschaft **Opacity** in der Gruppe **Darstellung** für die beiden Schaltflächen auf 50, damit sie ein wenig abgeblendet werden.

Erweitern Sie anschließend die Methode `browser_Navigated()` um die folgenden Anweisungen:

```
if (browser.CanGoBack)
{
    navigationRueckwaerts.IsEnabled = true;
    buttonZurueck.IsEnabled = true;
    buttonZurueck.Opacity = 1;
}
else
{
    navigationRueckwaerts.IsEnabled = false;
    buttonZurueck.IsEnabled = false;
    buttonZurueck.Opacity = 0.5;
}
if (browser.CanGoForward)
{
    navigationVorwaerts.IsEnabled = true;
    buttonVorwaerts.IsEnabled = true;
    buttonVorwaerts.Opacity = 1;
}
else
{
    navigationVorwaerts.IsEnabled = false;
    buttonVorwaerts.IsEnabled = false;
    buttonVorwaerts.Opacity = 0.5;
}
```

Code 3.6: Das Aktivieren und Deaktivieren der Symbole und Menüeinträge

Hinweis:

Die Bezeichner der Menüeinträge und der Symbole in dem vorigen Code hängen von den Namen ab, die Sie in Ihrem Projekt verwenden.

Da die beiden Funktionen jetzt nur noch dann gestartet werden können, wenn die Navigation möglich ist, können Sie die entsprechenden Abfragen in den eigentlichen Methoden für die Vorwärts- beziehungsweise Rückwärtssnavigation auch wieder löschen.

Im letzten Schritt sorgen wir – als Tüpfelchen auf dem I – dafür, dass der Tooltipp für die Schaltfläche **Los geht's** auch noch die URL der Seite enthält, die beim Anklicken angezeigt wird. Dazu ändern wir die Eigenschaft **Tooltip** der Schaltfläche im Ereignis **MouseEnter**.



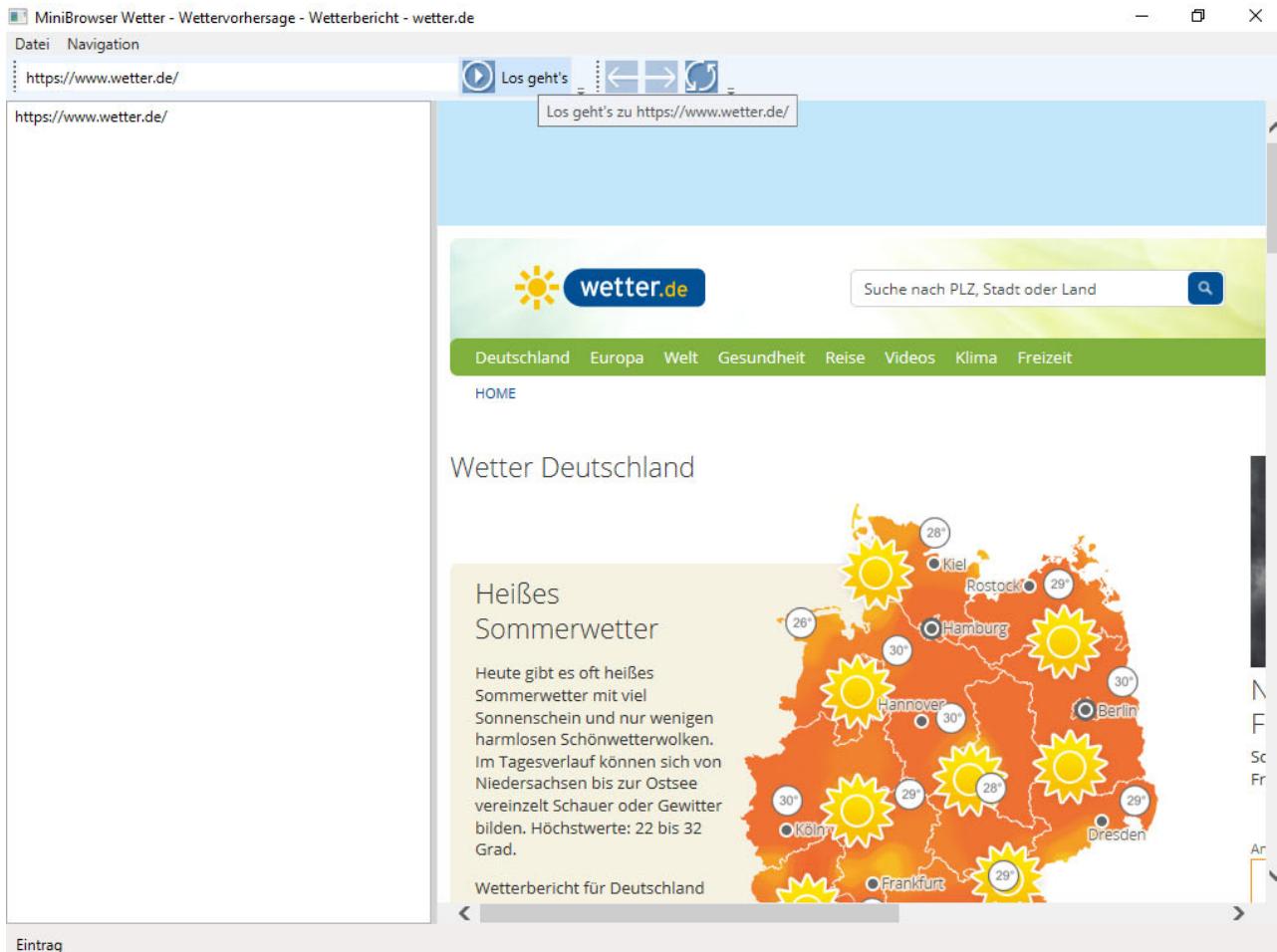
Zur Erinnerung:

Das Ereignis **MouseEnter** tritt ein, wenn die Maus auf das Steuerelement bewegt wird.

Lassen Sie die Methode für das Ereignis **MouseEnter** für die Schaltfläche **Los geht's** erstellen und übernehmen Sie die folgende Anweisung:

```
buttonLos.ToolTip = "Los geht's zu " + eingabefeld.Text;
```

Speichern Sie dann die Änderungen und testen Sie die Erweiterungen.



Damit sind auch die Symbolleisten zunächst einmal fertiggestellt. Im nächsten Kapitel werden wir noch ein paar Extras in unseren Browser einbauen.

Zusammenfassung

Um ein Symbol für einen Menüeintrag anzuzeigen, verwenden Sie die Eigenschaft **MenuItem.Icon**.

Bei den Tastaturereignissen wird zwischen **KeyDown** und **KeyUp** unterschieden.

Über die Eigenschaft **IsEnabled** können Sie ein Steuerelement aktivieren und deaktivieren.

Aufgaben zur Selbstüberprüfung

- 3.1 Beschreiben Sie kurz, wie Sie auf einer Schaltfläche gleichzeitig einen Text und eine Grafik anzeigen lassen können.

- 3.2 Sie wollen in einem Ereignis **KeyDown** prüfen, ob die Eingabetaste gedrückt wurde. Formulieren Sie eine entsprechende Abfrage.

4 Ein paar Extras

In diesem Kapitel werden wir unseren Browser noch um einige Extras erweitern. Wir stellen dem Anwender Funktionen zum Ein- und Ausblenden von Bereichen sowie Symbolleisten zur Verfügung und sorgen unter anderem dafür, dass über die Liste im linken Bereich navigiert werden kann. Außerdem erstellen wir ein Kontext-Menü für die Liste, über das Einträge in der Liste gelöscht werden können.

4.1 Das Menü Ansicht

Beginnen wir mit den Funktionen zum Ein- und Ausblenden.

Das Ein- und Ausblenden für die Symbolleisten ist schnell umgesetzt. Wir setzen hier die Eigenschaft **Visibility** entweder auf `Visibility.Collapsed` oder auf `Visibility.Visible`. Der Wert `Visibility.Collapsed` sorgt auch automatisch dafür, dass das Steuerelement beim Layout nicht mehr berücksichtigt wird.



Wenn Sie die Sichtbarkeit auf `Visibility.Hidden` setzen, wird das Steuerelement nur ausgeblendet. Der Platz im Layout bleibt aber weiter reserviert.

Für das Ein- und Ausblenden der Navigationsliste am linken Rand müssen wir etwas mehr Aufwand betreiben. Wenn wir nur die Liste ausblenden, wird die Spalte nach wie vor angezeigt. Das liegt unter anderem daran, dass wir eine Mindestbreite vorgegeben haben und die Breite der Spalte proportional gesetzt wird. Wir könnten nun die Mindestbreite weglassen und die Breite auf `Auto` setzen. Das führt dann aber dazu, dass sich die Breite der Liste an den längsten Eintrag anpasst. Und das ist ein recht ungewöhnliches Verhalten.

Wir gehen daher einen anderen Weg: Beim Ausblenden der Liste merken wir uns die aktuelle Breite und Mindestbreite der Spalte und setzen die Breite der Spalte auf `0`. Außerdem lassen wir auch die Spalte für den Splitter verschwinden, indem wir die Breite auf `0` setzen. Beim Einblenden stellen wir die alten Breiten dann wieder her.

Damit Sie auf die verschiedenen Steuerelemente und auch die Spalten im Grid zugreifen können, vergeben Sie jetzt bitte Namen. Wir benutzen die Bezeichner `toolbarNavigation` und `toolbarAdresse` für die Symbolleisten sowie `spalteLinks` und `spalteSplitter` für die beiden Spalten.

Erstellen Sie jetzt in der Menüleiste einen neuen Eintrag mit dem Titel `_Ansicht`. Legen Sie in dem Menü einen Eintrag `_Navigationsliste` und einen Eintrag `_Symbolleisten` an. Erstellen Sie dann ein Untermenü für den Eintrag `Symbolleisten` mit den beiden Einträgen `Ad_resse` und `Na_vigation`. Dazu legen Sie die beiden Einträge entweder im XAML-Code innerhalb des Eintrags `Symbolleisten` an oder Sie verwenden den Sammlungs-Editor. Vergeben Sie an die neuen Einträge bitte wieder Namen.

Damit der Anwender auch im Menü sofort sehen kann, ob die entsprechenden Elemente gerade angezeigt werden oder nicht, versehen wir die Einträge noch mit einem Häkchen. Dieses Häkchen soll allerdings nur dann erscheinen, wenn das Element aktuell angezeigt wird.

Damit ein Eintrag markiert wird, aktivieren Sie die Eigenschaft **IsChecked**. Sie finden diesen Eintrag in der Gruppe **Darstellung** des Eigenschaftenfensters. Damit die Markierung beim Anklicken automatisch umgeschaltet wird, markieren Sie außerdem noch die Eigenschaft **IsCheckable** in der Gruppe **Allgemein**. Sie können die Eigenschaften aber auch direkt im XAML-Code auf `True` setzen.

Führen Sie diese Änderungen jetzt für den Eintrag **Navigationsliste** und die beiden Einträge im Untermenü **Symbolleisten** durch. Das Menü sollte im Designer nun ungefähr so aussehen:

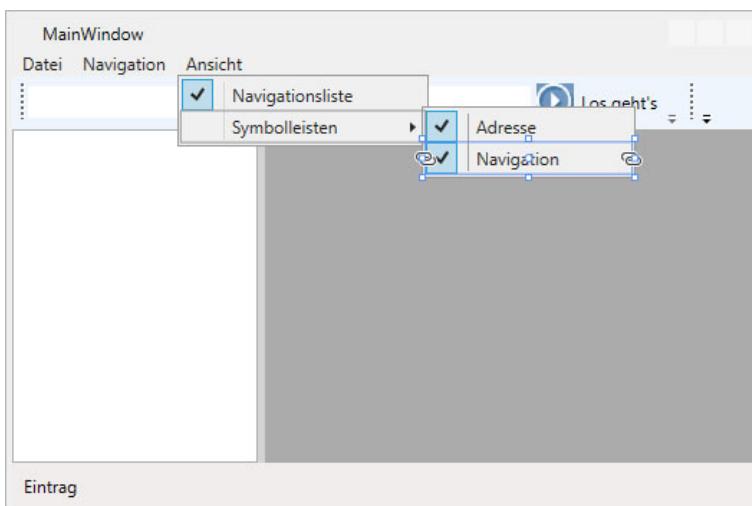


Abb. 4.1: Das Menü mit den Markierungen

Beim Anklicken der Menüeinträge prüfen wir jetzt, ob der Eintrag markiert ist oder nicht, und blenden dann die Elemente ein oder aus. Damit wir uns die Breite der Spalten merken können, legen Sie bitte im ersten Schritt zwei Felder vom Typ `GridLength` an. Über diesen Typ werden die Breite und der Layouttyp eines Grid-Elements festgelegt. Für das Speichern der Breite des Listenfelds verwenden wir ein Feld vom Typ `double`.

Die Anweisungen für die Methoden zum Ein- und Ausblenden finden Sie zusammengefasst im folgenden Code. Bitte beachten Sie dabei, dass die Anweisungen für die Methoden erst nach dem Verändern der Markierung ausgeführt werden. Wenn Sie also auf einen markierten Eintrag klicken, wird erst die Markierung entfernt und dann die Methode für das Ereignis `Click` ausgeführt.

```
private void AnsichtNavigationsliste_Click(object sender,
RoutedEventArgs e)
{
    //ist der Eintrag markiert?
    if (ansichtNavigationsliste.IsChecked == true)
    {
        //die beiden Spalten auf die ursprüngliche Breite setzen
        spalteLinks.Width = spalteLinksBreite;
        spalteSplitter.Width = spalteSplitterBreite;
        spalteLinks.MinWidth = spalteLinksMindestbreite;
        //die Liste wieder einblenden
        liste.Visibility = Visibility.Visible;
    }
}
```

```

        else
    {
        //die Spalten ausblenden
        //die alten Breiten sichern
        spalteLinksBreite = spalteLinks.Width;
        spalteSplitterBreite = spalteSplitter.Width;
        spalteLinksMindestbreite = spalteLinks.MinWidth;
        //die Liste ausblenden
        liste.Visibility = Visibility.Collapsed;
        //und die Breite der Spalten auf 0 setzen
        spalteLinks.MinWidth = 0;
        spalteLinks.Width = new GridLength(0);
        spalteSplitter.Width = new GridLength(0);
    }
}

private void AnsichtSymbolleistenAdresse_Click(object sender, RoutedEventArgs e)
{
    if (ansichtSymbolleistenAdresse.IsChecked == true)
        toolbarAdresse.Visibility = Visibility.Visible;
    else
        toolbarAdresse.Visibility = Visibility.Collapsed;
}

private void AnsichtSymbolleistenNavigation_Click(object sender, RoutedEventArgs e)
{
    if (ansichtSymbolleistenNavigation.IsChecked == true)
        toolbarNavigation.Visibility = Visibility.Visible;
    else
        toolbarNavigation.Visibility = Visibility.Collapsed;
}

```

Code 4.1: Die Methoden zum Ein- und Ausblenden der Symbolleisten und der Navigationsliste

Neu sind vor allem die Anweisungen:

```

spalteLinks.Width = new GridLength(0);
spalteSplitter.Width = new GridLength(0);

```

Hier setzen wir die Breite der Spalten auf 0. Das geht aber nicht direkt über einen numerischen Wert, sondern nur über einen Wert vom Typ `GridLength`. Denn neben einer Breite in Pixeln können Sie hier auch eine automatische Breite oder eine proportionale Breite setzen. Dazu müssen Sie dann als zweites Argument einen Wert der Aufzählung `GridUnitType`⁹ angeben. Bei einer absoluten Größenangabe können Sie dieses Argument auch weglassen.

Hinweis:

Sie können auf das Zwischenspeichern auch verzichten und die Spalten einfach wieder in einer Standardbreite beziehungsweise der festen Breite anzeigen lassen. Das kann aber zu Verwirrung beim Anwender führen, wenn die linke Spalte nach dem Einblenden schmäler oder breiter ist als beim Ausblenden.

9. `UnitType` lässt sich mit „Typ der Einheit“ übersetzen.

Durch das Zwischenspeichern der Spaltenbreite des Splitters sind Änderungen im Layout einfacher. Denn sonst müssten Sie ja an zwei Stellen Änderungen vornehmen, wenn Sie die Breite später einmal auf einen anderen Wert setzen wollen.

Auch bei diesem Code können die Namen der Methoden und auch der Steuerelemente abweichen. Sie hängen von den Bezeichnern ab, die Sie benutzt haben.

Neben dem Anklicken der Menüeinträge können Sie auch über die Ereignisse **Checked** und **Unchecked** auf Änderungen der Markierung reagieren. Sie treten ein, wenn eine Markierung gesetzt beziehungsweise entfernt wird. Allerdings wird das Ereignis **Checked** auch dann ausgelöst, wenn ein Eintrag beim Start des Programms markiert ist. Und das kann bei unserem Browser zu Problemen führen, da die Elemente unter Umständen noch gar nicht erzeugt sind und wir auch noch keine Informationen zu den verschiedenen Breiten gespeichert haben.

Übernehmen Sie die Erweiterungen aus dem vorigen Code und testen Sie das Programm. Die verschiedenen Bereiche im Fenster sollten sich jetzt beliebig ein- und ausblenden lassen.

4.2 Sonderfunktionen für die Liste

Im nächsten Schritt sorgen wir nun dafür, dass der Anwender durch einen Mausklick auf einen Eintrag in der Navigationsliste die Seite wieder anzeigen kann. Das lässt sich recht einfach über die Methode `Navigate()` für das WebBrowser-Steuerelement bewerkstelligen. Als Argument übergeben wir dabei die Zeichenkette des aktuell ausgewählten Eintrags im Listenfeld.

Die Anweisung rufen wir im Ereignis **SelectionChanged** der Liste auf. Die Methode für das Ereignis besteht nur aus einer Anweisung und sieht so aus:

```
private void Liste_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    //die Seite aufrufen
    browser.Navigate(new Uri(liste.SelectedItem.ToString()));
}
```

Code 4.2: Die Methode `Liste_SelectionChanged()`

Auf Prüfungen können wir hier verzichten, da ja keine ungültigen Adressen in der Liste stehen können.

Zusätzlich zur Navigation wollen wir jetzt noch eine Funktion zum Löschen der Liste erstellen. Diese Funktion soll über ein Kontext-Menü aufgerufen werden können.

Markieren Sie bitte das Listenfeld-Steuerelement im Designer. Klicken Sie dann auf die Schaltfläche **Neu** hinter der Eigenschaft **ContextMenu** in der Gruppe **Sonstiges**.

Legen Sie danach im XAML-Code den folgenden Menüeintrag für das Kontext-Menü an:

```
<MenuItem x:Name="kontextMenuLoeschen" Header="Löschen"
Click="KontextMenuLoeschen_Click"/>
```

Die Methode zum Löschen besteht ebenfalls nur aus einer Anweisung. Sie sieht so aus:

```
private void KontextMenuLoeschen_Click(object sender,
RoutedEventArgs e)
{
    //die Liste löschen
    liste.Items.Clear();
}
```

Code 4.3: Das Löschen der Liste

Hier löschen wir über die Methode `Items.Clear()` alle Einträge in der Liste.

Wenn allerdings ein Eintrag in der Liste markiert ist, führt das Löschen dazu, dass die Methode `Liste_SelectionChanged()` ausgeführt wird. Denn die Markierung verändert sich ja durch das Löschen. Da dann aber kein Eintrag mehr in der Liste vorhanden ist, wird beim Aufruf der Methode `Navigate()` eine Ausnahme ausgelöst.

Wir sorgen daher dafür, dass die Methode `Navigate()` nur dann ausgeführt wird, wenn auch tatsächlich ein Eintrag markiert ist. Dazu können wir die Eigenschaft `SelectedItem` der Liste verwenden. Sie liefert `null`, wenn kein Eintrag markiert ist.

Die geänderte Methode `Liste_SelectionChanged()` sieht dann so aus:

```
private void Liste_SelectionChanged(object sender,
System.Windows.Controls.SelectionChangedEventArgs e)
{
    //die Seite aufrufen, aber nur dann, wenn ein Eintrag
    //markiert ist
    if (liste.SelectedItem != null)
        browser.Navigate(new Uri(liste.SelectedItem.ToString()));
}
```

Code 4.4: Die geänderte Methode `Liste_SelectionChanged()`

4.3 Die Statusleiste

Nachdem jetzt das Kontext-Menü für die Liste funktioniert, kümmern wir uns um die Statusleiste. In dieser Leiste soll der Ladevorgang für eine Seite als Text und über eine Fortschrittsanzeige dargestellt werden. Beide Anzeigen sollen nur dann erscheinen, wenn gerade eine Seite geladen wird. Andernfalls soll der Text „Seite geladen“ in der Statusleiste stehen.

Für die Fortschrittsanzeige benutzen wir ein Steuerelement vom Typ **ProgressBar**¹⁰. Die Textanzeigen führen wir in dem Label durch, das wir bereits im Rohbau eingefügt haben. Damit die beiden Elemente korrekt positioniert werden, fügen wir sie als `StatusBarItem`s ein. Die Fortschrittsanzeige soll links angezeigt werden und das Label rechts.

10. `ProgressBar` bedeutet so viel wie „Fortschrittsbalken“.

Die Breite der Fortschrittsanzeige setzen wir auf 300 und die Höhe auf 20. Dadurch ist die Anzeige besser zu sehen. Damit in der Anzeige automatisch eine Animation erfolgt, markieren Sie bitte noch die Eigenschaft **IsIndeterminate**¹¹ im Bereich **Allgemein**. Danach sollte ein grünes Rechteck kontinuierlich von links nach rechts durchlaufen.

Den Text im Label lassen wir erst einmal leer.

Damit wir gleich aus dem Code auf die Steuerelemente zugreifen können, vergeben Sie bitte noch Namen. Wir benutzen die Bezeichner `statusbarProgress` und `statusbarLabel`.

Der XAML-Code für die Statusleiste sieht dann so aus:

```
<StatusBar Grid.Column="0" Grid.Row="3" Grid.ColumnSpan="3">
    <StatusBarItem>
        <!-- bitte in einer Zeile eingeben -->
        <ProgressBar x:Name="statusbarProgress" Height="20"
            Width="300" IsIndeterminate="True"/>
    </StatusBarItem>
    <StatusBarItem>
        <Label x:Name="statusbarLabel" Content="Eintrag"/>
    </StatusBarItem>
</StatusBar>
```

Code 4.5: Die Statusleiste

Die Statusleiste sollte dann ungefähr so aussehen:

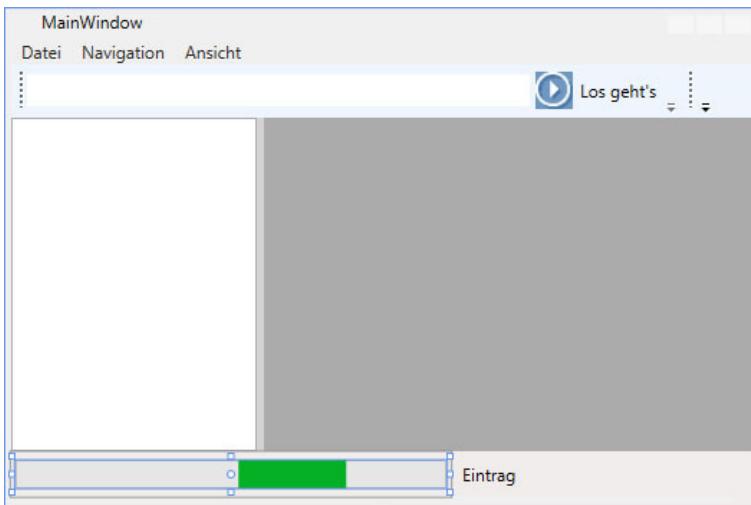


Abb. 4.2: Der Rohbau der Statusleiste

Blenden Sie die Fortschrittsanzeige jetzt bitte aus, indem Sie die Eigenschaft **Visibility** in der Gruppe **Darstellung** auf `Collapsed` setzen. Löschen Sie außerdem den Text im Label.

11. *Indeterminate* bedeutet so viel wie „unbestimmt“.

Beim Laden einer Seite blenden wir die Fortschrittsanzeige dann wieder ein und setzen einen Text in das Label. Dazu verwenden wir das Ereignis **Navigating** des WebBrowser-Steuerelements. Es tritt ein, **bevor** die eigentliche Navigation stattfindet. Legen Sie jetzt die Methode `browser_Navigating()` an und übernehmen Sie die folgenden Anweisungen:

```
//sind die Steuerelemente schon verfügbar?  
//dann die Anzeigen setzen  
if (statusbarProgress != null && statusbarLabel != null)  
{  
    statusbarProgress.Visibility = Visibility.Visible;  
    statusbarLabel.Content = "Seite wird geladen...";  
}
```

Code 4.6: Das Setzen der Anzeigen im Ereignis **Navigating**

Da das Ereignis **Navigating** in unserem Programm eintreten kann, bevor die Steuerelemente für die Statusleiste bereits verfügbar sind, prüfen wir vor dem Zugriff, ob die Anzeige möglich ist. Das erledigt die Abfrage:

```
if (statusbarProgress != null && statusbarLabel != null)
```



Zur Erinnerung:

Wenn Sie auf solche Abfragen verzichten, werden beim Zugriff auf nicht vorhandene Steuerelemente Ausnahmen ausgelöst. Sie werden aber beim normalen Ausführen des Programms nicht angezeigt, sondern erscheinen nur, wenn Sie das Programm im Debug-Modus starten.

Die Abfrage kann dann aber auch dazu führen, dass für die Seite, die wir direkt beim Starten laden, keine Anzeigen erfolgen. Wenn Sie das stört, müssen Sie das Programm so umbauen, dass Sie die Startseite nicht mehr im Eigenschaftenfenster zuweisen, sondern zum Beispiel im Ereignis **Loaded** des Fensters. Es wird ausgelöst, wenn das Fenster komplett geladen ist. Sie können die Seite aber auch im Konstruktor nach der Anweisung

```
InitializeComponent();  
laden.
```

Im Ereignis **Navigated** blenden wir die Fortschrittsanzeige dann wieder aus und setzen den Text im Label auf „Seite geladen“. Die entsprechenden Anweisungen sehen so aus:

```
statusbarProgress.Visibility = Visibility.Collapsed;  
statusbarLabel.Content = "Seite geladen";
```

Ergänzen Sie diese Anweisungen jetzt am Ende der Methode `Browser_Navigated()`. Speichern Sie alle Änderungen und testen Sie das Programm.

Damit ist auch die Statusleiste fertiggestellt.

4.4 Ein paar Kleinigkeiten

Abschließend wollen wir mit ein paar Kleinigkeiten noch für zwei weitere i-Tüpfelchen sorgen. Im ersten Schritt lassen wir beim Aufruf einer Funktion **Info** ... in einem Menü **Hilfe** ein Meldungsfenster mit einem Info-Symbol anzeigen. Das ist mit einer Anweisung erledigt. Sie sieht bei der WPF so aus:

```
//bitte in einer Zeile eingeben  
MessageBox.Show("Programmiert von <Mein Name>", "Info",  
MessageBoxButton.OK, MessageBoxIcon.Information);
```

Code 4.7: Das Meldungsfenster

Den Platzhalter <Mein Name> können Sie dabei natürlich durch Ihren eigenen Namen ersetzen.

Bitte beachten Sie:

Die Schaltflächen geben Sie bei der WPF über die Aufzählung `MessageBoxButton` an und nicht wie bei einer Windows Forms-Anwendung über `MessageBoxButtons`. Die Aufzählung für das Symbol heißt `MessageBoxImage` und nicht `MessageBoxIcon` wie bei Windows Forms.



Wenn Sie Lust haben, versehen Sie Ihr Programm außerdem noch mit einem eigenen Symbol. Dazu benötigen Sie eine Grafik in der Größe 16 × 16 Pixel. Diese Grafik weisen Sie der Eigenschaft **Icon** in der Gruppe **Allgemeine** des Fensters zu.

Hinweis:

Ein Muster für solch ein Symbol finden Sie im Ordner `\cshp16d_icons` bei den Beispielen im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Damit wollen wir die Arbeit an unserem Browser an dieser Stelle erst einmal beenden. Einige zusätzliche Funktionen warten noch – wie gewohnt – als Einsendeaufgaben auf Sie. Im nächsten Kapitel werden wir Ihnen aber noch eine alternative Möglichkeit zum Aufruf von Funktionen zeigen – den Einsatz von Commands.

Zusammenfassung

Wenn Sie eine Mindestgröße in einem Grid angeben, wird die Spalte beziehungsweise Zeile auch dann weiter angezeigt, wenn Sie alle Steuerelemente ausblenden.

Über die Eigenschaft **IsChecked** können Sie einen Menüeintrag markieren.

Die Breite einer Spalte in einem Grid wird über den Typ `GridLength` verarbeitet. Sie können daher der Eigenschaft **Width** einer Spalte auch nicht direkt einen numerischen Wert zuweisen.

Mit den Ereignissen **Checked** und **Unchecked** können Sie auf das Setzen und Entfernen einer Markierung reagieren.

Über ein Steuerelement **ProgressBar** können Sie eine Fortschrittsanzeige erstellen.

Über die Eigenschaft **Icon** können Sie ein Symbol für eine Anwendung setzen.

Aufgaben zur Selbstüberprüfung

- 4.1 Was unterscheidet die Sichtbarkeiten `Visibility.Collapsed` und `Visibility.Hidden`?

- 4.2 Welche Eigenschaft müssen Sie setzen, damit ein markierter Menüeintrag beim Anklicken automatisch umgeschaltet wird?

- 4.3 Welche Eigenschaft bestimmt der Typ `GridLength` außer der Spaltenbreite noch?

- 4.4 In einem Menü ist ein Eintrag markiert. Was geschieht zuerst, wenn Sie auf diesen Eintrag klicken: Wird erst die Markierung entfernt? Oder wird erst die Methode für das Ereignis `Click` ausgeführt?

- 4.5 Was steuert die Eigenschaft **IsIndeterminate** für ein Steuerelement vom Typ **ProgressBar**?

- 4.6 Wann tritt das Ereignis **Navigating** für ein WebBrowser-Steuerelement ein?

5 Der Einsatz von Commands

In unserem Browser haben wir bisher selbst geprüft, ob eine Navigation möglich ist, und die entsprechenden Funktionen ein- und ausgeschaltet. Das funktioniert zwar, ist allerdings recht mühselig. Außerdem gibt es eine Alternative: den Einsatz von Commands. Damit werden wir uns in diesem Kapitel beschäftigen.

Commands kennen Sie ja bereits von unserer kleinen WPF-Textverarbeitung. Hier haben wir fertige Befehle eingesetzt, die zum Beispiel Formatierungen durchführen oder Funktionen für die Zwischenablage zur Verfügung stellen. Und ähnliche Commands stellt die WPF auch für Navigationsfunktionen zur Verfügung – nämlich in der Klasse `NavigationCommands`. Hier finden Sie unter anderem auch Commands für die Vorwärts- und Rückwärtssnavigation – nämlich `NavigationCommands.BrowseForward` und `NavigationCommands.BrowseBack`.

Diese Commands können Sie auch ohne Probleme unseren Navigationsfunktionen zuordnen. Allerdings führen sie nur dazu, dass zum Beispiel für einen Menüeintrag auch ein Shortcut angezeigt wird. Aufrufen lassen sich die Funktionen nicht mehr.



Über einen Shortcut für eine Menüfunktion kann die entsprechende Funktion direkt über die Tastatur gestartet werden – auch dann, wenn das Menü nicht geöffnet ist.

Das Problem lässt sich einfach erklären und recht einfach lösen: Die Commands sind nicht implementiert. Wir müssen selbst festlegen, wann ein Command ausgeführt werden kann und was beim Ausführen des Commands geschehen soll. Dazu müssen Sie die Eigenschaften `CanExecute` und `Executed` festlegen. Bei `CanExecute` geben Sie eine Methode an, die die Eigenschaft `e.CanExecute` auf `true` setzt, wenn das Command ausgeführt werden kann. Bei `Executed` geben Sie die Methode an, die beim Ausführen des Commands aufgerufen wird.

Das Command selbst können Sie zum Beispiel bei der Eigenschaft `CommandBindings` des Fensters festlegen. Sie können aber auch die entsprechende Eigenschaft eines anderen Steuerelements benutzen und die Gültigkeit so begrenzen.

Die XAML-Anweisungen für die beiden Commands `NavigationCommands.BrowseBack` und `NavigationCommands.BrowseForward` würden dann auf Ebene des Fensters so aussehen:

```
<Window.CommandBindings>
    <!-- bitte jeweils in einer Zeile eingeben -->
    <CommandBinding Command="NavigationCommands.BrowseForward"
        CanExecute="BrowseForward_CanExecute"
        Executed="BrowseForward_Executed" />
    <CommandBinding Command="NavigationCommands.BrowseBack"
        CanExecute="BrowseBack_CanExecute"
        Executed="BrowseBack_Executed" />
</Window.CommandBindings>
```

Code 5.1: Die XAML-Anweisungen für die Commands

Die Methoden für die beiden Commands finden Sie im folgenden Code 5.2:

```
private void BrowseBack_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    //gibt es das Steuerelement und können wir zurücknavigieren?
    if (browser != null && browser.CanGoBack == true)
        e.CanExecute = true;
}

private void BrowseBack_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    //zurücknavigieren
    browser.GoBack();
}

private void BrowseForward_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    //gibt es das Steuerelement und können wir vorwärts
    //navigieren?
    if (browser != null && browser.CanGoForward == true)
        e.CanExecute = true;
}

private void BrowseForward_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    browser.GoForward();
}
```

Code 5.2: Die Methoden für die Commands

In den beiden `CanExecute()`-Methoden prüfen wir, ob es das Steuerelement für den Browser gibt und ob die Navigation in die gewünschte Richtung möglich ist. Wenn das der Fall ist, setzen wir die Eigenschaft `e.CanExecute` auf `true`. Damit wird markiert, dass das Command ausgeführt werden kann. Was genau beim Ausführen geschieht, legen wir in den `Executed()`-Methoden fest. Sie bestehen lediglich aus dem entsprechenden Befehl für die Navigation.

Die Zuordnung der Commands erfolgt dann wie gewohnt über die Eigenschaft **Command** des Steuerelements. Probieren Sie das jetzt selbst im Browser aus. Erstellen Sie dazu am besten eine Kopie Ihres Projekts und nehmen Sie dort dann die Änderungen vor. Löschen Sie außerdem alle Anweisungen, die die Steuerelemente aktivieren beziehungsweise deaktivieren. Sie werden sehen: Das Aktivieren und Deaktivieren erfolgen jetzt automatisch – und zwar abhängig davon, ob das Command ausgeführt werden kann oder nicht.

Nur das Setzen der Durchlässigkeit für die Symbole müssen Sie selbst erledigen. Die entsprechenden Anweisungen können Sie zum Beispiel in den Methoden `BrowseBack_CanExecute()` und `BrowseForward_CanExecute()` unterbringen. Da es dabei keine Besonderheiten gibt, stellen wir Ihnen die einzelnen Schritte nicht im Detail vor.



Den fertigen Webbrowser mit Commands finden Sie im Ordner **Minibrowser Commands** im heftbezogenen Download-Bereich Ihrer Online-Lernplattform.

Zusammenfassung

Um nicht vorhandene Commands zu implementieren, setzen Sie die Eigenschaften **CanExecute** und **Executed**.

Die Commands können Sie zum Beispiel bei der Eigenschaft **CommandBindings** des Fensters festlegen.

Aufgaben zur Selbstüberprüfung

- 5.1 Was geben Sie bei den Eigenschaften **CanExecute** und **Executed** eines Commands an?

- 5.2 Auf welchen Wert müssen Sie die Eigenschaft **e.CanExecute** setzen, um zu markieren, dass ein Command ausgeführt werden kann?

Schlussbetrachtung

In diesem Studienheft haben Sie ein etwas umfangreicheres Projekt mit der WPF umgesetzt – einen kleinen Webbrowser. Sie wissen jetzt, wie Sie Webseiten anzeigen können und wie Sie zwischen Webseiten navigieren. Außerdem haben Sie einige weitere Steuerlemente der WPF kennengelernt und erfahren, wie Sie Commands implementieren.

Wenn Sie Lust haben, erweitern Sie den Browser aus diesem Heft noch in Eigenregie. Sehen Sie sich zum Beispiel Funktionen in anderen Webbrowsers an und bauen Sie einige dieser Funktionen nach. Die eine oder andere Erweiterung wartet auch wieder wie gewohnt bei den Einsendeaufgaben auf Sie.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Die Nummerierung beginnt bei 0. Die Eigenschaft **Grid.Column** muss daher den Wert 0 erhalten. Da 0 der Standardwert ist, kann die Angabe auch ganz wegbleiben. Die Eigenschaft **Grid.Row** muss den Wert 1 bekommen.
- 1.2 Der Splitter wirkt in der Standardeinstellung auf alle Zeilen – also auch auf die Zeile mit dem Splitter selbst. Sie können dieses Verhalten über die Eigenschaft **ResizeBehavior** steuern.
- 1.3 Wenn die Seite vollständig geladen wurde, tritt das Ereignis **Navigated** ein.
- 1.4 Den Titel einer Seite erhalten Sie über die Eigenschaft **Document.Title** des Web-Browser-Steuerelements. Diese Eigenschaft steht allerdings nur zur Laufzeit zur Verfügung. Sie müssen daher mit dem Zusatz `dynamic` auf die Eigenschaft zugreifen.

Kapitel 2

- 2.1 Die Eigenschaft **DialogResult** wird für ein Fenster gesetzt. Die Eigenschaft kann den Wert `true` oder `false` erhalten.
- 2.2 Sie sollten den Eigentümer des modalen Fensters über die Eigenschaft **owner** setzen.
- 2.3 Um eine Schaltfläche zur Standardschaltfläche zu machen, setzen Sie die Eigenschaft **IsDefault**.
- 2.4 Wenn sich der Text in einem Eingabefeld ändert, wird das Ereignis **TextChanged** ausgelöst.

Kapitel 3

- 3.1 Um auf einer Schaltfläche gleichzeitig eine Grafik und einen Text anzuzeigen, setzen Sie ein **StackPanel** in die Schaltfläche und ordnen in diesem Container ein Steuerelement vom Typ **TextBlock** und eine Image-Komponente an.
- 3.2 Die Abfrage könnte so aussehen:
`if (e.Key == Key.Enter)`

Kapitel 4

- 4.1 Bei der Sichtbarkeit `Visibility.Collapsed` wird auch der Platz für das Steuerelement im Layout freigegeben. Das Steuerelement wird also beim Layout nicht mehr berücksichtigt. Bei der Sichtbarkeit `Visibility.Hidden` wird das Steuerelement lediglich ausgeblendet.
- 4.2 Damit die Markierung bei einem Menüeintrag automatisch umgeschaltet werden kann, müssen Sie die Eigenschaft **IsCheckable** setzen.
- 4.3 Der Typ `GridLength` bestimmt neben der Breite auch noch den Layouttyp der Spalte.
- 4.4 Erst wird die Markierung entfernt.
- 4.5 Die Eigenschaft **IsIndeterminate** steuert, ob eine Animation in dem Steuerelement angezeigt wird.
- 4.6 Das Ereignis tritt ein, bevor die eigentliche Navigation stattfindet.

Kapitel 5

- 5.1 Bei der Eigenschaft **CanExecute** geben Sie eine Methode an, die die Eigenschaft `e.CanExecute` auf `true` setzt, wenn das Command ausgeführt werden kann. Bei der Eigenschaft **Executed** geben Sie die Methode an, die beim Ausführen des Commands aufgerufen wird.
- 5.2 Die Eigenschaft muss den Wert `true` erhalten.

B. Glossar

Aktivierreihenfolge	Die Aktivierreihenfolge in einem Fenster bestimmt, in welcher Reihenfolge der Fokus auf die Steuerelemente gesetzt wird.
Fokus	Der Fokus in einem Fenster bestimmt, welches Steuer-element aktuell auf Tastatureingaben reagiert.
HTML	HTML steht für <i>Hypertext Markup Language</i> . HTML ist eine Auszeichnungssprache, die im Internet eingesetzt wird.
Hypertext Markup Language	Siehe HTML
Modales Fenster	Ein modales Fenster steht immer im Vordergrund und kann auch nicht minimiert werden. Mit der eigentlichen Anwendung kann erst dann weitergearbeitet werden, wenn das modale Fenster wieder geschlossen wird. Typische Beispiele für modale Fenster sind unter anderem die Öffnen- und Speicherndialoge von Windows.
Navigationsverlauf	Im Navigationsverlauf eines Webbrowsers werden die Seiten gespeichert, die Sie in der aktuellen Sitzung besucht haben.
Uniform Resource Identifier	Siehe URI
Uniform Resource Locator	Siehe URL
URI	URI steht als Abkürzung für <i>Uniform Resource Identifier</i> . Ein URI ist die eindeutige Kennzeichnung einer Ressource im Internet. Das kann eine Webseite oder auch eine Datei sein.
URL	URL steht als Abkürzung für <i>Uniform Resource Locator</i> . Eine URL ist die eindeutige Adresse eines Internetangebots – zum Beispiel www.wetter.de.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 6. Aufl., Bonn: Rheinwerk.

Speziell mit dem Thema WPF beschäftigt sich das folgende Buch:

Huber, T.C. (2019). *Windows Presentation Foundation. Das umfassende Handbuch*. 5. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Die Endversion des Webbrowsers	4
Abb. 1.2	Der schematische Aufbau des Grids	6
Abb. 1.3	Das Fenster im ersten Rohbau	7
Abb. 1.4	Das Fenster mit den Steuerelementen	10
Abb. 1.5	Der Webbrowser in Aktion	11
Abb. 1.6	Die Anzeige im Listenfeld und in der Titelleiste	13
Abb. 2.1	Das geöffnete Menü Datei im Designer	17
Abb. 2.2	Das Fenster zum Öffnen einer URL	18
Abb. 2.3	Das geänderte Menü Datei	21
Abb. 2.4	Der Dialog Öffnen im praktischen Einsatz	22
Abb. 3.1	Die Symbolleiste für die Navigation	28
Abb. 3.2	Die Navigationssymbole im praktischen Einsatz	29
Abb. 3.3	Die Symbolleiste Adresse	30
Abb. 3.4	Das Eingabefeld im Einsatz	32
Abb. 3.5	Der Browser nach dem Feinschliff	35
Abb. 4.1	Das Menü mit den Markierungen	38
Abb. 4.2	Der Rohbau der Statusleiste	43

E. Tabellenverzeichnis

Tab. 2.1	Die Eigenschaften für das zweite Fenster	18
Tab. 3.1	Eigenschaften für die Schaltflächen	28

F. Codeverzeichnis

Code 1.1	Das Grid	6
Code 1.2	Die Gerüste für die Bedienelemente	8
Code 1.3	Die Anweisungen für die Methode Browser_Navigated()	12
Code 2.1	Die Methoden für den Dialog	19
Code 2.2	Die Anweisungen für das Ereignis Click	20
Code 2.3	Die Methode für das Ereignis TextChanged	23
Code 2.4	Die Methoden für die neuen Menüeinträge	24
Code 3.1	Die XAML-Anweisungen für die Symbolleisten	28
Code 3.2	Der XAML-Code für die Schaltfläche	30
Code 3.3	Die Methode ButtonLos_Click()	31
Code 3.4	Das Symbol für den Menüeintrag Vorwärts	32
Code 3.5	Die Methode Eingabefeld_KeyDown()	33
Code 3.6	Das Aktivieren und Deaktivieren der Symbole und Menüeinträge	34
Code 4.1	Die Methoden zum Ein- und Ausblenden der Symbolleisten und der Navigationsliste	39
Code 4.2	Die Methode Liste_SelectionChanged()	40
Code 4.3	Das Löschen der Liste	41
Code 4.4	Die geänderte Methode Liste_SelectionChanged()	41
Code 4.5	Die Statusleiste	42
Code 4.6	Das Setzen der Anzeigen im Ereignis Navigating	43
Code 4.7	Das Meldungsfenster	44
Code 5.1	Die XAML-Anweisungen für die Commands	48
Code 5.2	Die Methoden für die Commands	49

G. Sachwortverzeichnis

C	Navigationsverlauf.....	24
Command		
implementieren	48	
D		
Definition		
für die Spalten	7	
für die Zeilen	6	
E		
Eigenschaft		
für die Schaltflächen	28	
Eigentümer		
eines modalen Fensters setzen	20	
F		
Fenster		
modales, erstellen.....	18	
Fokus		
setzen	22	
Fortschrittsanzeige	42	
G		
Grid.....	5	
GridSplitter	5	
Größe		
automatisch berechnete	6	
proportionale	6	
K		
Kontext-Menü	40	
L		
Listenfeld	9	
M		
Meldungsfenster	44	
Menüeintrag	16	
verschieben	21	
Menüleiste	8, 16	
N		
Navigation	23	
P		
Position		
im Grid festlegen.....	8	
Programmsymbol		
festlegen	45	
S		
Schaltfläche		
mit Grafik und Text erzeugen	30	
Shortcut	48	
Statusleiste	42	
Symbolleiste	27	
T		
Tabulatorreihenfolge		
setzen	22	
Tastaturereignis	33	
Taste		
abfragen.....	33	
V		
Verhalten		
des GridSplitters festlegen	11	

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH16D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Schicken Sie für die Lösungen bitte jeweils das vollständige Projekt mit allen Unterordnern und Dateien ein. Da sich alle Aufgaben auf den Browser beziehen, müssen Sie nur ein Projekt einreichen, das alle Lösungen enthält. Markieren Sie im Quelltext deutlich, zu welcher Aufgabe die jeweiligen Anweisungen gehören – zum Beispiel durch Kommentare. Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

Beschreiben Sie bei allen Aufgaben zusätzlich, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Änderungen Sie vornehmen beziehungsweise welche Steuerelemente Sie verwenden.

1. Erstellen Sie für den Browser eine Funktion, die die beiden Symbolleisten und die Statusleiste gleichzeitig aus- beziehungsweise einblendet. Der Aufruf der Funktion soll über einen Eintrag **Vollbild** im Menü **Ansicht** erfolgen. Sorgen Sie dafür, dass der Menüeintrag bei aktiver Vollbilddarstellung mit einem Häkchen markiert ist.

Achten Sie bitte auch darauf, dass die Markierungen bei den anderen Funktionen im Menü **Ansicht** beim Aufruf der Funktion **Vollbild** entsprechend geändert werden. Denken Sie außerdem daran, dass die Markierung für den Eintrag **Vollbild** geändert werden muss, wenn bei aktiver Vollbilddarstellung zum Beispiel eine Symbolleiste wieder eingeblendet wird.

30 Pkt.

2. Erweitern Sie den Browser so, dass bei der Eingabe einer Adresse geprüft wird, ob ein Protokoll angegeben wurde. Wenn nicht, soll automatisch das Protokoll `http://` ergänzt werden.

20 Pkt.

3. Überarbeiten Sie die Methode zum Einfügen der Einträge in die Navigationsliste so, dass nur noch Seiten eingefügt werden, die nicht bereits in der Liste stehen.

Ein Tipp zur Lösung:

Die komplette Liste der Einträge erhalten Sie über die Eigenschaft **Items** des Listenelements. Mit der Methode `Contains()` können Sie prüfen, ob diese Liste einen bestimmten Eintrag enthält.

50 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Ein Memory-Spiel

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1019N01

CSHP17D

Objektorientierte Software-Entwicklung mit C#

Ein Memory-Spiel

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Ein Memory-Spiel

Inhaltsverzeichnis

Einleitung	1
1 Vorgaben für das Spiel	3
2 Die ersten Schritte	5
2.1 Fenster einrichten	5
2.2 Spielfeld erstellen	6
2.3 Die Klasse MemoryKarte	7
2.4 Die Klasse für das Spielfeld	11
Zusammenfassung	17
3 Die Spielfeldverwaltung	18
3.1 Vorüberlegungen	18
3.2 Die Methode Karteoeffnen()	20
3.3 Die weiteren Methoden für den Spielablauf	25
3.4 Ein wenig Feinschliff	29
Zusammenfassung	33
4 Der Computer als Spielpartner	34
4.1 Vorüberlegungen	34
4.2 Die praktische Umsetzung	35
4.3 Das Einstellen der Spielstärke	38
4.4 Der letzte Feinschliff	39
Zusammenfassung	42
Schlussbetrachtung	44
Anhang	
A. Lösungen der Aufgaben zur Selbstüberprüfung	45
B. Glossar	46
C. Literaturverzeichnis	47
D. Abbildungsverzeichnis	48
E. Codeverzeichnis	49
F. Sachwortverzeichnis	50
G. Einsendeaufgabe	51

Einleitung

In diesem Studienheft werden wir ein Spiel programmieren – und zwar ein Memory-Spiel, das Sie wahrscheinlich noch aus Ihrer Kindheit kennen. Unser Spielpartner soll dabei der Computer sein.

Beim Erstellen der Oberfläche für das Spiel werden wir auf WPF-Steuerelemente zurückgreifen. Zum Teil erstellen wir diese Steuerelemente dynamisch – also zur Laufzeit des Spiels. Die gesamte Logik des Spiels werden wir ebenfalls per Hand programmieren. Dabei erstellen wir unter anderem eine zufällige Verteilung der Karten auf dem Spielfeld, Methoden zum Anzeigen und Verbergen der Spielkarten sowie die „Intelligenz“ des Spielpartners Computer.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie Tabellen mit dem Container **UniformGrid** erstellen,
- wie Sie Bilder in einer Liste verwalten,
- wie Sie Bilder zur Laufzeit auf einer Schaltfläche anzeigen lassen,
- wie Sie eine Methode dynamisch zur Laufzeit des Programms mit einem Ereignis verbinden,
- wie Sie Daten in einem Array zufällig mischen,
- wie Sie einen Timer in einer WPF-Anwendung einsetzen und
- wie Sie einen Algorithmus für den Computergegner implementieren.

Christoph Siebeck

1 Vorgaben für das Spiel

*Bevor wir mit der Erstellung des Fensters und der Umsetzung des Spiels beginnen, legen wir noch ein paar **Vorgaben** und die **Spielregeln** fest.*

Folgende Regeln und Vorgaben sollen gelten:

- Spielpartner ist immer der Computer.
- Unser Spielfeld hat eine feste Größe von 7×6 Feldern. Insgesamt spielen wir also mit 42 Karten.
- Jede Karte ist doppelt vorhanden. Wir haben also 21 Paare.
- Beim Spielbeginn enthält jedes Feld eine umgedrehte Karte.
- Die Spieler decken der Reihe nach jeweils zwei Spielkarten auf.
- Wenn ein Spieler zwei gleiche Karten aufdeckt, darf er das Paar vom Spielfeld nehmen und noch einmal zwei Karten umdrehen.
- Findet ein Spieler bei einem Zug keine identischen Karten, werden die Karten wieder umgedreht und der andere Spieler ist an der Reihe.
- Das Spiel soll beendet werden, wenn alle Paare entfernt wurden.
- Gewonnen hat der Spieler, der die meisten Paare gefunden hat.

Hinweis:

Die Grafiken für die Spielkarten finden Sie bei den Beispielen im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. Bitte kopieren Sie diese Grafiken nachher in einen eigenen Unterordner im Projekt. Andernfalls werden die Grafiken später beim Spielen nicht korrekt geladen.

So viel zu den Vorgaben und Regeln.

Programmiertechnisch setzen wir das Spiel im Wesentlichen mit drei Klassen um, die wir Schritt für Schritt ausbauen werden. Eine Klasse bildet eine einzelne Spielkarte ab, die andere Klasse das Spielfeld und die Logik des Spiels. Eine weitere Klasse steht für die eigentliche Anwendung. Sie enthält auch den Container für das Spielfeld.

Das fertige Spiel soll ungefähr so aussehen:

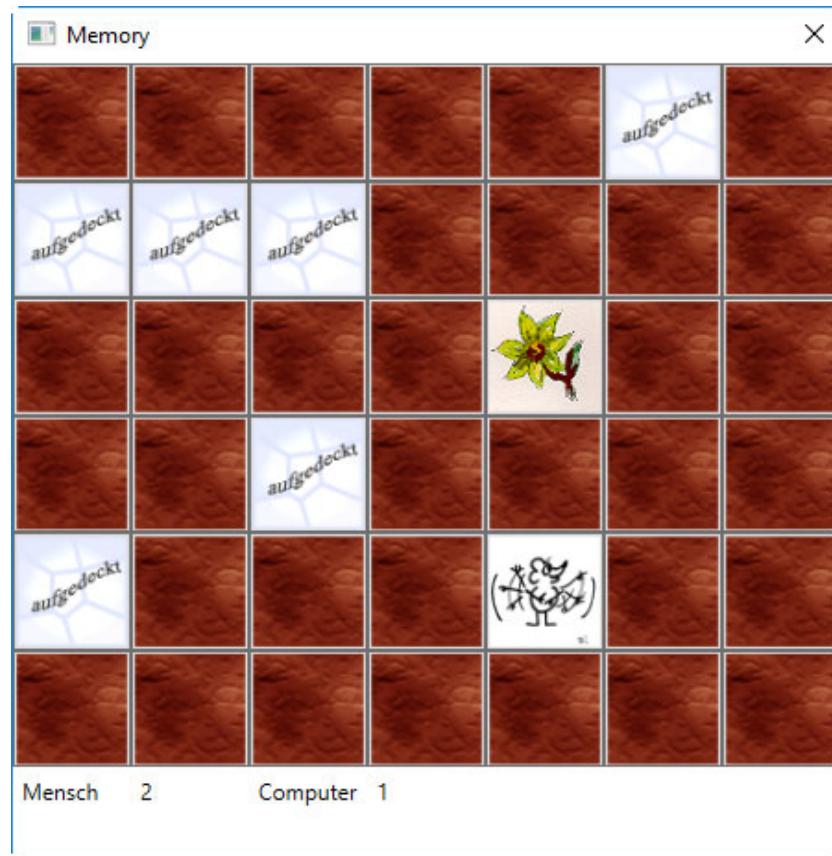


Abb. 1.1: Das Memory-Spiel

2 Die ersten Schritte

In diesem Kapitel werden wir das Fenster für das Spiel anlegen, die Karten für das Memory-Spiel erstellen und auf dem Spielfeld anzeigen.

Beginnen wir mit der Oberfläche.

2.1 Fenster einrichten

Für die Oberfläche des Spiels verwenden wir ein Fenster, in dem wir nur ein **UniformGrid** für das Spielfeld mit den Karten sowie die **Labels** für die Anzeige der gefundenen Paare unterbringen werden.

Legen Sie bitte ein neues Projekt für eine WPF-Anwendung an. Als Namen können Sie zum Beispiel **Memory** verwenden.

Setzen Sie dann die Eigenschaft **Title** für das Fenster auf **Memory**. Damit das Fenster der Anwendung nicht vergrößert oder verkleinert werden kann, setzen Sie die **Eigenschaft ResizeMode** in der Gruppe **Allgemein** des Eigenschaftenfensters auf **NoResize**.

Die Breite des Fensters ändern Sie bitte in 465 Pixel und die Höhe in 470 Pixel. Da jede unserer Spielkarten 64 Pixel hoch und 64 Pixel breit ist, lässt sich mit diesen Einstellungen eine Tabelle mit sieben Spalten und sieben Reihen darstellen. Zwischen den einzelnen Karten bleibt dabei ein wenig Platz. Die letzte Reihe benutzen wir für die Labels zur Anzeige der Punkte.

Zur Auffrischung:

Um die Breite und die Höhe eines Fensters exakt einzustellen, verwenden Sie die Eigenschaften **Width** (für die Breite) und **Height** (für die Höhe). Sie finden die beiden Eigenschaften in der Gruppe **Layout** des Eigenschaftenfensters. Sie können die Änderungen aber auch direkt im XAML-Code vornehmen.



Nach den Änderungen sollte Ihr Fenster so aussehen:

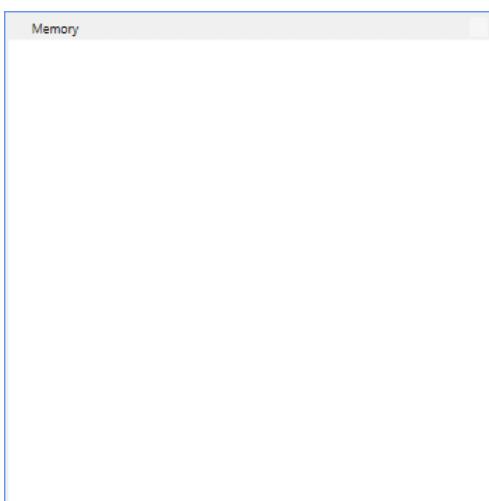


Abb. 2.1: Das Fenster

Im nächsten Schritt erstellen wir jetzt das Spielfeld.

2.2 Spielfeld erstellen

Unser Spielfeld hat eine feste Größe von 7×6 Feldern. Diese Struktur können wir sehr gut über eine Tabelle mit sieben Spalten und sechs Zeilen abbilden. Jede einzelne Zelle der Tabelle nimmt dabei eine Karte auf. Eine weitere Reihe benötigen wir später für die Anzeige der aktuellen Punkte. Damit brauchen wir also 7×7 Felder.

Für das Erstellen der Tabelle verwenden wir den Container **UniformGrid**. Er entspricht einem normalen Grid-Container, verwendet aber dieselbe Größe für alle Zellen. Ändern Sie bitte die Tags `<Grid>` und `</Grid>` in `<UniformGrid>` und `</UniformGrid>`.

Ändern Sie anschließend den Namen des Containers über das Eigenschaftenfenster – zum Beispiel in `spielfeld`. Die Höhe des Containers setzen Sie zunächst auf 445 und die Breite auf 448. Die entsprechenden Eigenschaften **Height** und **Width** finden Sie in der Gruppe **Layout** des Eigenschaftenfensters. Ändern Sie dann noch die Eigenschaft **Rows** für die Zeilen in 7 und die Eigenschaft **Columns** für die Spalten in 7. Die beiden Eigenschaften finden Sie in der Gruppe **Allgemein**.

Richten Sie den Container anschließend noch links oben im Fenster aus. Klicken Sie dazu bei den Eigenschaften **HorizontalAlignment** und **VerticalAlignment** jeweils auf das Symbol ganz links. Das Fenster sollte nun so aussehen:

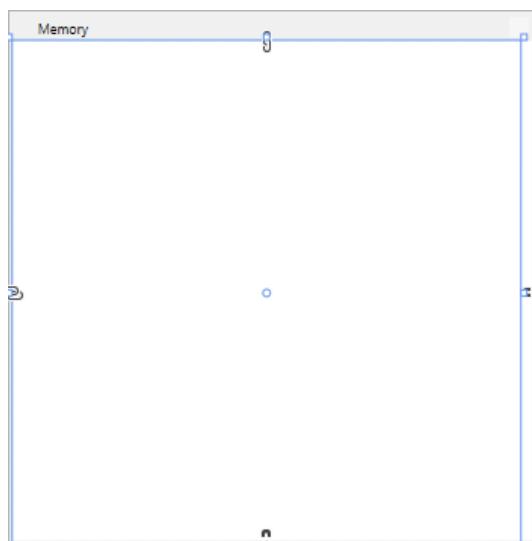


Abb. 2.2: Der Container im Fenster

Kommen wir jetzt zur Klasse für die Memory-Karten.

2.3 Die Klasse MemoryKarte

Die Klasse für die Memory-Karten soll folgende Eigenschaften einer Karte speichern können:

- ein Bild für die Vorderseite,
- ein Bild für die Rückseite,
- eine eindeutige Kennzeichnung für das Bild auf der Vorderseite,
- eine Markierung, ob gerade die Vorderseite der Karte angezeigt wird,
- eine Markierung, ob die Karte noch im Spiel ist, und
- die aktuelle Position der Karte im Spielfeld.

Als Methoden bilden wir in der Klasse nur das Anzeigen der Vorder- beziehungsweise Rückseite der Karte sowie einige Hilfsmethoden ab. Die eigentliche Spiellogik – also zum Beispiel die Prüfung, ob ein Paar gefunden wurde – programmieren wir später in einer eigenen Klasse.

Damit wir die Bilder möglichst einfach anzeigen lassen können und auch ohne großen Aufwand auf das Anklicken einer Karte reagieren können, leiten wir unsere Klasse für die Karten von der Klasse `Button` ab.

Legen Sie jetzt bitte mit Visual Studio eine Datei für eine neue Klasse an. Rufen Sie dazu die Funktion **Projekt/Neues Element hinzufügen ...** auf und wählen Sie den Eintrag **Klasse** in der mittleren Liste aus. Als Name verwenden Sie bitte `MemoryKarte`.

Übernehmen Sie für diese Klasse zunächst einmal die Anweisungen aus dem folgenden Code beziehungsweise passen Sie die vorhandenen Anweisungen entsprechend an:

```
using System;

//die zusätzlichen Namensräume
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Imaging;

namespace Memory
{
    //die Klasse für eine Karte des Memory-Spiels
    //Sie erbt von Button
    class MemoryKarte : Button
    {
        //die Felder
        //eine eindeutige ID zur Identifizierung des Bildes
        int bildID;
        //für die Vorder- und Rückseite
        Image bildVorne, bildHinten;

        //wo liegt die Karte im Spielfeld?
        int bildPos;

        //ist die Karte umgedreht?
        bool umgedreht;
```

```
//ist die Karte noch im Spiel?  
bool nochImSpiel;  
  
//der Konstruktor  
//er setzt die Größe, die Bilder und die Position  
public MemoryKarte(string vorne, int bildID)  
{  
    //die Vorderseite, der Dateiname des Bildes wird an den  
    //Konstruktor übergeben  
    bildVorne = new Image();  
    //bitte in einer Zeile eingeben  
    bildVorne.Source = new BitmapImage(new Uri(vorne,  
        UriKind.Relative));  
    //die Rückseite, sie wird fest gesetzt  
    bildHinten = new Image();  
    //bitte in einer Zeile eingeben  
    bildHinten.Source = new BitmapImage(new  
        Uri("grafiken/verdeckt.bmp", UriKind.Relative));  
    //die Eigenschaften zuweisen  
    Content = bildHinten;  
    //die Bild-ID  
    this.bildID = bildID;  
    //die Karte ist erst einmal umgedreht und noch im Feld  
    umgedreht = false;  
    nochImSpiel = true;  
    //die Methode mit dem Ereignis verbinden  
    Click += new RoutedEventHandler(ButtonClick);  
}  
  
//die Methode für das Anklicken  
private void ButtonClick(object sender, RoutedEventArgs e)  
{  
    //ist die Karte überhaupt noch im Spiel?  
    if (nochImSpiel == false)  
        return;  
    //wenn die Rückseite zu sehen ist, die Vorderseite  
    //anzeigen  
    if (umgedreht == false)  
    {  
        Content = bildVorne;  
        umgedreht = true;  
    }  
}  
  
//die Methode zeigt die Rückseite der Karte an  
public void RueckseiteZeigen(bool rausnehmen)  
{  
    //soll die Karte komplett aus dem Spiel genommen werden?  
    if (rausnehmen == true)  
    {  
        //das Bild aufgedeckt zeigen und die Karte aus dem  
        //Spiel nehmen  
        Image bildRausgenommen = new Image();
```

```

        //bitte in einer Zeile eingeben
        bildRausgenommen.Source = new BitmapImage(new
        Uri("grafiken/aufgedeckt.bmp", UriKind.Relative));
        Content = bildRausgenommen;
        nochImSpiel = false;
    }
    else
    {
        //sonst nur die Rückseite zeigen
        Content = bildHinten;
        umgedreht = false;
    }
}

//die Methode liefert die Bild-ID einer Karte
public int GetBildID()
{
    return bildID;
}

//die Methode liefert die Position einer Karte
public int GetBildPos()
{
    return bildPos;
}

//die Methode setzt die Position einer Karte
public void SetBildPos(int bildPos)
{
    this.bildPos = bildPos;
}
}
}

```

Code 2.1: Die Klasse MemoryKarte

Schauen wir uns die Besonderheiten der Klasse der Reihe nach an.

Mit der Anweisung

```
class MemoryKarte : Button
```

leiten wir unsere Klasse `MemoryKarte` von der Klasse `Button` ab. Dieses Verfahren kennen Sie ja bereits.

Hinweis:

Damit die Klasse `Button` bekannt ist, müssen Sie den Namensraum `System.Windows.Controls` einbinden.

Mit der Anweisung

```
Image bildVorne, bildHinten;
```

vereinbaren wir zwei Felder für ein Objekt der Klasse `Image`. Über diese Klasse können wir – wie Sie ja bereits wissen – eine **Grafik für eine Schaltfläche laden**. Das Feld `bildVorne` benutzen wir für die Vorderseite der Spielkarte und das Feld `bildHinten` für die Rückseite der Spielkarte.

Im Konstruktor der Klasse `MemoryKarte` weisen wir die Bilder für die Vorder- und Rückseite zu. Das übernehmen die Anweisungen

```
bildVorne = new Image();
bildVorne.Source = new BitmapImage(new
Uri(vorne, UriKind.Relative));
bildHinten = new Image();
bildHinten.Source = new BitmapImage(new
Uri("grafiken/verdeckt.bmp", UriKind.Relative));
```

Hier erzeugen wir jeweils eine neue Instanz der Klasse `Image` und weisen der Eigenschaft `Source` dieser Instanz über die Klasse `BitmapImage` das Bild zu. Die Klasse `BitmapImage` selbst beschafft sich die Daten über eine neue Instanz der Klasse `Uri`. Diese Instanz erwartet den Namen und gegebenenfalls den Pfad der Bilddatei. Für die Vorderseite wird dieser Name und Pfad als Argument an den Konstruktor der Klasse `MemoryKarte` übergeben, für die Rückseite nehmen wir die Grafik `verdeckt.bmp` im Unterordner `\grafiken` des Ordners, in dem sich das Projekt für das Memory-Spiel befindet. Wenn sich Ihre Grafiken in einem anderen Ordner befinden, müssen Sie den Pfad gegebenenfalls anpassen. Damit der aktuelle Projektordner als Ausgangspunkt für die Suche benutzt wird, müssen Sie jeweils die Angabe `UriKind.Relative` setzen. Sie stellt sicher, dass relative Pfadangaben genutzt werden.

Danach lassen wir die Grafik für die Rückseite der Karte auf der Karte anzeigen, setzen das Feld für die eindeutige Identifikation des Bildes auf der Karte sowie die beiden Felder `umgedreht` und `nochImSpiel`.

Abschließend verbinden wir die Karte mit einem **Eventhandler für das Klicken**. Das erledigt die Anweisung

```
Click += new RoutedEventHandler(ButtonClick);
```

Eine ähnliche Technik haben wir ja bereits verwendet, als wir den Menüeintrag für die zuletzt geöffnete Datei erzeugt haben. Hier verwenden wir allerdings nicht die Klasse `EventHandler`, sondern `RoutedEventHandler`.

Hinweis:

Bitte denken Sie unbedingt daran, dass Sie beim Eventhandler nur den Namen der Methode angeben. Die runden Klammern hinter dem Namen dürfen Sie nicht setzen.

In der Methode `ButtonClick()` überprüfen wir zunächst mit der Anweisung

```
if (nochImSpiel == false),
```

ob die Karte überhaupt noch im Spiel ist. Wenn das nicht der Fall ist, verlassen wir die Methode direkt wieder. Das heißt also, wenn auf eine Karte geklickt wird, die bereits aus dem Spiel genommen wurde, geschieht nichts weiter.

Befindet sich die Karte dagegen noch im Spiel und ist aktuell die Rückseite zu sehen, lassen wir mit der Anweisung

```
Content = bildVorne;
```

das Bild für die Vorderseite auf der Schaltfläche beziehungsweise der Spielkarte anzeigen und setzen das Feld `umgedreht` auf `true`, damit das Programm weiß, dass gerade die Vorderseite der Karte zu sehen ist.

Mit der Methode `RueckseiteZeigen()` drehen wir eine Karte wieder um. Über den Parameter `rausnehmen` können wir dabei steuern, ob die Karte komplett aus dem Spiel genommen werden soll oder einfach wieder nur die Rückseite angezeigt werden soll. Falls die Karte komplett aus dem Spiel genommen wird, lassen wir die Grafik **aufgedeckt.bmp** auf der Karte anzeigen und setzen das Feld `nochImSpiel` auf `false`.

Mit den drei Methoden `GetBildID()`, `GetBildPos()` und `SetBildPos()` schließlich ermöglichen wir von außen den Zugriff auf die Felder `bildPos` und `bildID`. Dabei gibt es keine Besonderheiten.

Hinweis:

Wir lassen die Größe der Karten in unserem Programm durch den Container und die Grafiken für die Spielkarten setzen. Wenn Sie die Größe selbst vorgeben wollen, können Sie die Eigenschaften `Width` und `Height` entsprechend setzen.

Die Verbindung zum Spielfeld haben wir in der Klasse `MemoryKarte` an dieser Stelle noch nicht umgesetzt. Das holen wir im nächsten Kapitel nach, wenn wir die ersten Teile der Klasse für das Spielfeld programmiert haben.

2.4 Die Klasse für das Spielfeld

Kommen wir nun zur Klasse `MemoryFeld`. Sie soll das Spielfeld mit den Karten verwalten. Die Spielkarten speichern wir dabei in einem Array vom Typ `MemoryKarte` und die Namen der Bilddateien für die Vorderseiten legen wir in einem Array vom Typ `string` ab.

Außerdem soll die Klasse `MemoryFeld` aber auch die Logik des Spiels abdecken – also zum Beispiel prüfen, wie viele Karten umgedreht sind, ob es sich bei den umgedrehten Karten um ein Paar handelt und so weiter. Dazu brauchen wir neben den passenden Methoden auch einige Felder, die unter anderem die Punkte der Spieler speichern, die Anzahl der aktuell umgedrehten Karten, das aktuell umgedrehte Paar, den aktuellen Spieler und noch einige Informationen mehr.

Eine erste Version der Klasse finden Sie im folgenden Code 2.2:

```
using System;

//die zusätzlichen Namensräume
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;

namespace Memory
{
    class MemoryFeld
    {
        //das Array für die Karten
        MemoryKarte[] karten;
```

```
//das Array für die Namen der Grafiken
string[] bilder =
{"grafiken/apfel.bmp", "grafiken/birne.bmp",
"grafiken/blume.bmp", "grafiken/blume2.bmp",
"grafiken/ente.bmp", "grafiken/fisch.bmp",
"grafiken/fuchs.bmp", "grafiken/igel.bmp",
"grafiken/kaenguruh.bmp", "grafiken/katze.bmp",
"grafiken/kuh.bmp", "grafiken/maus1.bmp",
"grafiken/maus2.bmp", "grafiken/mauss.bmp",
"grafiken/melone.bmp", "grafiken/pilz.bmp",
"grafiken/ronny.bmp", "grafiken/schmetterling.bmp",
"grafiken/sonne.bmp", "grafiken/wolke.bmp",
"grafiken/maus4.bmp"};
```

```
//für die Punkte
int menschPunkte, computerPunkte;
Label menschPunkteLabel, computerPunkteLabel;
```

```
//wie viele Karten sind aktuell umgedreht?
int umgedrehteKarten;
```

```
//für das aktuell umgedrehte Paar
MemoryKarte[] paar;
```

```
//für den aktuellen Spieler
int spieler;
```

```
//das Gedächtnis für den Computer
//er speichert hier, wo das Gegenstück liegt
int[,] gemerkteKarten;
```

```
//für das eigentliche Spielfeld
UniformGrid feld;
```

```
//der Konstruktor
public MemoryFeld(UniformGrid feld)
{
    //zum Zählen für die Bilder
    int count = 0;

    //das Array für die Karten erstellen, insgesamt 42 Stück
    karten = new MemoryKarte[42];

    //für das Paar
    paar = new MemoryKarte[2];

    //für das Gedächtnis
    //es speichert für jede Karte paarweise die Position im
    //Spielfeld
    gemerkteKarten = new int[2, 21];

    //keiner hat zu Beginn einen Punkt
    menschPunkte = 0;
    computerPunkte = 0;
}
```

```
//es ist keine Karte umgedreht  
umgedrehteKarten = 0;  
  
//der Mensch fängt an  
spieler = 0;  
  
//das Spielfeld setzen  
this.feld = feld;  
  
//es gibt keine gemerkten Karten  
for (int aussen = 0; aussen < 2; aussen++)  
    for (int innen = 0; innen < 21; innen++)  
        gemerkteKarten[aussen, innen] = -1;  
  
//das eigentliche Spielfeld erstellen  
for (int i = 0; i <= 41; i++)  
{  
    //eine neue Karte erzeugen  
    karten[i] = new MemoryKarte(bilder[count], count);  
    //die Position der Karte setzen  
    karten[i].SetBildPos(i);  
  
    //die Karte hinzufügen  
    feld.Children.Add(karten[i]);  
    //bei jeder zweiten Karte kommt auch ein neues Bild  
    if ((i + 1) % 2 == 0)  
        count++;  
}  
  
//die Labels für die Punkte  
Label mensch = new Label();  
mensch.Content = "Mensch";  
feld.Children.Add(mensch);  
menschPunkteLabel = new Label();  
menschPunkteLabel.Content = 0;  
feld.Children.Add(menschPunkteLabel);  
  
Label computer = new Label();  
computer.Content = "Computer";  
feld.Children.Add(computer);  
computerPunkteLabel = new Label();  
computerPunkteLabel.Content = 0;  
feld.Children.Add(computerPunkteLabel);  
}  
}  
}
```

Code 2.2: Eine erste Version der Klasse für die Anwendung

Schauen wir uns auch hier wieder die Besonderheiten an.

Mit der langen Anweisung

```
string[] bilder =
{"grafiken/apfel.bmp", "grafiken/birne.bmp",
"grafiken/blume.bmp", "grafiken/blume2.bmp",
"grafiken/ente.bmp", "grafiken/fisch.bmp",
"grafiken/fuchs.bmp", "grafiken/igel.bmp",
"grafiken/kaenguruh.bmp", "grafiken/katze.bmp",
"grafiken/kuh.bmp", "grafiken/maus1.bmp",
"grafiken/maus2.bmp", "grafiken/maus3.bmp",
"grafiken/melone.bmp", "grafiken/pilz.bmp",
"grafiken/ronny.bmp", "grafiken/schmetterling.bmp",
"grafiken/sonne.bmp", "grafiken/wolke.bmp",
"grafiken/maus4.bmp"};
```

legen wir ein Array vom Typ `string` mit den Namen der Bilddateien für die Vorderseiten an. In welcher Reihenfolge Sie die Grafiken in dem Array aufführen, ist dabei beliebig. Achten Sie lediglich darauf, dass Sie 21 Grafiken in dem Array angeben und dass der Pfad stimmt. Andernfalls gibt es später beim Anzeigen beziehungsweise beim Laden Schwierigkeiten.

Bei den anderen Feldern der Klasse sind vor allem die Felder `paar` und `gemerkteKarten` interessant. Das Feld `paar` ist ein Array vom Typ `MemoryKarte` und soll später die beiden aktuell umgedrehten Karten zwischenspeichern. Das Feld `gemerkteKarten` dagegen benutzen wir als eine Art Gedächtnis für den Computer. Hier legen wir nachher die Position jeder bereits einmal umgedrehten Karte ab.

Besondere Bedeutung hat auch noch die Anweisung:

```
UniformGrid feld;
```

Hier hinterlegen wir den Container aus dem Fenster der eigentlichen Anwendung. Die Verbindung erfolgt über den Konstruktor der Klasse `MemoryFeld`.

Im Konstruktor der Klasse erzeugen wir außerdem die Arrays für die Karten, für das aktuell umgedrehte Paar und für das Gedächtnis. Das erledigen die Anweisungen

```
karten = new MemoryKarte[42];
paar = new MemoryKarte[2];
gemerkteKarten = new int[2,21];
```

Das Array `gemerkteKarten` kann dabei paarweise in zwei Dimensionen die Position der Karten speichern. Wie das genau funktioniert, werden wir uns im weiteren Verlauf des Studienhefts noch im Detail ansehen.

Mit der geschachtelten Schleife

```
for (int aussen = 0; aussen < 2; aussen++)
    for (int innen = 0; innen < 21; innen++)
        gemerkteKarten[aussen,innen] = -1;
```

initialisieren wir alle Elemente im Array `gemerkteKarten` mit dem Wert `-1`. Er soll markieren, dass noch keine Karte an der entsprechenden Position gespeichert wurde.

Die folgende `for`-Schleife schließlich legt das eigentliche Spielfeld mit den Karten an. In der Schleife werden die 42 Instanzen der Klasse `MemoryKarte` erzeugt. An den Konstruktor der Klasse `MemoryKarte` übergeben wir dabei das Element mit dem Namen der Grafikdatei aus dem Array `bilder` und die eindeutige Identifizierung für das Bild.

Mit der Anweisung

```
feld.Children.Add(karten[i]);
```

werden die Karten in den Container `feld` eingefügt. Diesen Container haben wir über den Konstruktor an unsere Klasse durchgereicht.

Die `if`-Abfrage am Ende der Schleife sorgt dafür, dass der Wert von `bild` bei jedem zweiten Durchlauf der Schleife erhöht wird. Dazu überprüfen wir mit dem Modulo-Operator `%`, ob sich der Wert von `i + 1` ohne Rest durch 2 teilen lässt. `i + 1` müssen wir verwenden, da `i` selbst ja beim ersten Durchlauf den Wert 0 hat und damit immer um 1 hinter der echten Anzahl der Durchläufe herläuft.

Die `if`-Abfrage ist erforderlich, da wir jedes Bild beziehungsweise jede Karte doppelt benötigen. Nach dem Abarbeiten der Schleife stehen also hintereinander die Bilder der Vorderseiten für die einzelnen Paare im Spielfeld.

Zu guter Letzt nehmen wir jetzt noch vier Labels für die Anzeige der bereits gefundenen Paare in den Container für das Spielfeld auf. Das erledigen die Anweisungen:

```
Label mensch = new Label();
mensch.Content = "Mensch";
feld.Children.Add(mensch);
menschPunkteLabel = new Label();
menschPunkteLabel.Content = 0;
feld.Children.Add(menschPunkteLabel);
Label computer = new Label();
computer.Content = "Computer";
feld.Children.Add(computer);
computerPunkteLabel = new Label();
computerPunkteLabel.Content = 0;
feld.Children.Add(computerPunkteLabel);
```

Hinweis:

Wir erzeugen die Steuerelemente hier zur Übung dynamisch über C#-Anweisungen. Sie können sie auch wie gewohnt im XAML-Code erzeugen und positionieren.

Übernehmen Sie jetzt die Klassen und Anweisungen aus den vorigen Codes. Erzeugen Sie dann im Konstruktor der eigentlichen Anwendung eine neue Instanz der Klasse `MemoryFeld` und übergeben Sie dabei den Bezeichner für den Container. In unserem Beispiel könnte die Anweisung so aussehen:

```
new MemoryFeld(spielfeld);
```

Lassen Sie das Programm anschließend übersetzen, und drehen Sie die Karten jeweils mit einem Mausklick um. Das Ergebnis sollte ungefähr so aussehen:

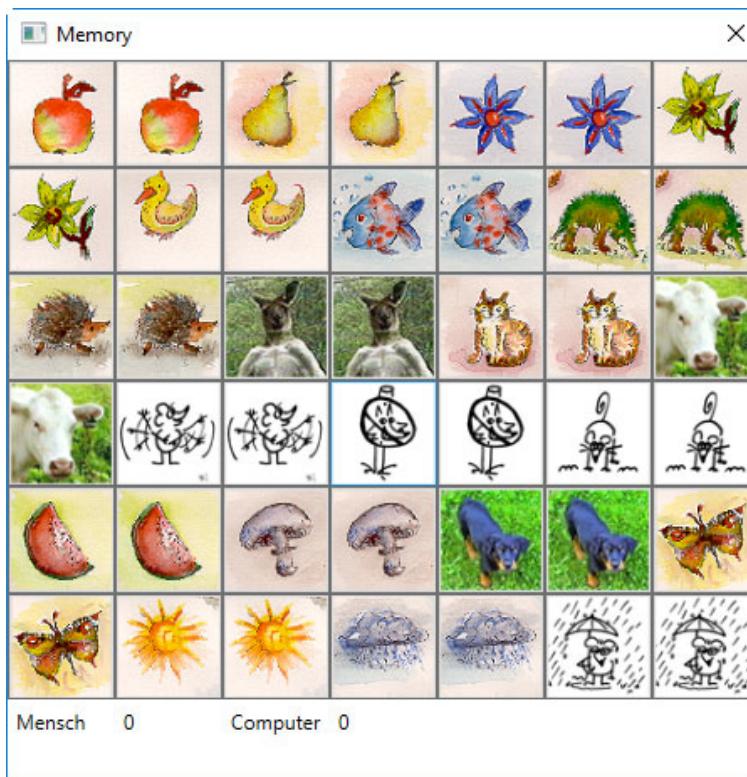


Abb. 2.3: Die erste Version des Memory-Spiels mit allen umgedrehten Karten

Hinweis:

Visual Studio wird Ihnen beim Test melden, dass einige Felder vereinbart, aber nie benutzt werden. Diese Warnung können Sie ignorieren, da wir die Felder ja erst später verwenden werden.

Falls in Ihrem Spielfeld keine Grafiken angezeigt werden, kontrollieren Sie bitte noch einmal sorgfältig, ob Ihnen beim Abtippen der Namen im Array `bilder` keine Fehler unterlaufen sind und ob sich die Grafiken im richtigen Ordner befinden.

Wenn Ihnen das Umdrehen der Karten per Hand zum Test zu mühselig ist, können Sie im Konstruktor der Klasse `MemoryKarte` auch zunächst die Vorderseite anzeigen lassen. Ändern Sie dazu die Anweisung

```
Content = bildHinten;
```

im Konstruktor der Klasse `MemoryKarte` in

```
Content = bildVorne;
```

Denken Sie aber bitte daran, nach dem Test wieder die Rückseite der Karten anzeigen zu lassen.

Wir lassen die Karten erst einmal sortiert als Paare hintereinander stehen, damit der Test der Spiellogik etwas leichter wird. Später sorgen wir noch dafür, dass die Karten zufällig auf dem Spielfeld verteilt werden.

Damit ist die erste Rohversion des Memory-Spiels fertiggestellt. In den nächsten Kapiteln werden wir uns mit der Programmierung der Spiellogik beschäftigen.

Zusammenfassung

Über ein **UniformGrid** können Sie eine Tabelle mit identischer Größe für alle Zellen erstellen.

Mit der Klasse `Image` können Sie Bilder verarbeiten.

Beim Laden von Grafiken über eigene Anweisungen müssen Sie genau auf die Pfadangaben achten.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 2.1 Sie wollen eine Grafik mit dem Namen `testbild.jpg` auf einer Schaltfläche `buttonTest` anzeigen lassen. Formulieren Sie bitte entsprechende C#-Anweisungen.

- 2.2 Sie wollen ein Label mit dem Bezeichner `labelTest` zu einem Container mit dem Namen `conTest` hinzufügen. Wie lautet die entsprechende C#-Anweisung?

3 Die Spielfeldverwaltung

In diesem Kapitel werden wir die Logik des Spiels programmieren – zum Beispiel das Umdrehen der Karten und die Regeln für die Paarbildung.

3.1 Vorüberlegungen

Bevor wir uns an die praktische Umsetzung machen, zuerst ein paar theoretische **Vorüberlegungen zur Steuerung des Spielablaufs:**

- Wir müssen die beiden Karten zwischenspeichern, die in einem Spielzug umgedreht wurden. Dazu verwenden wir das Array `paar`, das wir ja bereits vereinbart haben.
- Damit der Computer weiß, welche Karten sich im Spiel an welcher Stelle befinden, speichern wir für jede umgedrehte Karte die Position im Array `gemerkteKarten` ab. In diesem Array sehen wir dann beim Zug des Computers zuerst nach, ob dort möglicherweise ein Paar abgelegt ist.
- Wir müssen beim Anklicken beziehungsweise Umdrehen einer Karte prüfen, wie viele Karten bereits in einem Spielzug umgedreht wurden. Dazu erhöhen wir das Feld `umgedrehteKarten` um den Wert 1.
- Wenn der Spieler ein Paar gefunden hat, erhält er einen Punkt gutgeschrieben und die beiden Karten werden aus dem Spiel genommen. Findet er dagegen kein Paar, werden die Karten wieder umgedreht und der Spieler wechselt. Welcher Spieler gerade an der Reihe ist, legen wir im Feld `spieler` ab. Die Punkte der beiden Spieler speichern wir in den Feldern `menschPunkte` und `spielerPunkte`.
- Wenn alle Paare umgedreht sind, wird das Spiel beendet.

Als Aktivitätsdiagramm lässt sich dieser Ablauf so darstellen:

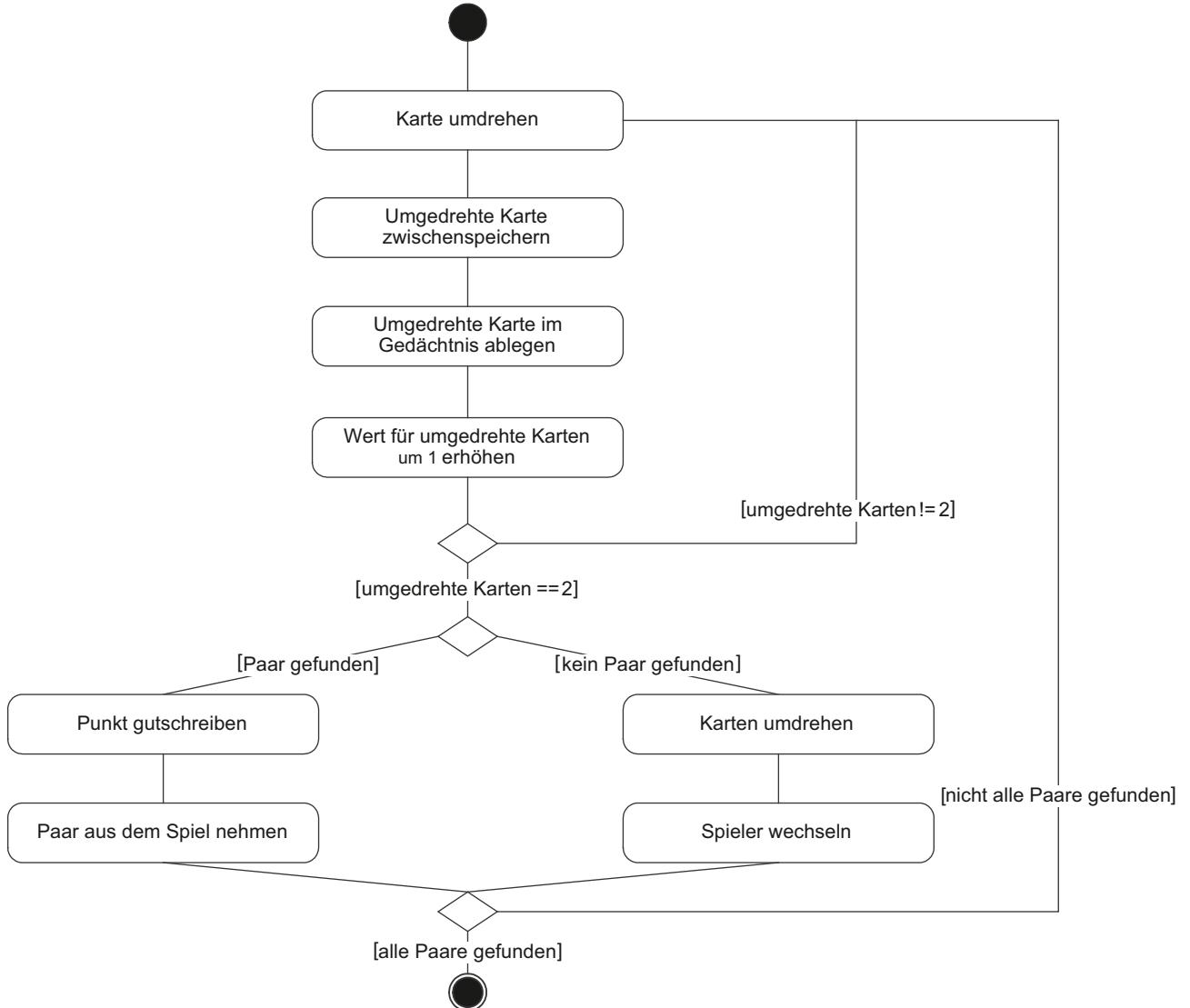


Abb. 3.1: Der vereinfachte Spielablauf

Hinweis:

Die Ausdrücke in den Klammern geben jeweils die Bedingungen an. Mit den Aktivitätsdiagrammen werden wir uns später noch intensiver beschäftigen.

3.2 Die Methode KarteOeffnen()

Den Spielablauf werden wir im Wesentlichen über eine **Methode KarteOeffnen()** in der Klasse `MemoryFeld` abbilden. In dieser Methode werden die umgedrehten Karten in das „Gedächtnis“ des Computers eingetragen, die Prüfungen auf die Paare durchgeführt und auch die Karten wieder umgedreht. Damit die Methode selbst nicht zu umfangreich wird, verteilen wir die Anweisungen auf mehrere andere Methoden.

Die Methode `KarteOeffnen()` rufen wir beim Anklicken einer Karte auf – also aus der Methode `ButtonClick()` der Klasse `MemoryKarte`. Dabei übergeben wir die angeklickte Karte als Argument an die Methode. Damit die Karte auch weiß, in welcher Klasse sich die Methode `KarteOeffnen()` befindet, müssen wir über den Konstruktor der Klasse `MemoryKarte` die Verbindung zwischen den Klassen `MemoryKarte` und `MemoryFeld` für das eigentliche Spiel herstellen.

Ergänzen Sie jetzt im ersten Schritt bitte in der Klasse `MemoryKarte` ein Feld vom Typ `MemoryFeld`. Wir benutzen in unserem Beispiel den Bezeichner `spiel`. Ändern Sie dann den Konstruktor der Klasse `MemoryKarte` so, dass er einen weiteren Parameter vom Typ `MemoryFeld` erhält, und weisen Sie den Wert dieses Parameters dem Feld `spiel` der Klasse `MemoryKarte` zu. Sorgen Sie abschließend noch dafür, dass in der Methode `ButtonClick()` der Klasse `MemoryKarte` die Methode `KarteOeffnen()` in der Klasse des Spielfeldes aufgerufen wird, und übergeben Sie dabei die Karte, die angeklickt wurde.

Die entsprechenden Änderungen sind im folgenden Code noch einmal zusammengefasst und fett markiert:

```
...
//die Klasse für eine Karte des Memory-Spiels
//Sie erbt von Button
class MemoryKarte : Button
{
    //die Felder
    ...
    //ist die Karte noch im Spiel?
    bool nochImSpiel;
    //für das Spielfeld für die Karte
MemoryFeld spiel;
    //der Konstruktor
    //er setzt die Größe, die Bilder und die Position
    public MemoryKarte(string vorne, int bildID, MemoryFeld spiel)
    {
        ...
        //die Karte ist erst einmal umgedreht und noch im Feld
        umgedreht = false;
        nochImSpiel = true;
        //mit dem Spielfeld verbinden
        this.spiel = spiel;
        ...
    }
}
```

```

//die Methode für das Anklicken
private void ButtonClick(object sender, RoutedEventArgs e)
{
    //ist die Karte überhaupt noch im Spiel?
    if (nochImSpiel == false)
        return;
    //wenn die Rückseite zu sehen ist, die Vorderseite anzeigen
    if (umgedreht == false)
    {
        Content = bildVorne;
        umgedreht = true;
        //die Methode Karteoeffnen() im Spielfeld aufrufen
        //übergeben wird dabei die Karte - also die this-Referenz
        spiel.Karteoeffnen(this);
    }
}
...

```

Code 3.1: Die Änderungen an der Klasse MemoryKarte (der Code ist nicht vollständig)

Übergeben Sie dann im Konstruktor der Klasse MemoryFeld die this-Referenz als drittes Argument beim Aufruf des Konstruktors der Klasse MemoryKarte. Die entsprechende Anweisung in der for-Schleife muss also so aussehen:

```
karten[i] = new MemoryKarte(bilder[count], count, this);
```

Im letzten Schritt übernehmen Sie die Methode Karteoeffnen() aus dem folgenden Code in die Klasse MemoryFeld. Was genau in der Methode geschieht, erklären wir Ihnen im Anschluss.

```

//die Methode übernimmt die wesentliche Steuerung des Spiels
//Sie wird beim Anklicken einer Karte ausgeführt
public void Karteoeffnen(MemoryKarte karte)
{
    //zum Zwischenspeichern der ID und der Position
    int kartenID, kartenPos;

    //die Karten zwischenspeichern
    paar[umgedrehteKarten] = karte;
    //die ID und die Position beschaffen
    kartenID = karte.GetBildID();
    kartenPos = karte.GetBildPos();

    //die Karte in das Gedächtnis des Computers eintragen,
    //aber nur dann, wenn es noch keinen Eintrag an der
    //entsprechenden Stelle gibt
    if ((gemerkteKarten[0, kartenID] == -1))
        gemerkteKarten[0, kartenID] = kartenPos;
    else
        //wenn es schon einen Eintrag gibt und der nicht mit
        //der aktuellen Position übereinstimmt, dann haben wir
        //die zweite Karte gefunden
        //Sie wird in die zweite Dimension eingetragen
        if (gemerkteKarten[0, kartenID] != kartenPos)
            gemerkteKarten[1, kartenID] = kartenPos;
}

```

```
//umgedrehte Karten erhöhen
umgedrehteKarten++;

//sind zwei Karten umgedreht worden?
if (umgedrehteKarten == 2)
{
    //dann prüfen wir, ob es ein Paar ist
    PaarPruefen(kartenID);
    //die Karten wieder umdrehen
    KarteSchliessen();
}

//haben wir zusammen 21 Paare, dann ist das Spiel vorbei
if (computerPunkte + menschPunkte == 21)
{
    MessageBox.Show("Das Spiel ist vorbei.");
    Application.Current.Shutdown();
}
```

Code 3.2: Die Methode KarteOeffnen()

Zunächst einmal legen wir in der Methode die gerade umgedrehte Karte, die über das Argument `karte` übergeben wird, in dem Array `paar` ab. Das erledigt die Anweisung:

```
paar[umgedrehteKarten] = karte;
```

Der Wert des Felds `umgedrehteKarten` entscheidet dabei darüber, an welcher Position im Array die Karte gespeichert wird.

Danach beschaffen wir uns die eindeutige Kennzeichnung des Bildes, das auf der Karte angezeigt wird, sowie die Position der Karte im Spielfeld und speichern diese Daten in lokalen Variablen zwischen.

Mit den Anweisungen

```
if ((gemerkteKarten[0, kartenID] == -1))
    gemerkteKarten[0, kartenID] = kartenPos;
else
    if (gemerkteKarten[0, kartenID] != kartenPos)
        gemerkteKarten[1, kartenID] = kartenPos;
```

tragen wir anschließend die Position der Karte in das Gedächtnis des Computers ein – und zwar genau an die Position im Array `gemerkteKarten`, die auch für den Index des Bildes steht, das auf der Karte angezeigt wird. Die Positionen der Karten mit der Grafik `apfel.bmp` würden zum Beispiel in beiden Dimensionen des Arrays an der ersten Stelle (also bei `gemerkteKarten[0, 0]` und `gemerkteKarten[1, 0]`) gespeichert, da diese Grafik im Array `bilder` an erster Stelle steht – also den Index 0 hat.

Zur Erinnerung:

Der Wert `-1` im Array `gemerkteKarten` steht für „keinen Eintrag“ und wurde von uns im Konstruktor allen Elementen im Array zugewiesen.



Den Index der Grafiken haben wir über das Argument `count` an den Konstruktor der Klasse `MemoryKarte` übergeben. Er hat sich genauso schrittweise verändert wie die Variable, über die wir auf die Namen der Grafiken im Array `bilder` zugriffen haben. Die erste Grafik hat also den Index beziehungsweise die Bild-ID 0, die zweite den Index 1 und so weiter.

Jede Karte – und damit auch jede Bild-ID – gibt es doppelt.

Damit wir die Positionen der Karten nicht ständig überschreiben, prüfen wir zuerst mit der Abfrage

```
if ((gemerkteKarten[0, kartenID] == -1)),
```

ob es schon einen Eintrag für die Karte in der ersten Dimension gibt. Wenn das nicht der Fall ist, tragen wir die Position in der ersten Dimension ein.

Gibt es dagegen einen Eintrag in der ersten Dimension des Arrays, überprüfen wir, ob dieser Eintrag mit der aktuellen Position übereinstimmt. Stimmen die beiden Werte nicht überein, haben wir die zweite Karte für das Bild gefunden und wir können die Position in der zweiten Dimension des Arrays `gemerkteKarten` an derselben Stelle wie in der ersten Dimension eintragen. Damit stehen dann die Positionen eines Paares sauber geordnet in den beiden Dimensionen des Arrays.

Da der Ablauf auf dem Trockenen nicht ganz so einfach nachzuvollziehen ist, schauen wir uns das Verfahren an einigen konkreten Beispielen an. Nehmen wir einmal an, der Anwender dreht zuerst die Karte ganz oben links im Spielfeld um und diese Karte hat die Bild-ID 0.

Zur Auffrischung:

Die Nummerierung der Elemente in einem Array beginnt bei 0 und nicht bei 1.

Nach den beiden Zuweisungen

```
kartenID = karte.getBildID();  
kartenPos = karte.getBildPos();
```

hat `kartenID` den Wert 0 (die Bild-ID) und `kartenPos` ebenfalls den Wert 0 – nämlich die Position der Karte ganz oben links.

Die Abfrage

```
if ((gemerkteKarten[0, kartenID] == -1))
```

ergibt `true`, da ja noch gar kein Eintrag erfolgt ist. Also wird die Position der Karte in die erste Dimension des Arrays `gemerkteKarten` eingetragen. Das Array sieht nun so aus:

0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Abb. 3.2: Die erste eingetragene Karte im Gedächtnis (ganz links oben)

Nehmen wir an, beim zweiten Klicken dreht der Anwender die dritte Karte in der ersten Reihe um. Sie hat die Bild-ID 1 (für die zweite Grafikdatei aus dem Array `bilder`) und die Position 2. Die Abfrage

```
if ((gemerkteKarten[0, kartenID] == -1))
```

liefert dann ebenfalls `true` und die Karte wird wieder eingetragen.

0	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Abb. 3.3: Die zweite eingetragene Karte im Gedächtnis

Klickt der Anwender nun im nächsten Zug auf die zweite Karte mit der Bild-ID 0, liefert die Abfrage

```
if ((gemerkteKarten[0, kartenID] == -1))
```

false, da an der Position 0 in der ersten Dimension bereits ein Wert steht.

Die Abfrage

```
if (gemerkteKarten[0, kartenID] != kartenPos)
```

im else-Zweig dagegen liefert `true`, da die „gemerkte“ Karte an einer anderen Position steht als die gerade geöffnete Karte. Die Position der neuen Karte wird also in der zweiten Dimension des Arrays `gemerkteKarten` eingetragen und das Array sieht nun so aus:

0	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Abb. 3.4: Die dritte eingetragene Karte im Gedächtnis

Der Computer weiß jetzt also, dass die Karten für das Paar mit dem Bildindex 0 an den Positionen 0 und 1 im Spielfeld liegen.

Schauen wir uns nun noch an, was geschieht, wenn der Anwender noch einmal auf eine Karte klickt, die bereits in der ersten Dimension des Arrays `gemerkteKarten` gespeichert ist. Hier liefert die Abfrage

```
if (gemerkteKarten[0, kartenID] != kartenPos)
```

im `else`-Zweig `false`, da die aktuelle Position und die bereits gespeicherte Position übereinstimmen. Die Karte wird also nicht noch einmal eingetragen.

Hinweis:

Das Abspeichern im Gedächtnis ist nicht komplett ausprogrammiert, da die Position für die zweite Karte eines Paares immer wieder abgelegt wird. Um das zu verhindern, müssten wir eine weitere Abfrage einbauen und den Ablauf noch tiefer schachteln. Wichtig beim Gedächtnis ist aber vor allem, dass für die erste und die zweite Karte eines Paares nicht identische Positionen abgelegt werden. Und dafür reicht unsere Konstruktion aus.

Schauen wir uns jetzt den weiteren Ablauf in der Methode `Karteoeffnen()` an.

Nach dem Eintragen der Karte in das Gedächtnis erhöhen wir die Anzahl der umgedrehten Karten um 1. Wenn zwei Karten umgedreht sind, kontrollieren wir über die Methode `PaarPruefen()`, ob es sich um ein Paar handelt, und drehen danach die Karten über die Methode `KarteSchliessen()` wieder um. Diese Methoden finden Sie ausprogrammiert im nächsten Kapitel.

Abschließend überprüfen wir, ob Mensch und Computer zusammen 21 Punkte haben. Wenn ja, sind alle Paare umgedreht und wir beenden das Spiel nach dem Ausgeben einer Meldung. Für das Beenden benutzen wir dabei die Anweisung:

```
Application.Current.Shutdown();
```

Sie fährt bei einer WPF-Anwendung herunter. Die Anweisung `Close()` können wir hier nicht verwenden, da es sich bei unserer Klasse ja nicht um ein Fenster handelt.

3.3 Die weiteren Methoden für den Spielablauf

Die weiteren Methoden für den Spielablauf sind wieder etwas einfacher. Beginnen wir mit der Methode `PaarPruefen()`.

```
//die Methode prüft, ob ein Paar gefunden wurde
private void PaarPruefen(int kartenID)
{
    if (paar[0].GetBildID() == paar[1].GetBildID())
    {
        //die Punkte setzen
        PaarGefunden();
        //die Karten aus dem Gedächtnis löschen
        gemerkteKarten[0, kartenID] = -2;
        gemerkteKarten[1, kartenID] = -2;
    }
}
```

Code 3.3: Die Methode `PaarPruefen()`

Hier gibt es eigentlich nur eine Besonderheit: Wenn ein Paar gefunden wurde, löschen wir die entsprechenden Einträge im Gedächtnis des Computers, damit er später nicht versucht, diese Karten umzudrehen. Dazu schreiben wir den Wert -2 in das entsprechende Element in beiden Dimensionen des Arrays. Den Wert -1 können wir nicht verwenden, da er ja für einen noch leeren Eintrag steht.

Die Methode `PaarGefunden()` macht nichts weiter, als die Punkte abhängig vom aktuellen Spieler zu erhöhen und den neuen Wert in dem jeweiligen Label anzuzeigen. Sie sieht so aus:

```
//die Methode setzt die Punkte, wenn ein Paar gefunden wurde
private void PaarGefunden()
{
    //spielt gerade der Mensch?
    if (spieler == 0)
    {
        menschPunkte++;
        menschPunkteLabel.Content = menschPunkte.ToString();
    }
    else
    {
        computerPunkte++;
        computerPunkteLabel.Content = computerPunkte.ToString();
    }
}
```

Code 3.4: Die Methode `PaarGefunden()`

Die Methode `KarteSchliessen()` dreht die Karten wieder auf die Rückseite beziehungsweise nimmt sie komplett aus dem Spiel. Außerdem wechselt sie über die Methode `spielerWechseln()` den Spieler, wenn kein Paar gefunden wurde, und lässt den Computer noch einmal ziehen, wenn er an der Reihe war. Dazu rufen wir eine Methode `ComputerZug()` auf.

```
//die Methode dreht die Karten wieder auf die Rückseite
//beziehungsweise nimmt sie aus dem Spiel
private void KarteSchliessen()
{
    bool raus = false;
    //ist es ein Paar?
    if (paar[0].GetBildID() == paar[1].GetBildID())
        raus = true;
    //wenn es ein Paar war, nehmen wir die Karten aus
    //dem Spiel, sonst drehen wir sie nur wieder um
    paar[0].RueckseiteZeigen(raus);
    paar[1].RueckseiteZeigen(raus);
    //es ist keine Karte mehr geöffnet
    umgedrehteKarten = 0;
    //hat der Spieler kein Paar gefunden?
    if (raus == false)
        //dann wird der Spieler gewechselt
        SpielerWechseln();
```

```

else
    //hat der Computer ein Paar gefunden?
    //dann ist er noch einmal an der Reihe
    if (spieler == 1)
        ComputerZug();
}

```

Code 3.5: Die Methode KarteSchliessen()

Die Methode `SpielerWechseln()` schließlich wechselt den aktuellen Spieler und sorgt dafür, dass der Computer zieht, wenn er an der Reihe ist. Dazu rufen wir wieder die Methode `ComputerZug()` auf.

```

private void SpielerWechseln()
{
    //wenn der Mensch an der Reihe war, kommt jetzt der Computer
    if (spieler == 0)
    {
        spieler = 1;
        ComputerZug();
    }
    else
        spieler = 0;
}

```

Code 3.6: Die Methode SpielerWechseln()**Hinweis:**

Die einzelnen Methoden lassen sich ein wenig optimieren und zusammenfassen. Wir haben hier vor allem darauf geachtet, dass der Ablauf möglichst einfach und übersichtlich ist. Wenn Sie Lust haben, können Sie später selbst noch Verbesserungen vornehmen. So müssen Sie ja eigentlich nicht zweimal prüfen, ob ein Paar gefunden wurde.

Die Methode `ComputerZug()` lassen wir erst einmal komplett leer, da sie wieder etwas komplizierter ist. Für einen ersten Test übernehmen Sie bitte einfach nur den folgenden leeren Rumpf der Methode.

```

//die Methode setzt die Computerzüge um
//Sie ist erst einmal leer, damit der Compiler nicht mault
private void ComputerZug()
{
}

```

Code 3.7: Die leere Methode ComputerZug()

Leere Methoden werden auch **Dummy^{a)}-Methoden** genannt.

a) *Dummy* bedeutet übersetzt so viel wie „Attrappe“.



Alternativ können Sie den Aufruf der Methode `ComputerZug()` auch zunächst auskommentieren. Die Erfahrung zeigt aber, dass auskommentierte Aufrufe sehr schnell übersehen werden. Sie programmieren dann die Methode und wundern sich, warum nichts geschieht. Sehr viel eindeutiger ist das Anlegen einer Dummy-Methode. Hier sehen Sie sofort am nicht vorhandenen Quelltext, dass noch etwas fehlt.

Tipp:

Wenn Sie ganz sichergehen wollen, lassen Sie von der Dummy-Methode noch eine Ausgabe durchführen – zum Beispiel „Hier fehlt noch etwas“.

Das Programm sollte nun erkennen können, wenn zwei Karten umgedreht sind, und die Karten entweder wieder zurückdrehen oder aus dem Spiel nehmen. Auch das Erhöhen der Punkte sollte funktionieren – allerdings noch nicht ganz korrekt. Es kann durchaus sein, dass der Computer Paare gutgeschrieben bekommt, obwohl er gar nicht zieht. Wenn Sie das stört, können Sie Anweisungen zum Wechseln des Spielers zunächst noch auskommentieren.

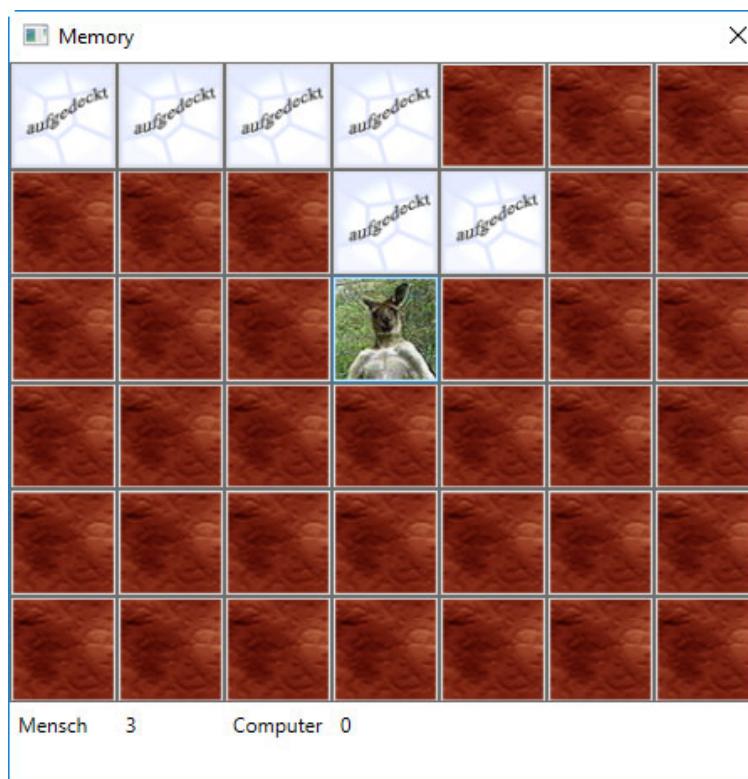


Abb. 3.5: Das Memory-Spiel mit einigen umgedrehten Paaren

3.4 Ein wenig Feinschliff

Bevor wir im nächsten Kapitel den Computergegner programmieren werden, wollen wir uns zunächst noch um zwei Sachen kümmern:

1. Die Karten sollen nicht mehr paarweise hintereinander auf dem Spielfeld abgelegt werden, sondern in zufälliger Reihenfolge erscheinen.
2. Vor dem Zurückdrehen sollen einen Moment lang die Vorderseiten der Karten angezeigt werden, damit sich der Spieler die Position einprägen kann. Im Moment wird vor allem die zweite Karte so schnell wieder zurückgedreht beziehungsweise aus dem Spiel genommen, dass sie kaum zu sehen ist.

Beginnen wir mit dem zufälligen Anordnen der Karten auf dem Spielfeld. Da es leider keine Methode zum zufälligen **Mischen von Werten in Arrays** gibt, müssen wir selber Hand anlegen. Dazu würfeln wird die Karten im Array `karten` zufällig durcheinander.

Erzeugen Sie bitte im ersten Schritt im Konstruktor der Klasse `MemoryFeld` eine neue Instanz der Klasse `Random`. Die entsprechende Anweisung könnte so aussehen:

```
Random zufallszahl = new Random();
```

Lassen Sie dann nach der Schleife für das Erstellen der Karten im Konstruktor die Karten zufällig mischen. Das erledigt zum Beispiel die folgende Konstruktion.

```
for (int i = 0; i <= 41; i++)  
{  
    int temp1;  
    MemoryKarte temp2;  
    //eine zufällige Zahl im Bereich 0 bis 41 erzeugen  
    temp1 = zufallszahl.Next(42);  
    //den alten Wert in Sicherheit bringen  
    temp2 = karten[temp1];  
    //die Werte tauschen  
    karten[temp1] = karten[i];  
    karten[i] = temp2;  
}
```

Code 3.8: Das Mischen der Karten im Array `karten`

Im ersten Schritt vereinbaren wir eine Variable `temp1` vom Typ `int` für den Index und eine Variable `temp2` vom Typ `MemoryKarte` zum Zwischenspeichern der Karte.

Danach erzeugen wir in der Schleife über die Anweisung

```
temp1 = zufallszahl.Next(42);
```

eine zufällige Zahl im Bereich zwischen 0 und 41.

Anschließend vertauschen wir im Array `karten` das Element mit diesem Index und das Element, das über den aktuellen Wert von `i` angesprochen wird. Damit die alte Karte dabei nicht verloren geht, speichern wir sie vor dem Überschreiben in der Variablen `temp2` zwischen.


Denken Sie bitte daran:

Wenn Sie nur ein Argument als obere Grenze angeben, liefert die Methode `Next()` der Klasse `Random` zufällige ganze Zahlen im Bereich 0 und obere Grenze – 1. Das Argument muss daher immer um den Wert 1 größer sein als die größte zufällige Zahl, die Sie benötigen.

Die Schleife mag zwar beim ersten Hinsehen ein wenig verzwickt wirken, ist aber eigentlich ganz einfach. Wir durchlaufen sämtliche Elemente im Array `karten` und erzeugen für jedes vorhandene Element eine zufällige Zahl im gültigen Bereich des Index. Dann wird der Inhalt des aktuellen Elements mit dem Inhalt des zufällig ermittelten Elements getauscht. Als Ergebnis erhalten wir nach dem Abarbeiten der Schleife eine zufällig gemischte Reihenfolge der Karten.

Damit nun auch tatsächlich die vermischten Karten in das Spielfeld gesetzt werden, müssen Sie die Anweisungen

```
karten[i].SetBildPos(i);
feld.Children.Add(karten[i]);
```

aus der ersten Schleife löschen und in einer dritten Schleife nach dem Mischen ausführen lassen.

Die gesamte Konstruktion für das Erstellen und Mischen der Karten im Konstruktor muss also so aussehen:

```
//das eigentliche Spielfeld erstellen
for (int i = 0; i <= 41; i++)
{
    //eine neue Karte erzeugen
    karten[i] = new MemoryKarte(bilder[count], count, this);
    //hier müssen die Anweisungen zum Setzen der Position und zum
    //Einfügen der Karte gelöscht werden
    //bei jeder zweiten Karte kommt auch ein neues Bild
    if ((i + 1) % 2 == 0)
        count++;
}

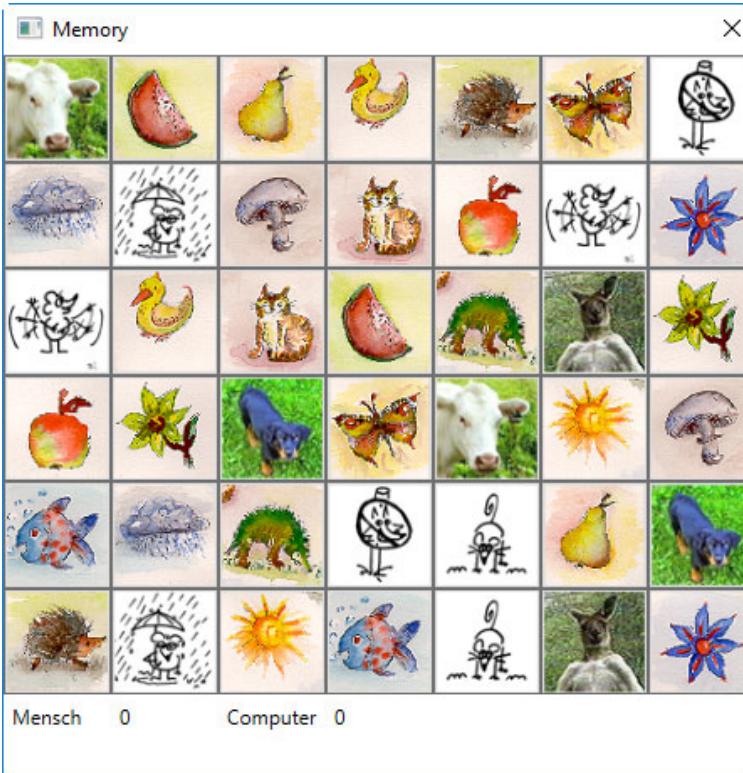
//die Karten mischen
for (int i = 0; i <= 41; i++)
{
    int temp1;
    MemoryKarte temp2;
    //eine zufällige Zahl im Bereich 0 bis 41 erzeugen
    temp1 = zufallszahl.Next(42);
    //den alten Wert in Sicherheit bringen
    temp2 = karten[temp1];
    //die Werte tauschen
    karten[temp1] = karten[i];
    karten[i] = temp2;
}

//die Karten ins Spielfeld bringen
for (int i = 0; i <= 41; i++)
```

```
{
    //die Position der Karte setzen
    karten[i].SetBildPos(i);
    //die Karte hinzufügen
    feld.Children.Add(karten[i]);
}
```

Code 3.9: Das Mischen und Zeichnen der Karten im Array karten

Übernehmen Sie die Erweiterungen jetzt bitte und testen Sie das Programm. Nun sollten die Karten bunt durcheinandergewürfelt werden. Das können Sie ganz einfach überprüfen, indem Sie im Konstruktor der Klasse `MemoryKarte` zunächst wieder die Vorderseite der Karten anzeigen lassen. Das Spielfeld könnte dann zum Beispiel so aussehen:

**Abb. 3.6:** Die gemischten Spielkarten

Kommen wir nun zu der Verzögerung beim Zurückdrehen der Karten. Dazu werden wir einen **Timer** benutzen. Ein ähnlich komfortables Steuerelement wie bei Windows Forms-Anwendungen gibt es bei WPF-Anwendungen leider nicht. Sie müssen den Timer komplett selbst erstellen. Dazu verwenden Sie die Klasse `System.Windows.Threading.DispatcherTimer`.

Aber auch wenn es kein fertiges Steuerelement gibt, ist der Einsatz eines Timers in einer WPF-Anwendung nicht allzu schwierig. Sie erzeugen eine neue Instanz der Klasse, setzen die Eigenschaft `Interval` für das Intervall und weisen der Eigenschaft `Tick` über einen Eventhandler die Methode zu, die vom Timer ausgeführt werden soll. Beim Intervall erwartet der Compiler eine Angabe vom Typ `TimeSpan`. Sie können sie sehr einfach zum Beispiel über die Methode `TimeSpan.FromMilliseconds()` erzeugen. Als Argument übergeben Sie an die Methode die gewünschte Zeit in Millisekunden.

Schauen wir uns das im praktischen Einsatz an. Das folgende Fragment vereinbart eine Instanz `timer`, setzt das Intervall und weist eine Methode zu. Abschließend wird der Timer über die Methode `Start()` gestartet.

```
//die Instanz erzeugen
//bitte in einer Zeile eingeben
System.Windows.Threading.DispatcherTimer timer = new System.
Windows.Threading.DispatcherTimer();
//das Intervall setzen
timer.Interval = TimeSpan.FromMilliseconds(2000);
//die Methode für das Ereignis zuweisen
timer.Tick += new EventHandler(Timer_Tick);
//den Timer starten
timer.Start();
```

Code 3.10: Einsatz eines Timers in der WPF (es handelt sich um ein Fragment)

Damit ist auch das Umdrehen der Karten in unserem Memory-Spiel über einen Timer nicht allzu aufwendig. Im ersten Schritt vereinbaren wir eine Instanz der Klasse `DispatcherTimer`, setzen das Intervall auf zwei Sekunden und weisen der Eigenschaft `Tick` eine Methode zu – zum Beispiel `Timer_Tick`. Danach starten Sie den Timer. Die entsprechenden Anweisungen finden Sie bereits im letzten Code. Sie können sie nahezu unverändert in den Konstruktor der Klasse `MemoryFeld` übernehmen. Sie müssen lediglich für die Instanz `timer` ein Feld verwenden, da wir auch in anderen Methoden auf den Timer zugreifen wollen.

Außerdem muss das Starten des Timers in der Methode `Karteoeffnen()` erfolgen. Ersetzen Sie hier die Anweisung

`KarteSchliessen();`

in der Abfrage

`if (umgedrehteKarten == 2)`

durch die Anweisung

`timer.Start();`

In der Methode `Timer_Tick()` für den Timer stoppen Sie den Timer direkt wieder und rufen dann die Methode `KarteSchliessen()` auf. Der Timer soll die Karten ja nur einmal umdrehen. Die vollständige Methode `Timer_Tick()` könnte so aussehen:

```
//die Methode für den Timer
private void Timer_Tick(object sender, EventArgs e)
{
    //den Timer anhalten
    timer.Stop();
    //die Karten zurückdrehen
    KarteSchliessen();
}
```

Code 3.11: Die Methode `Timer_Tick()`

Übernehmen Sie jetzt bitte die Erweiterungen für den Timer und testen Sie die Änderungen. Nun sollten die Karten vor dem Zurückdrehen zwei Sekunden lang angezeigt werden. Spielen Sie dann noch ein paar Runden Memory und genießen Sie das Gefühl des Siegers. Denn gleich werden wir einen sehr starken Computergegner programmieren, der kaum zu schlagen ist.

Hinweis:

Zur Sicherheit sollten Sie den Timer auch vor dem Herunterfahren der Anwendung beim Ende des Spiels ausdrücklich stoppen. Andernfalls kann es zu Schwierigkeiten kommen.

Zusammenfassung

Mit der Methode `Shutdown()` fahren Sie eine WPF-Anwendung herunter.

Leere Methoden werden auch Dummy-Methoden genannt.

Einen Timer für eine WPF-Anwendung erstellen Sie mit der Klasse `System.Windows.Threading.DispatcherTimer`.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie wollen die aktuelle WPF-Anwendung herunterfahren. Wie lautet die vollständige Anweisung?

- 3.2 Erstellen Sie einen Timer für eine WPF-Anwendung, der alle fünf Sekunden ausgelöst wird. Notieren Sie die dazu nötigen Anweisungen.

- 3.3 Verbinden Sie den Timer aus der letzten Aufgabe mit einer Methode `Timer_Anweisungen()`. Notieren Sie dazu wieder die Anweisungen.

4 Der Computer als Spielpartner

In diesem Kapitel werden wir dem Computer das Memory-Spielen beibringen und die Methode ComputerZug() ausprogrammieren.

Was sich zunächst vielleicht ein wenig kompliziert anhört, ist nicht schwer, wenn Sie herausfinden, wie die **Logik für die Computerzüge** umgesetzt werden muss.

4.1 Vorüberlegungen

Wir können davon ausgehen, dass der Computer die Position aller Karten, die im Spielverlauf bisher umgedreht wurden, kennt. Diese Positionen sind im Array `gemerkteKarten` gespeichert.

Da der menschliche Spieler immer den ersten Zug macht, können wir den Computer wie folgt ziehen lassen:

Er sucht im Array `gemerkteKarten` nach einem Paar. Dazu überprüft er, ob er in dem Array zweimal den gleichen Index einer Karte findet. Wenn der Computer ein Paar gefunden hat, dreht er die beiden Karten um.

Findet der Computer im Array `gemerkteKarten` kein Paar, dreht er zufällig zwei Karten um. Bei der ersten Karte müssen wir dabei lediglich überprüfen, ob sie noch im Spiel ist. Bei der zweiten Karte müssen wir außerdem noch sicherstellen, dass nicht zufällig die erste umgedrehte Karte noch einmal umgedreht wird.

Als Aktivitätsdiagramm lässt sich diese Logik so darstellen:

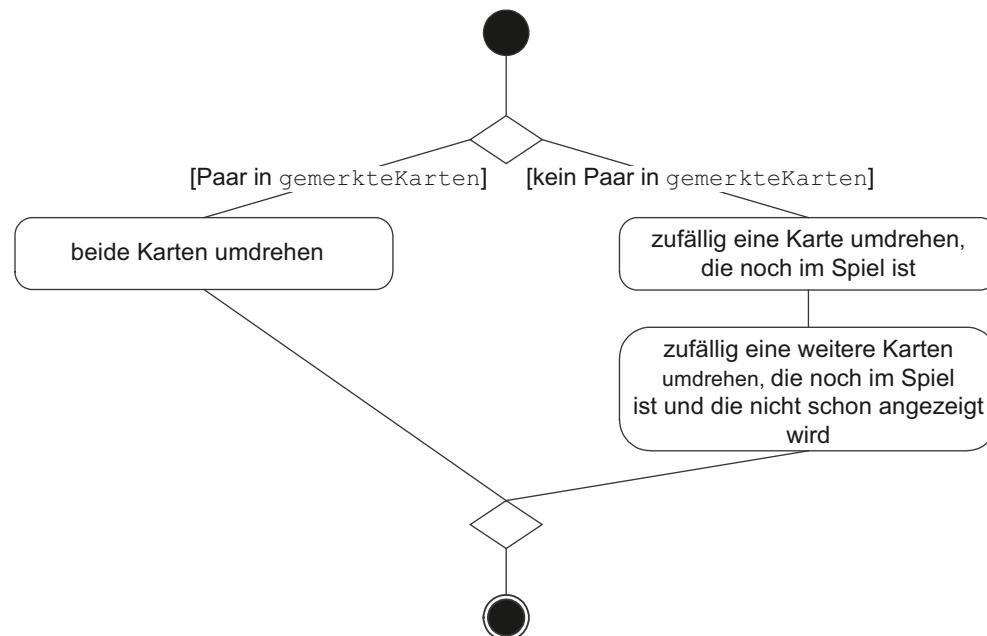


Abb. 4.1: Die Logik für die Computerzüge

Die Umsetzung ist nicht allzu schwierig:

Alle Elemente im Array `gemerkteKarten`, die nicht den Wert `-1` oder `-2` haben, stehen für eine Karte, die bereits umgedreht wurde und noch im Spiel ist. Wir müssen also nur das Array `gemerkteKarten` durchgehen, bis wir das erste Mal auf einen Wert ungleich `-1` oder `-2` treffen, und dann nach dem passenden Gegenstück suchen. Dieses Gegenstück muss sich im Array `gemerkteKarten` an der gleichen Position befinden, nur eben in der anderen Dimension.

Hinweis:

Die Ablage der Karten im Array `gemerkteKarten` haben wir in der Methode `Karteoeffnen()` programmiert. Sie erinnern sich? Wenn für eine neu aufgedeckte Karte das passende Gegenstück in `gemerkteKarten` gefunden wurde, haben wir die Position dieser Karte in der zweiten Dimension des Arrays abgespeichert.

Finden wir ein Paar, lesen wir die Positionen der beiden Karten aus dem Array `gemerkteKarten` aus und drehen die Karten um. Dazu simulieren wir mit der Methode `RaiseEvents()` der Klasse `Button` einen Klick auf die Spielkarten.

Finden wir kein Paar, lassen wir den Computer eine zufällige Zahl zwischen `0` und `41` ermitteln und drehen die Karte an dieser Position um. Dazu beschaffen wir uns über die Klasse `Random` und die Methode `Next()` eine zufällige Zahl im Zahlenraum zwischen `0` und `42`. Das Verfahren kennen Sie ja bereits.

4.2 Die praktische Umsetzung

Damit der Computer nicht versucht, bereits aufgedeckte oder aus dem Spiel genommene Karten umzudrehen, müssen wir vor dem Umdrehen der zufällig ermittelten Karten prüfen, ob die Karte noch im Spiel ist beziehungsweise im Moment nicht schon angezeigt wird. Dazu fragen wir den Wert der Felder `nochImSpiel` beziehungsweise `umgedreht` der jeweiligen Instanz der Klasse `MemoryKarte` ab. Da die beiden Felder privat vereinbart sind, müssen wir noch entsprechende Methoden erstellen, über die wir uns die Werte von außen beschaffen können. Diese Methoden könnten so aussehen:

```
//die Methode liefert den Wert des Felds umgedreht
public bool IsUmgredreht() {
    return umgedreht;
}

//die Methode liefert den Wert des Felds nochImSpiel
public bool IsNochImSpiel() {
    return nochImSpiel;
}
```

Code 4.1: Die Methoden `IsUmgredreht()` und `IsNochImSpiel()` der Klasse `MemoryKarte`

Das eigentliche Anzeigen der Karten und auch die Prüfungen, ob der Computer ein Paar gefunden hat, erfolgen beim simulierten Anklicken der Karten. Darum müssen wir uns also nicht weiter kümmern.

Die Methode `ComputerZug()` sieht dann so aus:

```
//die Methode setzt die Computerzüge um
private void ComputerZug()
{
    int kartenZaehler = 0;
    int zufall = 0;
    bool treffer = false;
    Random zufallszahl = new Random();
    //erst einmal nach einem Paar suchen
    //dazu durchsuchen wir das Array gemerkteKarten, bis wir
    //in beiden Dimensionen einen Wert für eine Karte finden

    while ((kartenZaehler < 21) && (treffer == false))
    {
        //gibt es in beiden Dimensionen einen Wert größer oder
        //gleich 0?
        //bitte in einer Zeile eingeben
        if ((gemerkteKarten[0, kartenZaehler] >=0) &&
            (gemerkteKarten[1, kartenZaehler] >=0)) {
            //dann haben wir ein Paar
            treffer = true;
            //die erste Karte umdrehen durch einen simulierten
            //Klick auf die Karte
            //bitte jeweils in einer Zeile eingeben
            karten[gemerkteKarten[0, kartenZaehler]].
            RaiseEvent(new RoutedEventArgs
            (ButtonBase.ClickEvent));
            //die zweite Karte auch
            karten[gemerkteKarten[1, kartenZaehler]].
            RaiseEvent(new RoutedEventArgs (ButtonBase.ClickEvent));
        }
        kartenZaehler++;
    }
    //wenn wir kein Paar gefunden haben, drehen wir zufällig
    //zwei Karten um
    if (treffer == false)
    {
        //so lange eine Zufallszahl suchen, bis eine Karte
        //gefunden wird, die noch im Spiel ist
        do
        {
            zufall = zufallszahl.Next(42);
        } while (karten[zufall].IsNochImSpiel() == false);
        //die erste Karte umdrehen
        //bitte in einer Zeile eingeben
        karten[zufall].RaiseEvent(new
        RoutedEventArgs(ButtonBase.ClickEvent));
        //für die zweite Karte müssen wir außerdem prüfen, ob
        //sie nicht gerade angezeigt wird
        do
        {
            zufall = zufallszahl.Next(42);
            //bitte in einer Zeile eingeben
        } while ((karten[zufall].IsNochImSpiel() == false) ||
            (karten[zufall].IsUmgedreht() == true));
    }
}
```

```
//und die zweite Karte umdrehen  
//bitte in einer Zeile eingeben  
karten[zufall].RaiseEvent(new  
RoutedEventArgs(ButtonBase.ClickEvent));  
}  
}
```

Code 4.2: Die Methode ComputerZug()

In der Schleife

```
while ((kartenZaehler < 21) && (treffer == false))
```

durchlaufen wir zunächst alle Elemente im Array `gemerkteKarten` und überprüfen mit der Anweisung

```
if ((gemerkteKarten[0, kartenZaehler] >=0) &&  
(gemerkteKarten[1, kartenZaehler] >=0))
```

ob wir ein Paar finden. Wenn ja, werden die beiden Karten durch einen simulierten Klick über die Methode `ButtonClick()` in der Klasse `MemoryKarte` umgedreht. Das erledigen die beiden Anweisungen

```
karten [gemerkteKarten [0, kartenZaehler] ].RaiseEvent (new RoutedEventArgs (ButtonBase.ClickEvent));
karten [gemerkteKarten [1, kartenZaehler] ].RaiseEvent (new RoutedEventArgs (ButtonBase.ClickEvent));
```

Wir rufen hier jeweils für die Karte an der entsprechenden Array-Position die Methode `RaiseEvent()` auf und übergeben dabei eine neue Instanz der Klasse `RoutedEventArgs`, die auf das Click-Ereignis für die Klasse `ButtonBase` – die Basisklasse für eine Schaltfläche – verweist. Die Zuordnung der Methode erfolgt dabei über die Instanz der Klasse `Button`, für die wir die Methode `RaiseEvent()` aufrufen. Die Methode selber, die ausgeführt werden soll, müssen Sie daher nicht angeben. Es wird immer die Methode genutzt, die dem Click-Ereignis zugeordnet ist.



Bitte beachten Sie:

Wenn wir ein Paar finden, wird die weitere Suche abgebrochen. Die gesamte weitere Steuerung des Spiels übernehmen dann die Methode `Karteoeffnen()`, die durch das Anklicken aufgerufen wird, und die Methoden, die von der Methode `Karteoeffnen()` aufgerufen werden.

Wenn kein Paar gefunden wurde, drehen wir zufällig zwei Karten um. Das übernehmen die Anweisungen

```
do
{
    zufall = zufallszahl.Next(42);
} while (karten[zufall].IsNochImSpiel() == false);
karten[zufall].RaiseEvent(new
RoutedEventArgs(ButtonBase.ClickEvent));
do
{
    zufall = zufallszahl.Next(42);
} while ((karten[zufall].IsNochImSpiel() == false) ||
(karten[zufall].IsUmgedreht() == true));
```

```
karten[zufall].RaiseEvent(new  
RoutedEventArgs(ButtonBase.ClickEvent));
```

am Ende der Methode `ComputerZug()`. Auch hier erfolgt die weitere Verarbeitung durch die Methode `Karteoeffnen()`.

Übernehmen Sie jetzt die Erweiterungen aus den vorigen Codes und spielen Sie dann einmal eine Runde gegen den Computer. Wenn Sie nicht über ein ausgezeichnetes Gedächtnis verfügen oder sich Notizen zu den Positionen der Karten machen, werden Sie allerdings eine unangenehme Überraschung erleben: Wir haben dem Computer so gut Memory-Spielen beigebracht, dass er kaum zu besiegen ist.

4.3 Das Einstellen der Spielstärke

Durch einen einfachen Kniff können wir die **Spielstärke** aber beliebig anpassen: Wir lassen in der Methode `ComputerZug()` noch eine zufällige Zahl ermitteln und greifen nur dann auf das Gedächtnis des Computers zu, wenn diese Zahl 0 ist. Je größer der Zahlenbereich ist, den wir für die Ermittlung der zufälligen Zahl benutzen, desto geringer ist die Wahrscheinlichkeit, dass die zufällig ermittelte Zahl 0 ist und desto schwächer spielt der Computer.

Vereinbaren Sie jetzt bitte ein weiteres Feld vom Typ `int` für die Klasse `MemoryFeld`. Wir benutzen in unserem Beispiel den Bezeichner `spielstaerke`. Setzen Sie dieses Feld im Konstruktor zum Beispiel auf den Wert 10 und erweitern Sie dann die Methode `ComputerZug()` um eine `if`-Abfrage, die den Zugriff auf das Gedächtnis des Computers steuert. Die entsprechenden Erweiterungen sind im folgenden Code fett markiert:

```
private void ComputerZug() {  
  
    ...  
  
    //zur Steuerung über die Spielstärke  
    if (zufallszahl.Next(spielstaerke) == 0)  
    {  
        //erst einmal nach einem Paar suchen  
        //dazu durchsuchen wir das Array gemerkteKarten, bis wir in  
        //beiden Dimensionen einen Wert für eine Karte finden  
        while ((kartenZaehler < 21) && (treffer == false))  
        {  
            ...  
            kartenZaehler++;  
        }  
    }  
  
    //wenn wir kein Paar gefunden haben, drehen wir zufällig  
    //zwei Karten um  
    ...  
}
```

Code 4.3: Die erweiterte Methode `ComputerZug()`
(die Methode ist nicht vollständig abgedruckt)

Tipp:

Wenn Sie Lust haben, können Sie das Memory-Spiel ja so erweitern, dass der Anwender die Spielstärke selbst über ein Steuerelement einstellen kann.

4.4 Der letzte Feinschliff

Nun gibt es noch ein Problem beim Memory-Spiel, das aber nur beim intensiven Testen auffällt. Sie können nämlich als Spieler auch sehr viel mehr als zwei Karten in einem Spielzug umdrehen und auch dann auf das Spielfeld klicken, wenn eigentlich der Computer am Zug ist. In beiden Fällen kommt unser Spiel durcheinander und löst möglicherweise eine Ausnahme aus.

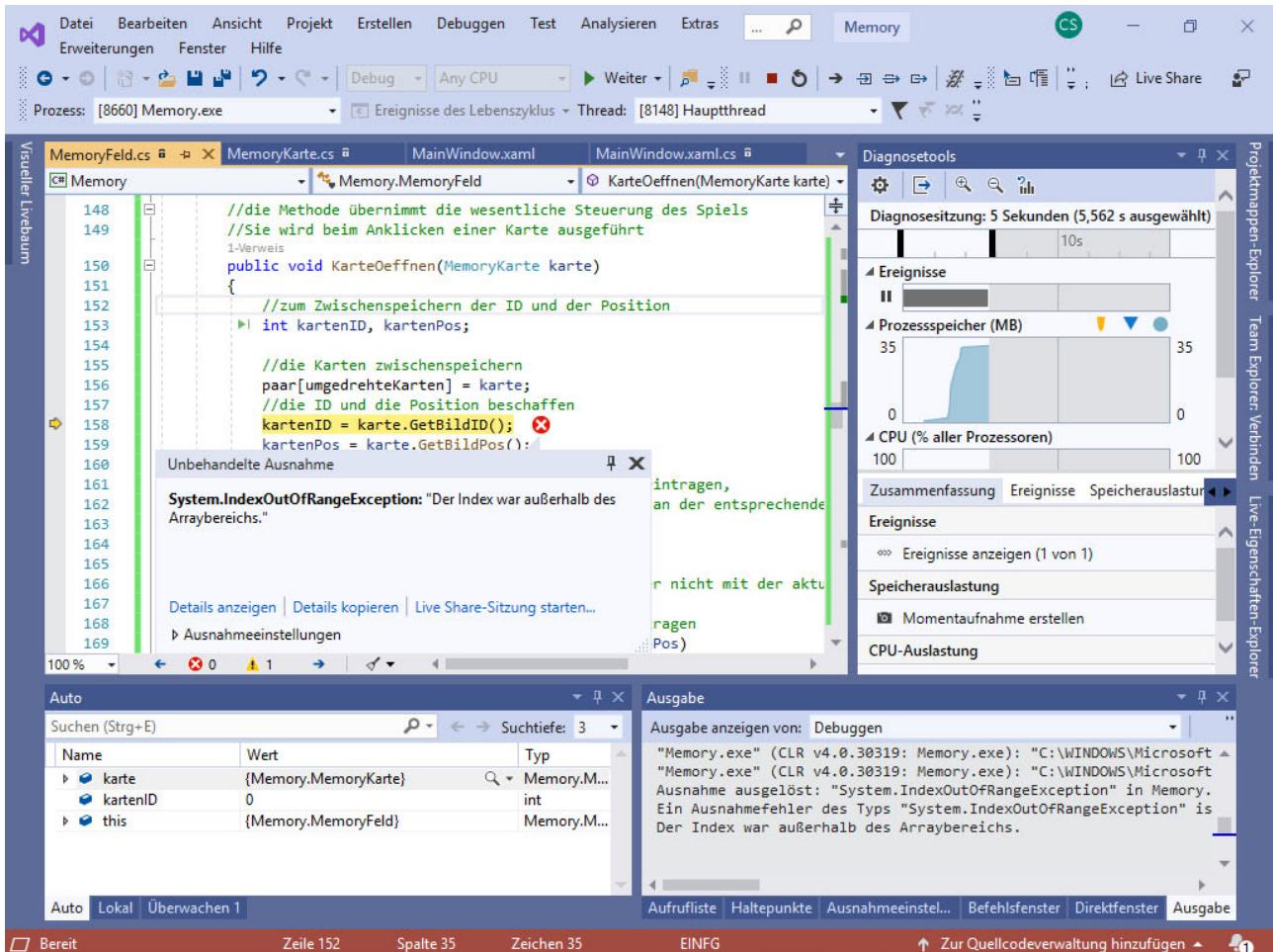


Abb. 4.2: Mehrere umgedrehte Karten führen zu Problemen

Wir müssen also zum einen dafür sorgen, dass durch den menschlichen Spieler nicht mehr als zwei Karten umgedreht werden können, zum anderen aber auch sicherstellen, dass der Mausklick auf eine Karte nur dann Wirkung zeigt, wenn nicht gerade der Computer zieht.

Dazu erstellen wir im ersten Schritt eine Methode `ZugErlaubt()` in der Klasse `MemoryFeld`, die `true` liefert, wenn der Mensch ziehen darf, und `false`, wenn der Mensch nicht ziehen darf. Diese Methode könnte so aussehen:

```
//die Methode liefert, ob Züge des Menschen erlaubt sind
//die Rückgabe ist false, wenn gerade der Computer zieht oder
//wenn schon zwei Karten umgedreht sind
//sonst ist die Rückgabe true
public bool ZugErlaubt() {
    bool erlaubt = true;
    //zieht der Computer?
    if (spieler == 1)
        erlaubt = false;
    //sind schon zwei Karten umgedreht?
    if (umgedrehteKarten == 2)
        erlaubt = false;
    return erlaubt;
}
```

Code 4.4: Die Methode `ZugErlaubt()` für die Klasse `MemoryFeld`

In der Methode `ButtonClick()` der Klasse `MemoryKarte` erweitern wir die erste Abfrage und überprüfen zusätzlich, ob der Mensch ziehen darf.

```
//die Methode für das Anklicken
private void ButtonClick(object sender, RoutedEventArgs e)
{
    //ist die Karte überhaupt noch im Spiel?
    //und sind Züge erlaubt?
    if ((nochImSpiel == false) || (spiel.ZugErlaubt() == false))
        return;
    //wenn die Rückseite zu sehen ist, die Vorderseite anzeigen
    ...
}
```

Code 4.5: Die Erweiterung in der Methode `ButtonClick()` der Klasse `MemoryKarte` (die Klasse ist nicht vollständig abgedruckt)

Wenn Sie aufmerksam mitgedacht haben, sollten Sie jetzt stutzig werden. Denn wir lassen den Computer ja durch einen simulierten Klick auf die Schaltflächen ziehen – und verhindern genau das jetzt durch die Abfrage, ob der Computer gerade spielt. Im letzten Schritt müssen wir also noch eine eigene Methode in der Klasse `MemoryKarte` erstellen, die die Vorderseite einer Karte ohne einen Klick auf die entsprechende Schaltfläche anzeigen kann. Diese Methode benutzen wir dann gemeinsam mit der Methode `Karteoeffnen()` für das Umdrehen der Karten beim Computerspielzug.

Die Methode `VorderseiteZeigen()` der Klasse `MemoryKarte` ist nicht allzu schwierig, da sie nur bereits bekannte Sachen noch einmal umsetzt.

```
//die Methode zeigt die Vorderseite der Karte an
public void VorderseiteZeigen()
{
    Content = bildVorne;
    umgedreht = true;
}
```

Code 4.6: Die Methode VorderseiteZeigen() der Klasse MemoryKarte

Hinweis:

Sie sollten die beiden Anweisungen zum Anzeigen der Vorderseite in der Methode `ButtonClick()` der Klasse `MemoryKarte` durch den Aufruf der Methode `VorderseiteZeigen()` ersetzen. Dann müssen Sie Änderungen gegebenenfalls nur an einer Stelle im Quelltext vornehmen.

Ersetzen Sie dann noch in der Klasse `MemoryFeld` in der Methode `ComputerZug()` den Aufruf der Methode `RaiseEvent()` für die Karten jeweils durch die Aufrufe der Methoden `VorderseiteZeigen()` und `Karteoeffnen()`. Die entsprechenden Stellen sind im folgenden Code noch einmal fett markiert dargestellt. Bitte achten Sie bei den Änderungen genau darauf, dass Sie auch die richtige Karte anzeigen und umdrehen lassen.

```
//die Methode setzt die Computerzüge um
private void ComputerZug()

{
    ...
    while ((kartenZeahler < 21) && (treffer == false))
    {
        //gibt es in beiden Dimensionen einen Wert größer
        //oder gleich 0?
        if ((gemerkteKarten[0, kartenZeahler] >= 0) &&
            (gemerkteKarten[1, kartenZeahler] >= 0))
        {
            //dann haben wir ein Paar
            treffer = true;
            //die erste Karte umdrehen durch einen simulierten
            //Klick auf die Karte
            //der simulierte Klick wird nicht mehr ausgeführt
            //karten[gemerkteKarten[0,
            //kartenZeahler]].RaiseEvent(new
            //RoutedEventArgs(ButtonBase.ClickEvent));
            //die Vorderseite zeigen
            //bitte jeweils in einer Zeile eingeben
            karten[gemerkteKarten[0,
            kartenZeahler]].VorderseiteZeigen();
            //und die Karte öffnen
            Karteoeffnen(karten[gemerkteKarten[0,kartenZeahler]]);
            //die zweite Karte auch
            //karten[gemerkteKarten[1,
            //kartenZeahler]].RaiseEvent(new
            //RoutedEventArgs(ButtonBase.ClickEvent));
```

```

        //die Vorderseite zeigen
        karten[gemerkteKarten[1,
        kartenZaehler]].VorderseiteZeigen();
        //und die Karte öffnen
        Karteoeffnen(karten[gemerkteKarten[1,kartenZaehler]]));
    }
    kartenZaehler++;
}
}
//wenn wir kein Paar gefunden haben, drehen wir zufällig
//zwei Karten um
if (treffer == false)
{
    //solange eine Zufallszahl suchen, bis eine Karte
    //gefunden wird, die noch im Spiel ist
    do
    {
        zufall = zufallszahl.Next(42);
    } while (karten[zufall].IsNochImSpiel() == false);
    //die erste Karte umdrehen
    //karten[zufall].RaiseEvent(new
    //RoutedEventArgs(ButtonBase.ClickEvent));
    //die Vorderseite zeigen
    karten[zufall].VorderseiteZeigen();
    //und die Karte öffnen
    Karteoeffnen(karten[zufall]);
    //für die zweite Karte müssen wir außerdem prüfen, ob
    //sie nicht gerade angezeigt wird
    do
    {
        zufall = zufallszahl.Next(42);
    } while ((karten[zufall].IsNochImSpiel() == false)
    || (karten[zufall].IsUmgedreht() == true));
    //und die zweite Karte umdrehen
    //karten[zufall].RaiseEvent(new
    //RoutedEventArgs(ButtonBase.ClickEvent));
    //die Vorderseite zeigen
    karten[zufall].VorderseiteZeigen();
    //und die Karte öffnen
    Karteoeffnen(karten[zufall]);
}
}

```

Code 4.7: Die geänderte Methode ComputerZug()
(die Methode ist nicht vollständig abgedruckt)

Übernehmen Sie jetzt auch noch diese letzten Änderungen und führen Sie einen abschließenden Test durch.

Zusammenfassung

Sie können Ereignisse für Steuerelemente im Quelltext simulieren lassen.

Die Zuordnung erfolgt dabei zu der Methode, die für das Ereignis hinterlegt ist.

Aufgaben zur Selbstüberprüfung

- 4.1 Formulieren Sie eine Anweisung, mit der ein Klick auf eine Schaltfläche buttonTest simuliert wird.

- 4.2 Sie wollen, dass Anweisungen mit einer Wahrscheinlichkeit von 10 Prozent ausgeführt werden. Wie können Sie dazu vorgehen?

Schlussbetrachtung

Sie haben es geschafft! Sie haben ein Memory-Spiel programmiert und dabei einen sehr spielstarken Gegner programmiert.

Sie wissen jetzt, wie Sie Steuerelemente in Tabellenform anzeigen lassen und wie Sie zum Beispiel den Klick auf ein Steuerelement simulieren. Außerdem können Sie Daten in Arrays mischen.

Die gesamte Logik des Spiels haben Sie selbst programmiert. Dabei haben wir sehr viele Techniken eingesetzt, die Sie bereits kannten – zum Beispiel das Verarbeiten von Arrays, Schleifen und `if`-Abfragen. Wie Sie gesehen haben, besteht die eigentliche Kunst beim Programmieren gar nicht so sehr darin, einen Quelltext zu schreiben. Viel wichtiger ist es, die Aufgabenstellung auf die Arbeitsweise des Computers abzubilden. Dabei kann Ihnen leider keine noch so komfortable Entwicklungsumgebung helfen.

Bevor Sie also anfangen, eine Lösung für ein Problem zu programmieren, setzen Sie sich erst einmal ins „stille Kämmerlein“ und überlegen Sie, wie Sie das Problem angehen müssen. Das Umsetzen der Lösung in konkrete Anweisungen ist dann relativ schnell erledigt. Oder hätten Sie vor dem Bearbeiten dieses Studienhefts gedacht, dass ein paar Anweisungen ausreichen, um dem Computer Memory beizubringen?

Nutzen Sie auch das Projekt aus diesem Studienheft für eigene Erweiterungen. Ein paar Vorschläge:

- Sie können das Aufdecken der Karten mit Sound-Effekten versehen.
- Bieten Sie dem Spieler die Möglichkeit, Optionen selbst zu setzen – zum Beispiel die Dauer, bis die Karten wieder umgedreht werden oder die Spielstärke des Computers. Dazu benötigen Sie ein Fenster, in dem der Spieler die entsprechenden Daten eingeben kann. Statt fester Werte verwenden Sie dann in den Methoden des Spiels die Daten, die der Spieler bei den Optionen eingegeben hat.

Zwei kleinere Erweiterungen – die Anzeige des Gewinners und eine Schummelfunktion – warten als Einsendeaufgabe auf Sie.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 2

- 2.1 Die Anweisungen könnten so aussehen:

```
bild = new Image();
bild.Source = new BitmapImage(new Uri("testbild.jpg",
UriKind.Relative));
buttonTest.Content = bild;
```

Der Bezeichner für die Instanz der Klasse `Image` kann auch abweichen.

- 2.2 Die Anweisung könnte so aussehen:

```
conTest.Children.Add(labelTest);
```

Kapitel 3

- 3.1 Die Anweisung lautet:

```
Application.Current.Shutdown();
```

- 3.2 Die Anweisungen könnten so aussehen:

```
System.Windows.Threading.DispatcherTimer timer = new
System.Windows.Threading.DispatcherTimer();
timer.Interval = TimeSpan.FromMilliseconds(5000);
```

Der Bezeichner für den Timer kann auch abweichen.

- 3.3 Die Anweisung muss so aussehen:

```
timer.Tick += new EventHandler(Timer_Anweisungen);
```

Der Bezeichner hängt von dem Namen ab, den Sie in der vorigen Aufgabe verwendet haben.

Kapitel 4

- 4.1 Die Anweisung könnte so aussehen:

```
buttonTest.RaiseEvent(new
RoutedEventArgs(ButtonBase.ClickEvent));
```

- 4.2 Sie erstellen eine `if`-Abfrage, die prüft, ob sich eine zufällig ermittelte Zahl im Bereich von 1 bis 100 glatt durch 10 teilen lässt. Wenn das der Fall ist, lassen Sie die Anweisungen ausführen.

B. Glossar

Container	Ein Container nimmt weitere Steuerelemente in einer WPF-Anwendung auf.
Dummy-Methode	Eine <i>Dummy</i> -Methode ist eine leere Methode. <i>Dummy</i> -Methoden werden oft für den Test eingesetzt, wenn die entsprechende Funktionalität noch nicht programmiert wurde.
Event handling	<i>Event handling</i> ist die englische Bezeichnung für Ereignisverarbeitung.
Eventhandler	Der <i>Eventhandler</i> stellt eine Verbindung zwischen einem Ereignis und einer Methode her, die beim Eintreten des Ereignisses ausgeführt werden soll.
Registrierung (Eventhandler)	Bei der Registrierung eines Eventhandlers wird ein Ereignis mit einer Methode verbunden.
Timer	Ein <i>Timer</i> löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis aus, in dem Sie Anweisungen verarbeiten lassen können. Ein <i>Timer</i> ermöglicht damit die zeitgesteuerte Verarbeitung von Anweisungen.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 6. Aufl., Bonn: Rheinwerk.

Speziell mit dem Thema WPF beschäftigt sich das folgende Buch:

Huber, T.C. (2019). *Windows Presentation Foundation. Das umfassende Handbuch*. 5. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Das Memory-Spiel	4
Abb. 2.1	Das Fenster	5
Abb. 2.2	Der Container im Fenster	6
Abb. 2.3	Die erste Version des Memory-Spiels mit allen umgedrehten Karten ...	16
Abb. 3.1	Der vereinfachte Spielablauf	19
Abb. 3.2	Die erste eingetragene Karte im Gedächtnis (ganz links oben)	24
Abb. 3.3	Die zweite eingetragene Karte im Gedächtnis	24
Abb. 3.4	Die dritte eingetragene Karte im Gedächtnis	24
Abb. 3.5	Das Memory-Spiel mit einigen umgedrehten Paaren	28
Abb. 3.6	Die gemischten Spielkarten	31
Abb. 4.1	Die Logik für die Computerzüge	34
Abb. 4.2	Mehrere umgedrehte Karten führen zu Problemen	39
Abb. G.1	Die Schummelfunktion	52

E. Codeverzeichnis

Code 2.1	Die Klasse MemoryKarte	9
Code 2.2	Eine erste Version der Klasse für die Anwendung	13
Code 3.1	Die Änderungen an der Klasse MemoryKarte (der Code ist nicht vollständig)	21
Code 3.2	Die Methode KarteOeffnen()	22
Code 3.3	Die Methode PaarPruefen().....	25
Code 3.4	Die Methode PaarGefunden()	26
Code 3.5	Die Methode KarteSchliessen()	27
Code 3.6	Die Methode SpielerWechseln()	27
Code 3.7	Die leere Methode ComputerZug()	27
Code 3.8	Das Mischen der Karten im Array karten	29
Code 3.9	Das Mischen und Zeichnen der Karten im Array karten	31
Code 3.10	Einsatz eines Timers in der WPF (es handelt sich um ein Fragment) ...	32
Code 3.11	Die Methode Timer_Tick()	32
Code 4.1	Die Methoden IsUmgedreht() und IsNochImSpiel() der Klasse MemoryKarte	35
Code 4.2	Die Methode ComputerZug()	37
Code 4.3	Die erweiterte Methode ComputerZug()	38
Code 4.4	Die Methode ZugErlaubt() für die Klasse MemoryFeld	40
Code 4.5	Die Erweiterung in der Methode ButtonClick() der Klasse MemoryKarte	40
Code 4.6	Die Methode VorderseiteZeigen() der Klasse MemoryKarte	41
Code 4.7	Die geänderte Methode ComputerZug()	42

F. Sachwortverzeichnis

C		W	
Computerzug		Werte	
Logik für den	34	in Arrays mischen	29
D			
Dummy-Methode	27		
E			
Eigenschaft			
SizeMode	5		
Eventhandler			
für das Klicken	10		
G			
Grafik			
für eine Schaltfläche laden	9		
K			
Klasse			
für die Memory-Karten	7		
MemoryFeld.....	11		
M			
Methode			
Karteoeffnen()	20		
S			
Spielablauf			
Vorüberlegungen zur Steuerung des	18		
Spielregeln	3		
Spielstärke			
anpassen	38		
T			
Timer	31		
U			
UniformGrid	6		
V			
Vorgaben	3		

G. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP17D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Schicken Sie für die Lösungen bitte das vollständige Projekt mit allen Unterordnern und Dateien ein. Markieren Sie im Quelltext deutlich, zu welcher Aufgabe die jeweiligen Anweisungen gehören – zum Beispiel durch Kommentare.

Beschreiben Sie bei allen Einsendeaufgaben für dieses Studienheft zusätzlich, welche grundsätzlichen Schritte für die Erweiterungen erforderlich sind – also zum Beispiel, welche Steuerelemente Sie einfügen und wie Sie die Eigenschaften dieser Elemente setzen.

Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

1. Erweitern Sie die Meldung am Ende des Memory-Spiels um eine Ausgabe des Spielergebnisses. Lassen Sie sich dazu in dem Dialog, der am Spielende erscheint, noch einmal ausdrücklich anzeigen, wie viele Paare der Gewinner gezogen hat.

20 Pkt.

2. Bauen Sie eine Schummelfunktion in das Memory-Spiel ein. Ergänzen Sie dazu eine Schaltfläche **Schummel**. Beim Anklicken der Schaltfläche sollen alle noch nicht aufgedeckten Karten für eine bestimmte Zeit angezeigt und danach automatisch wieder umgedreht werden.

Die Schaltfläche können Sie wie die Labels im Spiel dynamisch erzeugen und in das Spielfeld setzen.

Das Ergebnis könnte ungefähr so aussehen:

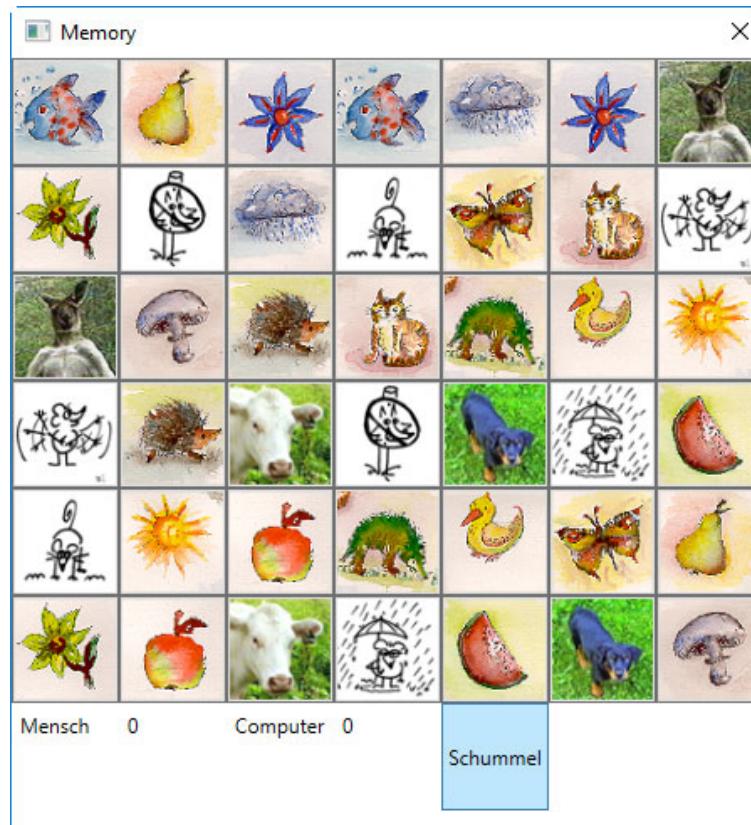


Abb. G.1: Die Schummelfunktion

Sorgen Sie dafür, dass die Schummelfunktion nur dann aufgerufen werden kann, wenn gerade der Mensch am Zug ist und wenn keine andere Karte umgedreht ist. Dazu können Sie zum Beispiel die Schaltfläche zum Starten der Schummelfunktion je nach Spielzustand aktivieren beziehungsweise deaktivieren.

Die Lösung ist nicht allzu schwierig, wenn Sie den richtigen Dreh finden.

80 Pkt.

Gesamt 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Snake – ein Computerspiel

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1119N01

CSHP18D

Objektorientierte Software-Entwicklung mit C#

Snake – ein Computerspiel

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Snake – ein Computerspiel

Inhaltsverzeichnis

Einleitung	1
1 Spielregeln und Vorüberlegungen	3
2 Spielfeld erstellen	5
2.1 Die Menüleiste	5
2.2 Das Spielfeld	6
2.3 Die Schlange	9
2.4 Das Bewegen der Schlange	16
2.5 Die Äpfel	19
2.6 Die Kollisionsabfrage	22
2.7 Die Anzeige der Spielzeit	30
Zusammenfassung	31
3 Spielsteuerung und Bestenliste	34
3.1 Vorüberlegungen	34
3.2 Die Spielsteuerung	35
3.3 Die Bestenliste	41
Zusammenfassung	48
4 Ein wenig Feinschliff	49
4.1 Automatische Beschleunigung der Schlange	49
4.2 Tastatursteuerung für die Pausenfunktion	50
4.3 Ein Startbildschirm	53
Zusammenfassung	54
Schlussbetrachtung	56
Anhang	
A. Lösungen der Aufgaben zur Selbstüberprüfung	57
B. Glossar	59
C. Literaturverzeichnis	61
D. Abbildungsverzeichnis	62
E. Tabellenverzeichnis	63
F. Codeverzeichnis	64
G. Sachwortverzeichnis	66
H. Einsendeaufgabe	67

Einleitung

In diesem Studienheft werden wir einen weiteren Computerspielklassiker programmieren – und zwar das Spiel „Snake“. Bei diesem Spiel muss der Spieler eine Schlange über den Bildschirm bewegen und so viele Äpfel wie möglich „fressen“ – ohne mit einem Hindernis zu kollidieren.

Auch dieses Spiel werden wir wie gewohnt Schritt für Schritt umsetzen und erweitern.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie die aktuelle Breite und Höhe eines Steuerelements ermitteln,
- was ein Generic ist,
- wie Sie die Klasse `List` des .NET Frameworks einsetzen,
- wie Sie Elemente an eine Liste vom Typ `List` anhängen,
- wie Sie eine Liste vom Typ `List` leeren,
- wie Sie die Schlange und die Schlangenteile bewegen,
- wie Sie die Schlange über die Tastatur steuern,
- wie Sie den Zufallsgenerator für die Klasse `Random` initialisieren,
- was ein Hashcode ist,
- wie Sie den Hashcode einer Instanz ermitteln,
- wie Sie eine Kollisionsabfrage zwischen zwei Objekten mit der WPF programmieren,
- wie Sie ermitteln können, mit welchem Objekt ein anderes Objekt kollidiert ist,
- wie Sie untergeordnete Elemente aus einer grafischen Oberfläche entfernen,
- wie Sie bei der WPF die Bildschirmauflösung ermitteln,
- wie Sie bei der WPF den Ordner einer Anwendung ermitteln,
- wie Sie die Daten der Bestenliste in einem Grid anzeigen lassen,
- wie Sie aus einer Code-Behind-Datei neue Zeilen an ein Grid anhängen,
- wie Sie eigene Commands erstellen und
- wie Sie einen Startbildschirm für eine WPF-Anwendung erstellen.

Beginnen werden wir aber zunächst einmal mit den Spielregeln und einigen Vorüberlegungen zur Umsetzung.

Christoph Siebeck

1 Spielregeln und Vorüberlegungen

In dieser Lektion stellen wir Ihnen die Spielregeln vor und stellen einige Überlegungen zur praktischen Umsetzung an.

Beginnen wir mit den Spielregeln.

- Das Spielfeld besteht aus Balken an den vier Seiten, einer Schlange und Äpfeln. Die Schlange bewegt sich über das Spielfeld und kann vom Spieler gesteuert werden. Die Äpfel werden zufällig auf dem Spielfeld verteilt. Es wird allerdings immer nur ein Apfel gleichzeitig angezeigt.
- Der Spieler muss versuchen, mit der Schlange möglichst viele Äpfel zu „fressen“. Dazu muss er einen Apfel mit dem Kopf der Schlange berühren.
- Wenn ein Apfel „gefressen“ wurde, erhält der Spieler Punkte. Außerdem verlängert sich die Schlange und ein neuer Apfel wird an einer zufälligen Position angezeigt.
- Die Schlange darf kein Hindernis berühren. Als Hindernis gelten dabei die Spielfeldränder und auch die Schlange selbst.
- Die Spielzeit ist nicht begrenzt. Das Spiel wird beendet, wenn die Schlange ein Hindernis berührt.
- Der Spieler kann zwischen verschiedenen Schwierigkeitsgraden wählen, die die Größe des Spielfelds verändern. Je kleiner das Spielfeld ist, desto mehr Punkte gibt es für jeden „gefressenen“ Apfel.
- Die Punkte speichern wir in einer Bestenliste. Dabei setzen wir die Klasse `Score` ein, die wir bereits beim Pong-Spiel verwendet haben. Damit wir die Klasse auch bei einer WPF-Anwendung einsetzen können, müssen wir allerdings ein paar kleine Anpassungen vornehmen.

Wir werden für das Spiel mehrere Klassen anlegen, die das Spielfeld, die Logik und die verschiedenen Spielobjekte abbilden. Die einzelnen Klassen erstellen wir dabei wieder Schritt für Schritt.

Die Schlange bauen wir aus einzelnen Teilen zusammen, die wir in einer Liste verwalten. Dabei benutzen wir aber kein Array, sondern die Klasse `List`. Sie erlaubt beliebig lange Listenkonstruktionen. Was sich genau hinter diesem Typ verbirgt, erfahren Sie später.

Das Spielfeld soll zunächst eine feste Größe von $800 * 600$ Punkten haben. Später werden wir die Größe aber von den Einstellungen des Schwierigkeitsgrads abhängig machen. Je höher der Schwierigkeitsgrad, desto kleiner soll das Spielfeld sein.

Die Bewegungen der Schlange steuern wir über die Cursortasten auf der Tastatur. Damit können wir die vier Richtungen ja sehr gut abbilden.

Für die Schlange benutzen wir Rechtecke zur Darstellung. Der Schlangenkopf soll rot sein und die Teile des Schlangenkörpers schwarz. Die Äpfel bilden wir durch grüne Ellipsen ab.

Für die Funktionen zur Steuerung des Spiels und zum Setzen der Einstellungen stellen wir dem Spieler eine Menüleiste zur Verfügung.

Das fertige Spiel sollte ungefähr so aussehen:

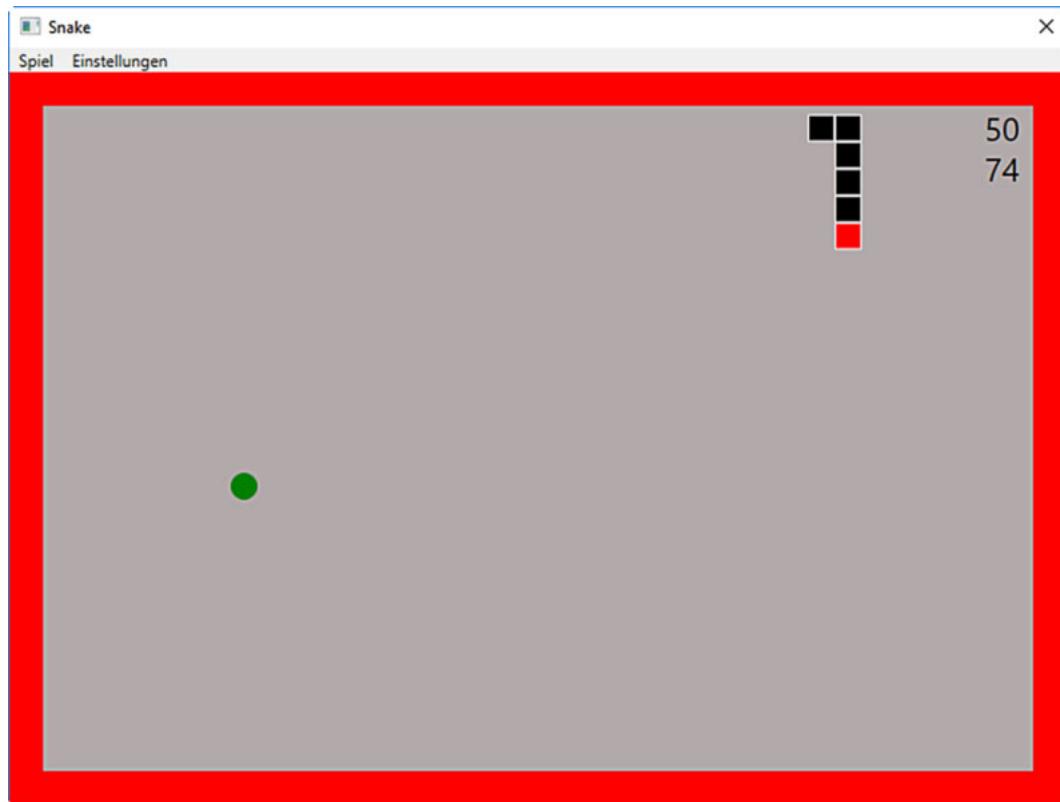


Abb. 1.1: Das Snake-Spiel

Beginnen wir jetzt mit dem Erstellen des Spielfelds.

2 Spielfeld erstellen

In diesem Kapitel erstellen wir das Spielfeld.

Legen Sie eine neue leere WPF-Anwendung für unser Spiel an. Damit keine Änderungen der Fenstergröße möglich sind, ändern Sie die Eigenschaft **ResizeMode** in `NoResize`. Setzen Sie die Fensterbreite auf `800` und die Fensterhöhe auf `600`. Ändern Sie dann die Eigenschaft **Title** für das Fenster in `Snake`. Damit das Fenster automatisch beim Start auf dem Bildschirm zentriert wird, setzen Sie außerdem die Eigenschaft **WindowsStartupLocation**¹ in der Gruppe **Allgemein** auf `CenterScreen`. Stellen Sie noch sicher, dass das Fenster unserer Anwendung immer im Vordergrund bleibt. Aktivieren Sie dazu die Eigenschaft **Topmost** des Fensters. Sie finden sie ebenfalls in der Gruppe **Allgemein**.

Im Grid des Fensters legen wir zwei Zeilen an. Die obere Zeile soll gleich die Menüleiste aufnehmen und die untere das Spielfeld. Die Höhe der oberen Zeile setzen wir auf `Auto`. Die untere Zeile soll den Rest einnehmen. Die entsprechenden Vereinbarungen finden Sie im folgenden Code 2.1:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

Code 2.1: Die Zeilen für das Grid

2.1 Die Menüleiste

Fügen wir jetzt eine Menüleiste ein. Das erfolgt wie gewohnt über ein Steuerelement vom Typ **Menu**.

Hinweis:

Da wir ja bereits mehrfach Menüs erstellt haben, zeigen wir Ihnen hier nur den Code und stellen Ihnen die weiteren Schritte kurz vor.

Das erste grobe Gerüst für unsere Menüleiste finden Sie im folgenden XAML-Code:

```
<Menu Grid.Row="0">
    <MenuItem Header="_Spiel">
        <MenuItem Header="_Beenden" />
    </MenuItem>
</Menu>
```

Code 2.2: Das Gerüst für das Menü

Beim Anklicken des Menüeintrags **Beenden** lassen Sie bitte gewohnt die Anwendung schließen. Damit wollen wir an dieser Stelle das Anlegen der Menüleiste auch schon wieder beenden. Die restlichen Einträge werden wir im weiteren Verlauf ergänzen.

1. *Windows startup location* lässt sich mit „Startposition des Fensters“ übersetzen.

2.2 Das Spielfeld

Für das Spielfeld verwenden wir ein Steuerelement vom Typ **Canvas**. Setzen Sie es bitte in die zweite Zeile. Stellen Sie dann sicher, dass das Steuerelement den gesamten freien Platz einnimmt. Das geht am einfachsten, wenn Sie alle automatisch gesetzten Werte für die Positionierung im XAML-Code löschen und nur die Zuweisung der Zeile stehen lassen. Vergeben Sie dann noch den Namen `spielfeld` an das Steuerelement.

Fügen Sie danach oben rechts in die zweite Zeile zwei Labels ein. Lassen Sie die beiden Labels oben und rechts ausrichten und positionieren Sie sie mit ein wenig Abstand vom oberen und vom rechten Rand. Denken Sie dabei daran, dass wir auch noch ein wenig Platz für die Spielfeldbegrenzungen brauchen.

Als Namen für die Labels verwenden Sie bitte `punktAnzeige` und `zeitAnzeige`. Den Inhalt setzen Sie jeweils auf `0`. Damit die Anzeige besser zu erkennen ist, können Sie auch noch die Schriftgröße verändern. Die entsprechende Eigenschaft finden Sie in der Gruppe **Text** für ein Label.

Passen Sie dann noch die Hintergrundfarbe des Grids an. Wir benutzen in unserem Beispiel ein helles Grau. Sie können aber auch eine beliebige andere Farbe verwenden.



Bitte beachten Sie:

Die Hintergrundfarbe des eigentlichen Spielfelds dürfen Sie nicht setzen. Wir werden gleich eine Kollisionsabfrage programmieren, die nur dann funktioniert, wenn das Steuerelement für das Spielfeld keine Farbe hat.

Das Fenster mit dem Spielfeld sollte jetzt ungefähr so aussehen:

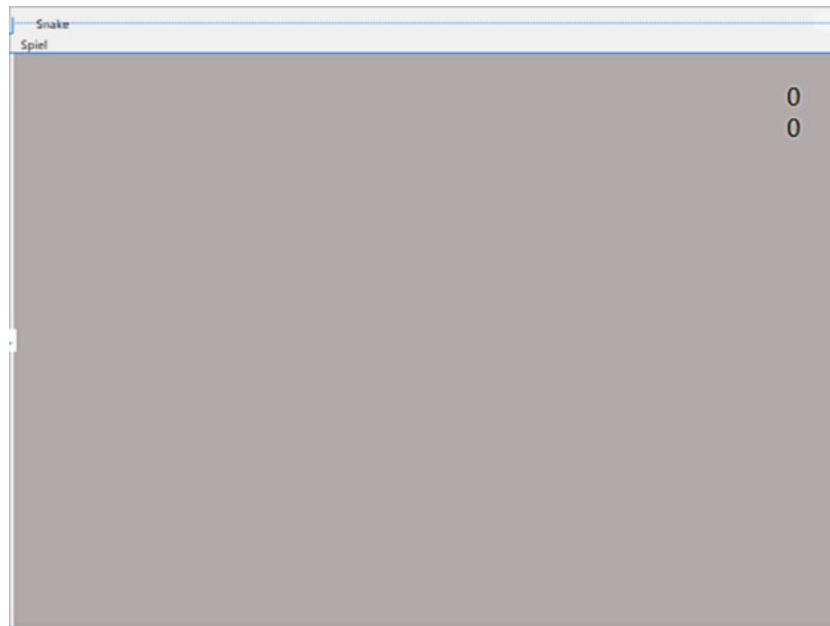


Abb. 2.1: Das Fenster mit dem Spielfeld

Wechseln Sie anschließend in den Quelltext des Fensters und legen Sie für die Klasse `MainWindow` die Felder aus dem Code 2.3 an. Eine kurze Beschreibung der Felder finden Sie dabei jeweils in den Kommentaren.

```
//die Felder
//für die Punkte
int punkte;
//für die Zeit
int zeit;
//für die Richtung der Schlange
int richtung;
//für die Breite der Spielfeldbegrenzung
int balkenBreite;
```

Code 2.3: Die Felder für das Spiel

Legen Sie dann eine Methode `start()` an, die die Felder auf Initialwerte setzt. Das könnten wir zwar auch im Konstruktor machen, mit einer eigenen Methode können wir aber gleich den Neustart des Spiels einfacher umsetzen. Die Methode finden Sie im Code 2.4:

```
void Start()
{
    punkte = 0;
    zeit = 0;
    richtung = 0;
    balkenBreite = 25;
    zeitAnzeige.Content = zeit;
    punktAnzeige.Content = punkte;
}
```

Code 2.4: Die Methode Start()**Hinweis:**

Die Balkenbreite können Sie auch im Konstruktor setzen. Der Wert ändert sich ja nicht. Daher müssen wir sie nicht bei jedem Spielstart neu setzen.

Jetzt kommt das Zeichnen des Spielfelds an die Reihe. Hier werden die vier Balken als Rechtecke an die Ränder gezeichnet. Die Startkoordinaten der Rechtecke sowie die Länge und Breite berechnen wir dabei über die Größe des Spielfelds. Da das Erstellen eines Rechtecks in der WPF ein wenig aufwendiger ist als bei einer Windows-Forms-Anwendung, legen wir für das Zeichnen noch eine Hilfsmethode an. Sie erhält die Werte für das Rechteck über Parameter. Die Hilfsmethode `zeichneRechteck()` und die Methode `zeichneSpielfeld()` finden Sie zusammengefasst im folgenden Code 2.5:

```
//eine Hilfsmethode zum Zeichnen der Begrenzungen
void zeichneRechteck(Point position, double laenge, double
breite)
{
    //einen neuen Balken erzeugen
    Rectangle balken = new Rectangle();
    Canvas.SetLeft(balken, position.X);
    Canvas.SetTop(balken, position.Y);
```

```

balken.Width = laenge;
balken.Height = breite;
SolidColorBrush fuellung = new SolidColorBrush(Colors.Red);
balken.Fill = fuellung;
//und hinzufügen
spielfeld.Children.Add(balken);
}

//zum Erstellen der Begrenzungen
void ZeichneSpielfeld()
{
    //der Balken oben
    //bitte jeweils in einer Zeile eingeben
    ZeichneRechteck(new Point(0, 0), spielfeld.ActualWidth,
    balkenBreite);
    //der Balken rechts
    ZeichneRechteck(new Point(spielfeld.ActualWidth -
    balkenBreite, 0), balkenBreite, spielfeld.ActualHeight);
    //der Balken unten
    ZeichneRechteck(new Point(0, spielfeld.ActualHeight -
    balkenBreite), spielfeld.ActualWidth, balkenBreite);
    //und der Balken links
    ZeichneRechteck(new Point(0, 0), balkenBreite,
    spielfeld.ActualHeight);
}

```

Code 2.5: Die Methoden ZeichneRechteck() und ZeichneSpielfeld()

Neu ist eigentlich nur das Ermitteln der Breite beziehungsweise Höhe des Spielfelds. Die Eigenschaften `Width` und `Heigth` können wir hier nicht verwenden, da unser Spielfeld ja automatisch angepasst wird und wir die Werte gar nicht festgelegt haben. Wir benutzen daher die Eigenschaften `ActualWidth` und `ActualHeigth`. Sie liefern die aktuelle Breite und Höhe nach Anpassungen. Beide Werte sind vom Typ `double`.

Übernehmen Sie jetzt die Anweisungen aus den vorigen Codes. Rufen Sie die Methoden `Start()` und `ZeichneSpielfeld()` dann im Ereignis **Loaded** des Fensters auf. Speichern Sie alle Änderungen und lassen Sie das Programm ausführen. Das Spielfeld sollte ungefähr so aussehen:



Abb. 2.2: Das Spielfeld

Bitte beachten Sie:

Wir greifen auf die berechnete Größe des Spielfelds zu. Sie steht aber erst dann zur Verfügung, wenn das Fenster vollständig geladen wurde. Wenn Sie sich die Werte zu früh beschaffen – zum Beispiel im Konstruktor, erhalten Sie unter Umständen nur den Standardwert 0 für die berechnete Größe.



Hinweis:

Beim Ausführen erscheint eine Warnung, dass das Feld `richtung` einen Wert erhält, der nicht genutzt wird. Diese Warnung können Sie ignorieren. Sie verschwindet gleich, wenn wir die Schlange erstellen.

Im nächsten Schritt können wir uns um die Schlange kümmern.

2.3 Die Schlange

Das Zeichnen des Schlangenkopfes ist nicht weiter schwierig. Hier benutzen wir ein Rechteck, das wir im Spielfeld bewegen. Ein wenig aufwendiger dagegen ist der Schlangenkörper. Er kann aus beliebig vielen einzelnen Elementen bestehen. Bei einer Bewegung der kompletten Schlange werden die Elemente immer auf die alte Position des Vorgängers gesetzt. Das heißt: Die Elemente folgen nicht direkt dem Schlangenkopf, sondern dem Weg, den der Schlangenkopf genommen hat.

Und diesen Weg müssen wir irgendwie speichern. Eine Variante wäre es, die Koordinaten in einer Liste abzulegen und dann die Teile des Schlangenkörpers der Reihe nach an diese Koordinaten zu verschieben. Dazu müssten wir aber relativ viel Aufwand betreiben. Wir gehen daher einen anderen Weg: Wir speichern nicht den kompletten Weg des Schlangenkopfes, sondern merken uns lediglich die alte Position eines Schlangenteils. An diese Position verschieben wir dann beim Bewegen den nachfolgenden Teil der Schlange.

Das können wir recht einfach über eine Schleife und eine Listenkonstruktion der Klasse `List` umsetzen. Bei dieser Klasse handelt es sich um einen Generic².



Generics – auch Generika^{a)} genannt – sind Methoden, Klassen, Strukturen und so weiter, die Platzhalter für die Typen benutzen, die von ihnen verarbeitet werden. Die konkreten Typen für die Platzhalter werden erst zur Laufzeit des Programms eingesetzt.

- a) Der deutsche Begriff Generika bezeichnet auch Medikamente, die bereits vorhandenen Markenmedikamenten in der Wirkstoffzusammensetzung exakt gleichen.

Grundsätzlich lässt sich eine generische Liste der Klasse `List` mit einem Array vergleichen. Es werden Daten vom selben Typ sequenziell abgelegt und können über einen Index wieder abgerufen werden.



Der Index beginnt auch bei der Klasse `List` mit dem Wert 0.

Anders als ein Array verfügt eine generische Liste allerdings über zahlreiche Methoden, mit denen Sie die Liste direkt verarbeiten können. So können Sie zum Beispiel neue Elemente an beliebigen Positionen einfügen und vorhandene Elemente löschen. Außerdem wird die Liste beim Einfügen neuer Elemente automatisch erweitert.

Das Anlegen einer generischen Liste der Klasse `List` unterscheidet sich nicht weiter vom Anlegen eines Arrays. Sie vereinbaren zunächst eine Variable vom Typ `List` und erzeugen anschließend über den Operator `new` eine neue Instanz der Klasse `List`. Damit der Compiler weiß, welche Daten in der Liste verwaltet werden, geben Sie hinter dem Namen der Klasse `List` die gewünschten Datentypen in spitzen Klammern an. Zusätzlich müssen Sie in runden Klammern den Konstruktor der Klasse aufrufen.

Die folgende Anweisung vereinbart zum Beispiel eine Variable `listeString` und erzeugt über diese Variable eine Liste mit Zeichenketten.

```
List<string> listeString = new List<string>();
```

Hinweis:

Die Klasse `List` befindet sich im Namensraum `System.Collections.Generic`. Dieser Namensraum wird aber in der Regel automatisch durch das Quelltextgerüst von Visual Studio eingebunden.

Da wir beim Verhalten zwischen dem Kopf der Schlange und dem Rumpf unterscheiden müssen, erstellen wir getrennte Klassen für den Schlangenkopf und die Schlangenteile. Die Klasse für den Schlangenkopf leiten wir dabei von der Klasse für die Schlangenteile ab. Auf diese Weise können wir Instanzen der beiden Klassen in einer Liste verwalten und durch eine Schleife verarbeiten.

2. *Generic* bedeutet übersetzt so viel „allgemeingültig“.

Zur Erinnerung:

Abgeleitete Klassen können Sie überall dort benutzen, wo Sie auch die übergeordnete Klasse verwenden können. Dieses Prinzip wird **Substitution** genannt.

Machen wir uns jetzt an die praktische Umsetzung. Beginnen wir mit den beiden Klassen für ein Schlangenteil und den Schlangenkopf. Sie finden sie in den folgenden Codes. Die Besonderheiten sehen wir uns jeweils im Anschluss an.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Snake
{
    class Schlangenteil
    {
        //die Felder
        //die Position
        protected Point position;
        //die alte Position
        protected Point altePosition;
        //die Farbe
        protected Color farbe;
        //die Form
        protected Rectangle quadrat;
        //die Größe
        protected int groesse;

        //die Methoden
        //der Konstruktor zum Setzen der Position und der Farbe
        public Schlangenteil(Point position, Color farbe)
        {
            this.position.X = position.X;
            this.position.Y = position.Y;
            //alte und neue Position sind erst einmal identisch
            altePosition.X = this.position.X;
            altePosition.Y = this.position.Y;
            //Farbe setzen
            this.farbe = farbe;
            //die Größe wird fest gesetzt
            groesse = 20;
            //ein neues Rechteck erzeugen
            quadrat = new Rectangle();
        }

        //eine leere Methode zum Bewegen
        //Sie wird in der abgeleiteten Klasse überschrieben
        public virtual void Bewegen(int richtung)
        {
        }

        //die neue Position setzen
    }
}
```

```

public void SetzePosition(Point neuePosition)
{
    //die alte Position speichern
    altePosition = position;
    position = neuePosition;
}

//das Teil anzeigen
public void Zeichnen(Canvas meinCanvas)
{
    //das Quadrat löschen
    meinCanvas.Children.Remove(quadrat);
    //positionieren
    Canvas.SetLeft(quadrat, position.X);
    Canvas.SetTop(quadrat, position.Y);
    //die Größe setzen
    quadrat.Width = groesse;
    quadrat.Height = groesse;
    //Farbe und Rahmen setzen
    SolidColorBrush fuellung = new SolidColorBrush(farbe);
    quadrat.Fill = fuellung;
    SolidColorBrush rahmen = new
    SolidColorBrush(Colors.White);
    quadrat.Stroke = rahmen;
    //wieder hinzufügen
    meinCanvas.Children.Add(quadrat);
}

//die alte Position liefern
public Point LiefereAltePosition()
{
    return altePosition;
}

//die Größe liefern
public int LiefereGroesse()
{
    return groesse;
}

//die Position liefern
//die Methode wird in der abgeleiteten Klasse überschrieben
public virtual Point LieferePosition()
{
    return position;
}
}

```

Code 2.6: Die Klasse Schlangenteil

Zunächst einmal vereinbaren wir einige Felder für die Klasse. Damit wir auch aus der abgeleiteten Klasse auf diese Felder zugreifen können, verwenden wir die Sichtbarkeit `protected`.

Im Konstruktor setzen wir die Initialwerte. Da es direkt nach dem Erzeugen keine vorige Position gibt, stimmen die Daten zunächst einmal mit der aktuellen Position überein.

Danach folgt eine virtuelle Methode `Bewegen()`. Sie ist in unserer Klasse für die Schlangenteile leer, da sich die Teile nicht selbst bewegen, sondern nur dem Schlangenkopf folgen. Wir werden sie aber gleich in der Klasse für den Schlangenkopf überschreiben.

Das Positionieren der Schlangenteile erfolgt über die Methode `SetzePosition()`. Hier setzen wir die Position neu und speichern die vorige Position im Feld `altePosition`.

In der Methode `Zeichnen()` erstellen wir ein neues Schlangenteil und lassen es im Spielfeld anzeigen. Damit der Eindruck von Bewegung entsteht und die Schlange beim Zeichnen nicht immer länger wird, löschen wir das Teil vor dem Verschieben.

Hinweis:

Beim ersten Zeichnen wird das Rechteck gelöscht und genau an derselben Stelle wieder neu gezeichnet. Um das zu vermeiden, müssten wir aber einen Aufwand betreiben. Wir nehmen diese überflüssige Aktion daher in Kauf.

Die anderen Methoden liefern dann lediglich Werte zurück. Die Methode `LieferePosition()` werden wir gleich in der abgeleiteten Klasse ebenfalls überschreiben. Sie ist daher virtuell.

Die abgeleitete Klasse `Schlangenkopf` sieht erst einmal so aus:

```
using System.Windows;
using System.Windows.Media;

namespace Snake
{
    class Schlangenkopf : Schlangenteil
    {
        //der Konstruktor
        //bitte in einer Zeile eingeben
        public Schlangenkopf(Point position, Color farbe) :
            base(position, farbe)
        {
        }

        public override void Bewegen(int richtung)
        {
            //die alte Position speichern
            altePosition = position;
            //und verändern
            switch (richtung)
            {
                //nach oben
                case 0:
                    position.Y = position.Y - groesse;
                    break;
                //nach rechts
                case 1:
```

```
        position.X = position.X + groesse;
        break;
    //nach unten
    case 2:
        position.Y = position.Y + groesse;
        break;
    //nach links
    case 3:
        position.X = position.X - groesse;
        break;
    }
}

public override Point LieferePosition()
{
    return position;
}
}
```

Code 2.7: Die Klasse Schlangenkopf

Schauen wir uns auch hier wieder die Besonderheiten an. Da unsere Basisklasse `Schlangenteil` keinen Standardkonstruktor hat, müssen wir im Konstruktor der Klasse `Schlangenkopf` über `base` den passenden Konstruktor der Basisklasse selbst aufrufen.



Zur Erinnerung:

Eine abgeleitete Klasse kann automatisch nur auf den Standardkonstruktor der übergeordneten Klasse zugreifen. Wenn es keinen Standardkonstruktor gibt, müssen Sie den Konstruktor selbst aufrufen.

In der überschriebenen Methode `Zeichnen()` speichern wir die alte Position und verändern dann die Position abhängig vom Wert des Arguments `richtung` entweder nach oben, nach rechts, nach unten oder nach links.

Die überschriebene Methode `LieferePosition()` macht zunächst einmal genau das, was auch die Methode der Basisklasse macht – nämlich die Position zurückgeben. Wir werden sie aber gleich so anpassen, dass wir sie für eine Kollisionsabfrage nutzen können.

Legen Sie die beiden Klassen aus den vorigen Codes jetzt bitte mit der Funktion **Projekt/Klasse hinzufügen** ... an.

Zum Test lassen wir nun eine Schlange mit einem Kopf und einigen Teilen erzeugen. Legen Sie dazu im ersten Schritt in der Klasse für das Fenster ein Feld für die Liste an. Die Vereinbarung könnte so aussehen:

```
List<Schlangenteil> schlange;
```

Erzeugen Sie die Liste dann im Konstruktor mit der Anweisung

```
schlange = new List<Schlanqenteil>();
```

Danach können Sie in der Methode `Start()` die Liste füllen. Das erste Element ist eine Instanz der Klasse `Schlangenkopf`. Danach folgen Instanzen der Klasse `Schlangenteil`. Die entsprechenden Anweisungen sehen so aus:

```
//den Schlangenkopf erzeugen und positionieren
//bitte in einer Zeile eingeben
Schlangenkopf meineSchlangeKopf = new Schlangenkopf(new
Point(spielfeld.ActualWidth / 2, spielfeld.ActualHeight / 2),
Colors.Red);
//in die Liste setzen
schlange.Add(meineSchlangeKopf);
//zum Test ein paar Teile erzeugen
for (int index = 0; index < 10; index++)
{
    //bitte in einer Zeile eingeben
    Schlangenteil sTeil = new Schlangenteil(new
    Point(schlange[index].LiefereAltePosition().X,
    schlange[index].LiefereAltePosition().Y +
    schlange[index].LiefereGroesse()), Colors.Black);
    schlange.Add(sTeil);
}
```

Code 2.8: Das Erzeugen einer Testschlange

Mit der Anweisung

```
Schlangenkopf meineSchlangeKopf = new Schlangenkopf(new
Point(spielfeld.ActualWidth / 2, spielfeld.ActualHeight / 2),
Colors.Red);
```

erzeugen wir den Kopf der Schlange. Er wird in die Mitte des Spielfelds gesetzt und ist rot.

Die Anweisung

```
schlange.Add(meineSchlangeKopf);
```

fügt den Schlangenkopf an das Ende unserer Liste für die Schlange an.

In der Schleife

```
for (int index = 0; index < 10; index++)
{
    Schlangenteil sTeil = new Schlangenteil(new
    Point(schlange[index].LiefereAltePosition().X,
    schlange[index].LiefereAltePosition().Y +
    schlange[index].LiefereGroesse()), Colors.Black);
    schlange.Add(sTeil);
}
```

erzeugen wir dann die Teile des Schlangenkörpers und fügen sie in die Liste für die Schlange ein. Die Position des Teils wird dabei immer durch die Position des Vorgängers bestimmt. Sie erhalten wir über den Ausdruck

`schlange[index].LiefereAltePosition().X` beziehungsweise den Ausdruck `schlange[index].LiefereAltePosition().Y`. Auf die Y-Position addieren wir noch die Größe eines Elements auf. Damit wird das Element unten an das Ende der Schlange angehängt.



Der Ausdruck `schlange[0]` liefert bei unserer Konstruktion immer den Schlangenkopf. Daher müssen wir die Schleife auch mit dem Index 0 beginnen lassen.

Übernehmen Sie die Anweisungen jetzt und testen Sie die Erweiterungen. Es wird allerdings nichts Sichtbares geschehen. Denn gezeichnet wird die Schlange ja erst beim Bewegen.

2.4 Das Bewegen der Schlange

Dazu benutzen wir einen Timer, der jede Sekunde ausgelöst wird. Er ruft die Methoden `Bewegen()` und `Zeichnen()` für den Schlangenkopf und die Methoden `SetzePosition()` und `Zeichnen()` für die Schlangenteile auf.

Da wir später aus mehreren Methoden auf den Timer zugreifen müssen, vereinbaren Sie bitte zunächst ein Feld `timerSchlange` vom Typ `DispatcherTimer` für den Timer. Denken Sie dabei bitte daran, dass die Klasse `DispatcherTimer` im Namensraum `System.Windows.Threading` liegt.

Legen Sie dann die Methode aus dem folgenden Code 2.9 an. Wir lassen sie gleich durch den Timer ausführen:

```
private void Timer_SchlangeBewegen(object sender, EventArgs e)
{
    //den Kopf in die angegebene Richtung bewegen
    schlange[0].Bewegen(richtung);
    //und zeichnen
    schlange[0].Zeichnen(spielfeld);
    //die Teile in einer Schleife bewegen
    for (int index = 1; index < schlange.Count; index++)
    {
        //bitte in einer Zeile eingeben
        schlange[index].SetzePosition(schlange[index - 1].LiefereAltePosition());
        schlange[index].Zeichnen(spielfeld);
    }
}
```

Code 2.9: Die Methode für den Timer

Mit den ersten beiden Anweisungen

```
schlange[0].Bewegen(richtung);
schlange[0].Zeichnen(spielfeld);
```

bewegen wir den Kopf der Schlange in die angegebene Richtung und lassen ihn neu zeichnen.

Die Schleife

```
for (int index = 1; index < schlange.Count; index++)
{
    schlange[index].SetzePosition(schlange[index - 1].LiefereAltePosition());
    schlange[index].Zeichnen(spieldfeld);
}
```

übernimmt dann das Verschieben der Schlangenteile im Rumpf. Dazu gehen wir die Liste mit den Schlangenteilen durch und verschieben jedes Element an die Position des Vorgängers. Anders als beim Erstellen der Schlange müssen wir hier aber mit dem Index 1 beginnen und auf den Vorgänger über den Index `index - 1` zugreifen. Denn den Kopf der Schlange haben wir ja bereits mit den Anweisungen vor der Schleife verschoben.

Hinweis:

Die Anzahl der Elemente in der Liste ermitteln wir über die Eigenschaft `Count`. Denken Sie auch hier bitte daran, dass der Index mit 0 beginnt. Die Abfrage muss also auf „kleiner“ erfolgen und nicht auf „kleiner gleich“.

Erzeugen Sie jetzt noch eine neue Instanz für den Timer und lassen Sie die Methode `Timer_Bewegen()` jede Sekunde ausführen. Dazu können Sie die folgenden Anweisungen am Ende des Konstruktors ergänzen:

```
//die Instanz erzeugen
timerSchlange = new DispatcherTimer();
//das Intervall setzen
timerSchlange.Interval = TimeSpan.FromMilliseconds(1000);
//die Methode für das Ereignis zuweisen
timerSchlange.Tick += new EventHandler(Timer_SchlangeBewegen);
//den Timer starten
timerSchlange.Start();
```

Code 2.10: Das Erzeugen und Starten des Timers

Führen Sie dann noch einmal einen Test durch. Jetzt sollte sich die Schlange auch nach oben bewegen.

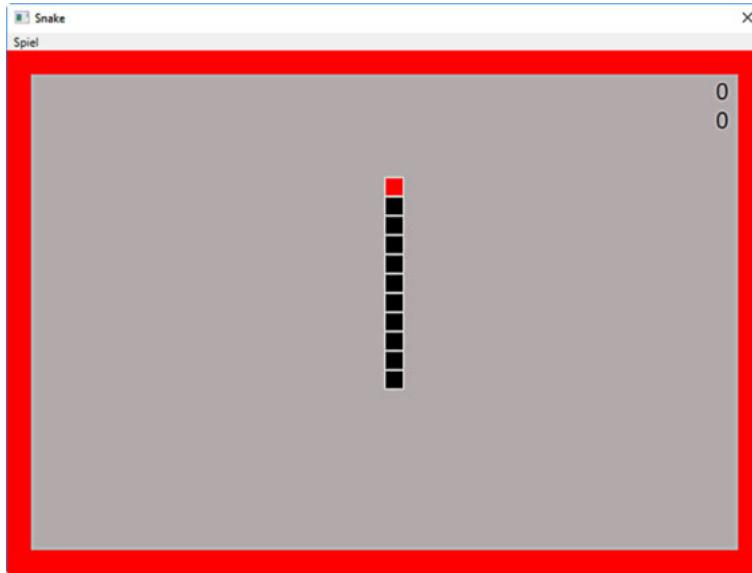


Abb. 2.3: Die Schlange

Hinweis:

Die Schlange wird erst beim ersten Auslösen des Timers angezeigt. Es dauert also eine Sekunde, bis etwas zu sehen ist.

Sorgen wir jetzt noch dafür, dass sich die Schlange lenken lässt. Da ist recht schnell erledigt. Wir prüfen im Ereignis **KeyDown** für das Fenster, ob eine der Cursortasten gedrückt wurde, und setzen den Wert unseres Felds `richtung` entsprechend.

Die komplette Methode sieht so aus:

```
private void Window_KeyDown(object sender, KeyEventArgs e)
{
    //je nach Taste die Richtung setzen
    //oben
    if (e.Key == Key.Up)
        richtung = 0;
    //unten
    if (e.Key == Key.Down)
        richtung = 2;
    //links
    if (e.Key == Key.Left)
        richtung = 3;
    //rechts
    if (e.Key == Key.Right)
        richtung = 1;
}
```

Code 2.11: Die Methode `Window_KeyDown()`

Jetzt sollte sich die Schlange auch lenken lassen.

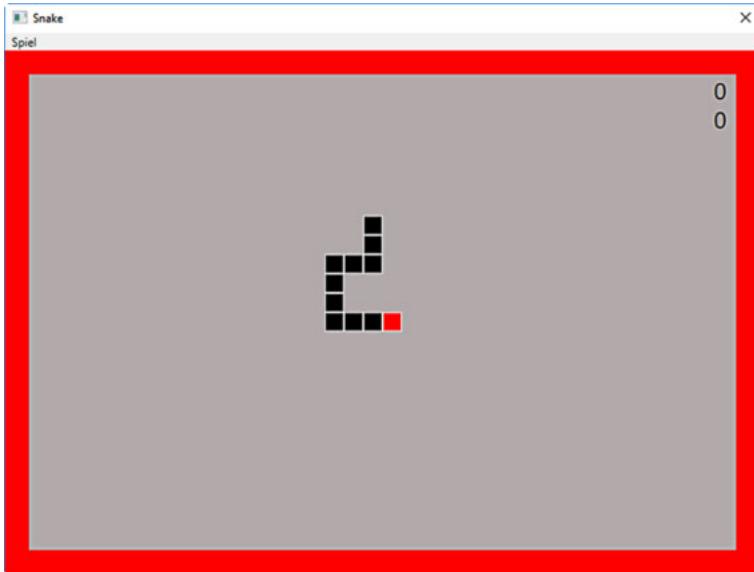


Abb. 2.4: Die Schlange lässt sich lenken

Allerdings verschwindet die Schlange aus dem Spielfeld, wenn sie die Grenzen passiert. Darum werden wir uns gleich kümmern, wenn wir die Kollisionsabfrage programmieren. Jetzt erstellen wir erst einmal eine Klasse für das „Schlangenfutter“ – die Äpfel.

2.5 Die Äpfel

In dieser Klasse erzeugen wir einen Kreis an einer zufälligen Position im Spielfeld und lassen ihn anzeigen. Außerdem brauchen wir eine Methode, um den Kreis wieder aus dem Spielfeld zu löschen. Die Klasse sieht erst einmal so aus:

```
using System;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Snake
{
    class Apfel
    {
        //die Felder
        //die Farbe
        Color farbe;
        //die Form
        Ellipse kreis;
        //die Größe
        int groesse;

        //die Methoden
        //der Konstruktor zum Setzen der Farbe und der Größe und des
        //Spielfelds
        public Apfel(Color farbe, int groesse)
        {
```

```

    //Farbe setzen
    this.farbe = farbe;
    //die Größe setzen
    this.groesse = groesse;
    //einen neuen Kreis erzeugen
    kreis = new Ellipse();
}

//den Apfel anzeigen
public void Anzeigen(Canvas meinCanvas, int balkenbreite)
{
    //den Zufallsgenerator initialisieren
    Random zufall = new Random();
    //das Minimum ist die Balkenbreite
    int min = balkenbreite;
    //das Maximum ermitteln
    //die Balken und die Größe für die maximalen Positionen
    //abziehen
    //bitte jeweils in einer Zeile eingeben
    int maxX = (int)meinCanvas.ActualWidth - balkenbreite -
    groesse;
    int maxY = (int)meinCanvas.ActualHeight - balkenbreite -
    groesse;
    //positionieren
    Canvas.SetLeft(kreis, zufall.Next(min, maxX));
    Canvas.SetTop(kreis, zufall.Next(min, maxY));

    //die Größe setzen
    kreis.Width = groesse;
    kreis.Height = groesse;
    //Farbe setzen
    SolidColorBrush fuellung = new SolidColorBrush(farbe);
    kreis.Fill = fuellung;
    //hinzufügen
    meinCanvas.Children.Add(kreis);
}

//den Apfel entfernen
public void Entfernen(Canvas meinCanvas)
{
    meinCanvas.Children.Remove(kreis);
}
}
}

```

Code 2.12: Die Klasse Apfel

Interessant sind vor allem die Anweisungen zum Positionieren. Denn wir können den Apfel ja nicht einfach an einer beliebigen Position ablegen, sondern müssen die Balken für die Spielfeldgrenzen berücksichtigen. Daher ermitteln wir mit den Anweisungen

```

int min = balkenbreite;
int maxX = (int)meinCanvas.ActualWidth - balkenbreite - groesse;
int maxY = (int)meinCanvas.ActualHeight - balkenbreite -
groesse;

```

zunächst den minimalen Wert für die Position und den maximalen X- und Y-Wert. Beim minimalen Wert müssen wir lediglich die Balkenbreite berücksichtigen. Sie übergeben wir als Argument an die Methode. Für die maximale Position dagegen müssen wir neben der Balkenbreite auch noch die Größe des Apfels abziehen. Denn angegeben wird ja beim Positionieren die linke obere Ecke.

Mit der minimalen und der maximalen Position ermitteln wir dann über die Klasse `Random` und die Methode `Next()` eine zufällige Position und fügen die Ellipse an dieser Position in das Spielfeld ein.

Legen Sie jetzt bitte die neue Klasse für den Apfel an. Führen Sie dann einen Test durch und lassen Sie zum Beispiel in der Methode `Start()` der Klasse der Anwendung mehrere Äpfel erzeugen. Dazu können Sie zum Beispiel die folgende Schleife verwenden:

```
for (int test = 0; test < 20; test++)
{
    Apfel testApfel = new Apfel(Colors.Green, 20);
    testApfel.Anzeigen(spieldfeld, balkenBreite);
}
```

Code 2.13: Das Erzeugen der Äpfel

Beim Ausführen werden Sie allerdings eine Überraschung erleben. Denn angezeigt wird scheinbar lediglich ein einziger Apfel.

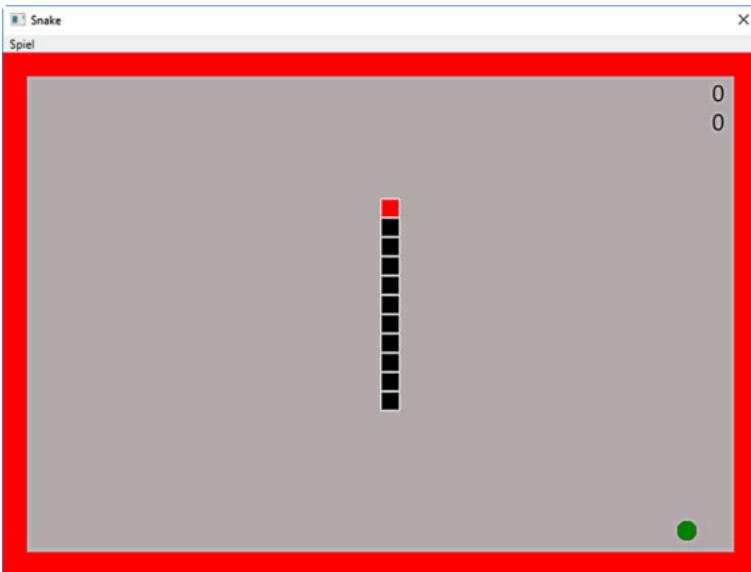


Abb. 2.5: Ein Apfel im Spielfeld

Tatsächlich werden alle Äpfel, die Sie erzeugen, auch ins Spielfeld gesetzt – allerdings an derselben Position. Denn die Initialisierung des Zufallsgenerators für die Klasse `Random` erfolgt ohne weitere Angaben über die Uhrzeit des Systems. Und das führt beim schnellen Erzeugen mehrere Instanzen hintereinander dazu, dass die Instanzen dieselben Zufallswerte liefern.

Das Problem lässt sich aber recht einfach lösen. Wir übergeben an den Konstruktor der Klasse `Random` einen Wert, den wir aus dem Hashcode der Instanzen für die Klasse der Äpfel ableiten.



Ein Hashcode ist ein numerischer Wert, über den Daten nahezu eindeutig identifiziert werden können. So haben unterschiedliche Instanzen einer Klasse in der Regel auch unterschiedliche Hashcodes.

Um den Hashcode für die Initialisierung zu benutzen, übergeben Sie das Ergebnis der Methode `GetHashCode()` an den Konstruktor der Klasse `Random`. Die Anweisung sieht dann so aus:

```
Random zufall = new Random(GetHashCode());
```

Jetzt sieht die Verteilung der Äpfel schon deutlich besser aus:

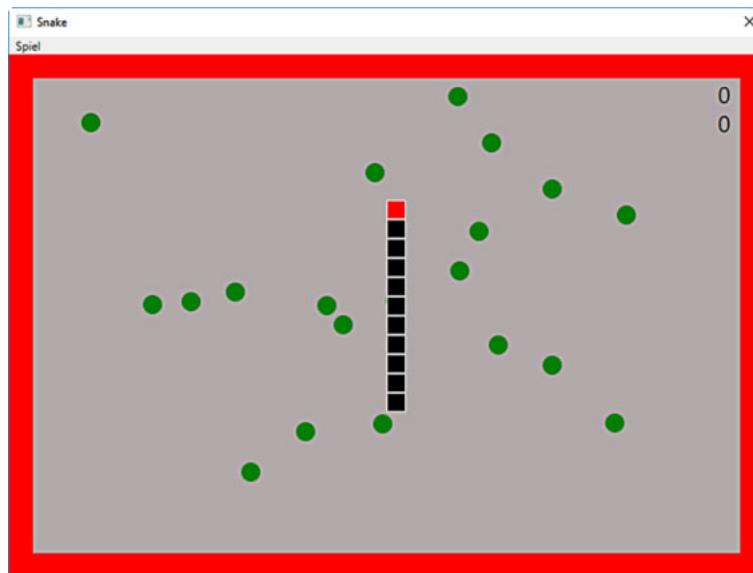


Abb. 2.6: Mehrere Äpfel auf dem Spielfeld



Bitte beachten Sie:

Die Methode `GetHashCode()` liefert nicht garantiert unterschiedliche Werte für verschiedene Instanzen einer Klasse. Da das in unserem Fall aber nur dazu führt, dass mehrere Äpfel an derselben Position stehen, müssen wir uns um dieses Problem nicht weiter kümmern.

2.6 Die Kollisionsabfrage

Jetzt fehlt noch eine Kollisionsabfrage. Hier müssen wir unterscheiden können, ob die Schlange mit einem Apfel zusammengestoßen ist oder mit einem anderen Objekt. Bei einem Zusammenstoß mit einem Apfel soll der Spieler Punkte bekommen und die Schlange ein neues Schlangenteil angehängt bekommen. Bei einer Kollision mit einem anderen Objekt – also dem Rumpf der Schlange oder der Spielfeldbegrenzung – dagegen soll das Spiel beendet werden.

Was sich kompliziert anhört, ist recht einfach programmiert. Denn das .NET Framework stellt uns eine Methode `HitTest()`³ der Klasse `System.Windows.Media.VisualTreeHelper` zur Verfügung. Mit dieser Methode

können wir bei der WPF prüfen, ob sich an einer bestimmten Position in einem bestimmten Objekt ein anderes Objekt befindet. Die Methode liefert ein Ergebnis vom Typ `HitTestResult`. Es enthält entweder Angaben zu dem Objekt, das gefunden wurde, oder den Wert `null`, wenn nichts gefunden wurde.

In unserem Beispiel können wir prüfen, ob sich nach dem Verschieben des Schlangenkopfes ein anderes Objekt unter der Mitte des Schlangenkopfes im Spielfeld befindet. Dazu müssen wir die Methode `LieferePosition()` für den Schlangenkopf ein wenig anpassen.

```
//die überschriebene Methode zum Liefern der Position
public override Point LieferePosition()
{
    //bitte in einer Zeile eingeben
    return (new Point(position.X + (groesse / 2), position.Y +
    (groesse / 2)));
}
```

Code 2.14: Die neue Methode zum Liefern der Position

Wir lassen jetzt hier den Punkt zurückgeben, der in der Mitte des Rechtecks für den Kopf liegt. Dazu addieren wir auf die Position oben links jeweils die Hälfte der Größe.

Die Abfrage auf einen Treffer kann dann so aussehen:

```
//die Trefferabfrage
//bitte in einer Zeile eingeben
HitTestResult treffer = VisualTreeHelper.HitTest(spielfeld,
schlange[0].LieferePosition());
if (treffer != null)
{
    MessageBox.Show(treffer.VisualHit.ToString());
}
```

Code 2.15: Die Abfrage auf einen Treffer

Mit der Anweisung

```
HitTestResult treffer = VisualTreeHelper.HitTest(spielfeld,
schlange[0].LieferePosition());
```

überprüfen wir, ob es an der Position, die die Methode `LieferePosition()` für den Schlangenkopf liefert, ein anderes Objekt im Spielfeld gibt.

Wenn das der Fall ist – `treffer` also einen Wert ungleich `null` hat – lassen wir über die Anweisung

```
MessageBox.Show(treffer.VisualHit.ToString());
```

den Typ des getroffenen Objekts ausgeben.

3. `HitTest` lässt sich mit „Treffertest“ übersetzen.


Bitte beachten Sie:

Sie müssen das Spielfeld auf Treffer prüfen. Wenn Sie das Fenster oder das Grid des Fensters für die Prüfung verwenden, löst jede Bewegung einen Treffer aus – nämlich entweder mit dem Grid oder mit dem Spielfeld. Denn diese Objekte befinden sich in der Hierarchie der Elemente direkt unterhalb des Fensters beziehungsweise direkt unterhalb des Grids.

Legen Sie jetzt eine neue Methode `KollisionPruefen()` in der Klasse für die Anwendung an. Übernehmen Sie in dieser Methode die Anweisungen aus dem vorigen Code und rufen Sie die Methode im Timer für das Bewegen der Schlange auf. Achten Sie dabei bitte darauf, dass Sie die Prüfung erst nach dem Verschieben der Schlange ausführen lassen dürfen.

Testen Sie die Erweiterung dann. Das Programm sollte jetzt auf Kollisionen reagieren können.

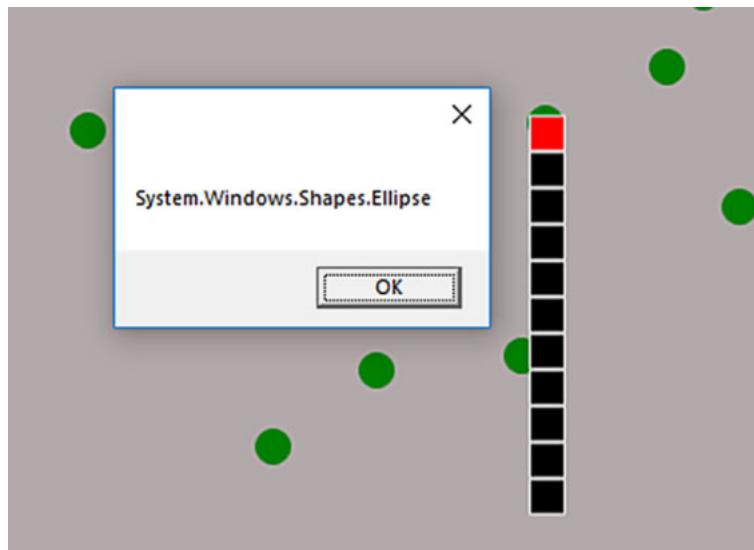


Abb. 2.7: Eine Kollision zwischen zwei Objekten

Allerdings funktioniert die Abfrage nicht immer ganz präzise. Eine Kollision mit der Spielbegrenzung tritt zum Beispiel mit unseren Werten erst auf, wenn sich der Schlangenkopf bereits ein wenig in der Begrenzung findet. Das nehmen wir aber bewusst in Kauf.

Sehr viel ärgerlicher für den Spieler ist es aber, dass auch die Abfrage auf eine Kollision mit den Äpfeln nicht immer korrekt funktioniert. Wenn zum Beispiel der Rand des Schlangenkopfes einen Apfel nur leicht berührt, passiert nichts. Denn wir benutzen ja den Mittelpunkt des Schlangenkopfes für die Prüfung.

Sie könnten nun die Prüfung auf den gesamten Kopf der Schlange ausdehnen. Das ist allerdings recht aufwendig. Wir gehen daher einen anderen Weg und setzen um jeden Apfel noch ein etwas größeres Rechteck, das wir für die Kollisionsabfrage verwenden. Das funktioniert zwar ebenfalls nicht absolut perfekt, reicht für unser Spiel aber völlig aus.

Ergänzen Sie bitte im ersten Schritt ein neues Feld vom Typ `Rectangle` in der Klasse `Apfel`. Wir verwenden den Bezeichner `rechteckKollision`. Weisen Sie dem Feld im Konstruktor eine neue Instanz der Klasse `Rectangle` zu und setzen Sie den Namen für das Rechteck auf `Kollision`. Die entsprechenden Anweisungen sehen so aus:

```
rechteckKollision = new Rectangle();
rechteckKollision.Name = "Kollision";
```

In der Methode `Anzeigen()` setzen Sie dann die Größe und die Position für das Rechteck und zeigen es an. Die Anweisungen sehen so aus:

```
//den Dummy für die Kollision erstellen
rechteckKollision.Width = kreis.Width + (groesse - 1);
rechteckKollision.Height = kreis.Height + (groesse - 1);
//Farbe setzen
fuellung = new SolidColorBrush(Colors.Aqua);
rechteckKollision.Fill = fuellung;
//bitte jeweils in einer Zeile eingeben
Canvas.SetLeft(rechteckKollision, Canvas.GetLeft(kreis) -
((groesse - 1) / 2));
Canvas.SetTop(rechteckKollision, Canvas.GetTop(kreis) -
((groesse - 1) / 2));

//hinzufügen
//erst das Rechteck für die Kollision
meinCanvas.Children.Add(rechteckKollision);
//dann den Kreis
meinCanvas.Children.Add(kreis);
```

Code 2.16: Das Erzeugen des Rechtecks für die Kollision

Die Größe des Rechtecks verdoppeln wir fast im Verhältnis zum Kreis. Damit reichen auch kleinen Berührungen am Rand aus. Dann setzen wir die Farbe und positionieren das Rechteck so, dass es genau mittig zum Kreis liegt. Anschließend lassen wir die beiden Objekte anzeigen. Bitte beachten Sie dabei, dass die Objekte in der Reihenfolge eingefügt werden, in der Sie die Methode `Add()` aufrufen. Wenn Sie die Anweisungen zum Einfügen in dem vorigen Code vertauschen, würde das Rechteck also vor dem Kreis angezeigt. Damit wäre der Kreis unsichtbar.

Das Ergebnis sollte dann so aussehen wie in der Abb. 2.8:

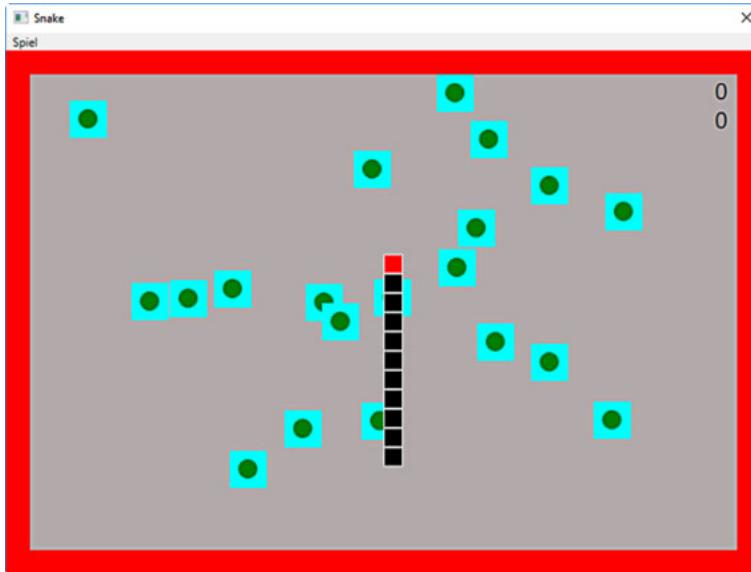


Abb. 2.8: Die Rechtecke für die Kollision



Bitte beachten Sie:

Jetzt kann auch mehrfach eine Kollision mit einem Apfel ausgelöst werden – einmal mit dem umgebenden Rechteck und einmal mit dem Kreis. Da wir aber gleich sofort auf die erste Kollision reagieren werden, spielt das keine besondere Rolle.

Sorgen Sie dann noch dafür, dass das Rechteck für die Kollisionsabfrage auch wieder aus der Zeichenfläche entfernt wird. Ergänzen Sie dazu die Anweisung

```
meinCanvas.Children.Remove(rechteckKollision);
```

in der Methode `Entfernen()` der Klasse `Apfel`.

Jetzt bauen wir die Kollisionsabfrage noch so um, dass auf einen Zusammenstoß mit unterschiedlichen Objekten auch unterschiedlich reagiert wird. Da wir nun auch ein Rechteck für die Kollision mit dem Apfel verwenden, reicht eine einfache Abfrage auf den Typ des Objekts, das wir gefunden haben, nicht aus. Wir müssen den Namen vergleichen.

Vergeben Sie bitte im ersten Schritt an die Spielfeldbegrenzungen einen Namen wie `Grenze`. Ergänzen Sie dazu die Anweisung

```
balken.Name = "Grenze";
```

in der Methode `ZeichneRechteck()`.

Vergeben Sie dann auch an die Schlangenteile einen Namen. Ergänzen Sie dazu am Ende des Konstruktors der Klasse `Schlangenteil` die Anweisung

```
quadrat.Name = "Schlange";
```

Passen Sie danach die Methode `KollisionPruefen()` so an wie im folgenden Code. Was dort genau passiert, erklären wir Ihnen im Anschluss.

```

private void KollisionPruefen()
{
    //die Trefferabfrage
    //bitte in einer Zeile eingeben
    HitTestResult treffer = VisualTreeHelper.HitTest(spielfeld,
        schlange[0].LieferePosition());
    //haben wir etwas getroffen?
    if (treffer != null)
    {
        //den Namen beschaffen
        string name = ((Shape)(treffer.VisualHit)).Name;
        //was haben wir getroffen?
        //war es der Rand oder die Schlange?
        if (name == "Grenze" || name == "Schlange")
            //dann halten wir erst einmal einfach den Timer an
            timerSchlange.Stop();
        //war es ein Apfel oder das Rechteck eines Apfels?
        if (name == "Apfel" || name == "Kollision")
        {
            //die Punkte erhöhen und neu anzeigen
            punkte = punkte + 10;
            punktAnzeige.Content = punkte;
            //ein Teil hinten an die Schlange anhängen
            //bitte in einer Zeile eingeben
            Schlangenteil sTeil = new Schlangenteil(new
                Point(schlange[schlange.Count -
                    1].LiefereAltePosition().X,
                    schlange[schlange.Count - 1].LiefereAltePosition().Y +
                    schlange[schlange.Count - 1].LiefereGroesse()),
                Colors.Black);
            schlange.Add(sTeil);
        }
    }
}

```

Code 2.17: Die Reaktion auf eine Kollision

Sehen wir uns die Anweisungen der Reihe nach an.

Zuerst prüfen wir, ob es eine Kollision im Spielfeld gegeben hat. Das übernehmen die beiden Anweisungen

```

HitTestResult treffer = VisualTreeHelper.HitTest(spielfeld,
    schlange[0].LieferePosition());
if (treffer != null)

```

Wenn es einen Treffer gab, beschaffen wir uns mit der Anweisung

```
string name = ((Shape)(treffer.VisualHit)).Name;
```

den Namen des Objekts. Das geht aber nicht direkt, sondern nur durch ein Casting des Objekts in die Klasse Shape. Über diese Klasse können wir auch auf den Namen zugreifen.

Dann prüfen wir, ob wir mit dem Rand oder der Schlange zusammengestoßen sind, und halten den Timer an. Das erledigen die Anweisungen

```
//war es der Rand oder die Schlange?
if (name == "Grenze" || name == "Schlange")
    timerSchlange.Stop();
```

Bei einem Zusammenstoß mit einem Apfel oder einem Rechteck, das einen Apfel umgibt, erhöhen wir die Punkte und hängen ein neues Element an die Schlange an. Das erfolgt durch die Anweisungen

```
if (name == "Apfel" || name == "Kollision")
{
    punkte = punkte + 10;
    punktAnzeige.Content = punkte;
    Schlangenteil sTeil = new Schlangenteil(new
        Point(schlange[schlange.Count - 1].LiefereAltePosition().X,
        schlange[schlange.Count - 1].LiefereAltePosition().Y +
        schlange[schlange.Count - 1].LiefereGroesse()), Colors.Black);
    schlange.Add(sTeil);
}
```

Die Position, an der das neue Element angehängt wird, erhalten wir über den Ausdruck `schlange.Count - 1`.



Zur Erinnerung:

Die Eigenschaft `Count` liefert die Anzahl der Elemente in einer Liste. Da der Index nullbasiert ist, müssen wir von der Anzahl noch 1 abziehen.

Übernehmen Sie die Erweiterungen jetzt und testen Sie das Spiel. Bei jedem „gefressenen“ Apfel wird die Schlange länger und der Spieler erhält 10 Punkte. Berührt die Schlange den Rand, bleibt sie stehen.

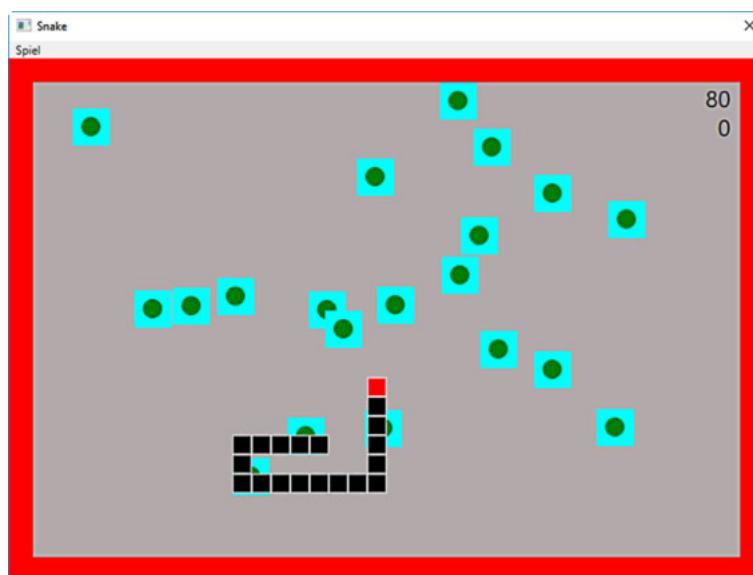


Abb. 2.9: Die Schlange wird immer länger

Hinweis:

Wir werden uns gleich noch detailliert darum kümmern, was bei einer Kollision mit dem Rand geschehen soll. Hier brechen wir die Bewegung erst einmal einfach ab.

Wenn das Spiel so weit funktioniert, können Sie jetzt auch die Anweisungen für den Test entfernen beziehungsweise ändern. Lassen Sie im ersten Schritt den Hintergrund für das Rechteck zur Kollisionsprüfung transparent darstellen. Dadurch wird es unsichtbar.

Ergänzen Sie dazu die Anweisung

```
fuellung.Opacity = 0;
```

nach dem Zuweisen der Farbe.

Entfernen Sie dann die Schleife, die die Testäpfel erzeugt. Erstellen Sie danach in der Klasse für die Anwendung ein Feld für einen Apfel. Lassen Sie in der Methode zum Starten des Spiels einen neuen Apfel erzeugen und anzeigen. Bei einer Kollision der Schlange mit dem Apfel oder dem umgebenden Rechteck löschen Sie den Apfel und erzeugen direkt einen neuen. Die entsprechenden Anweisungen sind im Code 2.18 fett markiert.

```
...
//war es ein Apfel oder das Rechteck eines Apfels?
if (name == "Apfel" || name == "Kollision")
{
    //die Punkte erhöhen und neu anzeigen
    punkte = punkte + 10;
    punktAnzeige.Content = punkte;
    //ein Teil hinten an die Schlange anhängen
    //bitte in einer Zeile eingeben
    Schlangenteil sTeil = new Schlangenteil(new
        Point(schlange[schlange.Count - 1].LiefereAltePosition().X,
        schlange[schlange.Count - 1].LiefereAltePosition().Y +
        schlange[schlange.Count - 1].LiefereGroesse())),
        Colors.Black);
    schlange.Add(sTeil);
    //den alten Apfel löschen
    meinApfel.Entfernen(spieldfeld);
    //und einen neuen erzeugen
    meinApfel = new Apfel(Colors.Green, 20);
    meinApfel.Anzeigen(spieldfeld, balkenBreite);
}
...
...
```

Code 2.18: Das Erzeugen eines neuen Apfels bei der Kollision
(es handelt sich um ein Fragment)

Hinweis:

Der Bezeichner `meinApfel` im vorigen Code steht für ein Feld aus der Instanz der Klasse `Apfel`.

Ändern Sie dann noch die Anweisungen zum Erzeugen der Schlange in der Methode `Start()` so, dass die Schlange direkt nach dem Start nur aus dem Kopf besteht. Löschen Sie dazu einfach die komplette Schleife, die die Schlangenteile erzeugt.

2.7 Die Anzeige der Spielzeit

Jetzt fehlt nur noch die Anzeige der Spielzeit. Da wir später die Geschwindigkeit der Schlange anpassen werden, benutzen wir dazu einen eigenen Timer. Er macht nichts weiter, als jede Sekunde die Spielzeit zu erhöhen und anzuzeigen.

Vereinbaren Sie bitte ein neues Feld `timerSpielzeit` vom Typ `DispatcherTimer`. Weisen Sie diesem Feld im Konstruktor der Klasse für die Anwendung eine neue Instanz der Klasse `DispatcherTimer` zu und lassen Sie eine Methode `Timer_Spielzeit()` jede Sekunde ausführen. Die entsprechenden Anweisungen sehen so aus:

```
timerSpielzeit = new DispatcherTimer();
timerSpielzeit.Interval = TimeSpan.FromMilliseconds(1000);
timerSpielzeit.Tick += new EventHandler(Timer_Spielzeit);
timerSpielzeit.Start();
```

Code 2.19: Der Timer für die Spielzeit

Denken Sie auch daran, den Timer für die Spielzeit bei einer Kollision der Schlange mit der Spielfeldbegrenzung wieder zu stoppen.

Das Spiel sollte nun so aussehen:

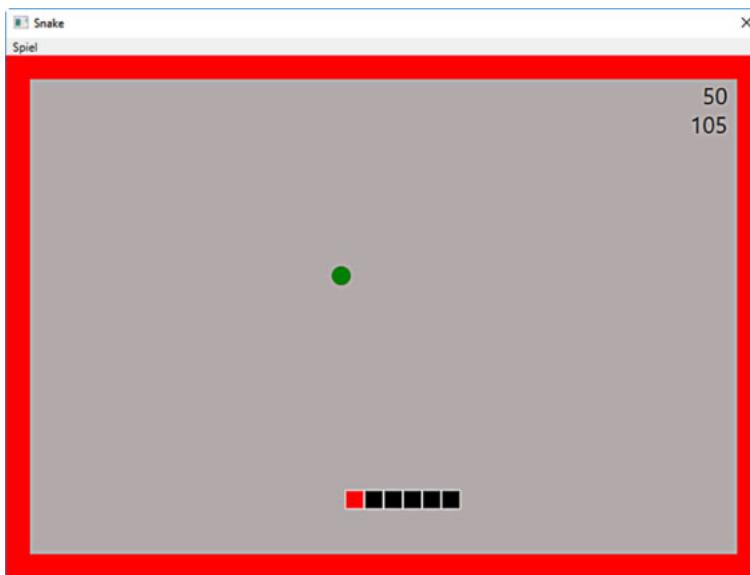


Abb. 2.10: Snake in der „Rohversion“

Wenn Sie das Spiel jetzt einige Male hintereinander starten, werden Sie allerdings feststellen, dass die Äpfel immer an denselben Positionen abgelegt werden. Das liegt daran, dass wir den Zufallsgenerator mit dem Hashcode initialisieren. Wenn Sie das stört, benutzen Sie einfach den Standardkonstruktor der Klasse `Random`. Denn wir müssen jetzt nicht mehr mehrere Äpfel in sehr kurzen Abständen erzeugen.

In den nächsten Kapiteln werden wir uns an die Funktionen zum Starten und Anhalten des Spiels machen. Außerdem ergänzen wir noch Spieleinstellungen und sorgen für ein wenig mehr Leben auf dem Spielfeld.

Zusammenfassung

Über die Eigenschaft **WindowsStartupLocation** können Sie bei einer WPF-Anwendung die Startposition eines Fensters festlegen.

Sie können für ein Steuerelement die aktuelle Breite und Höhe ermitteln. Das ist zum Beispiel wichtig, wenn Steuerelemente automatisch in der Größe angepasst werden und Sie selbst keine Größe angegeben haben.

Generics benutzen Platzhalter für die Werte, die von ihnen verarbeitet werden. So können Sie zum Beispiel mit einer generischen Methode nahezu beliebige Datentypen verarbeiten.

Mit dem generischen Typ `List` können Sie sehr einfach Listen für nahezu beliebige Datentypen erstellen. Die Liste wird beim Einfügen neuer Elemente automatisch vergrößert.

Die Klasse `Random` nutzt für die Initialisierung des Zufallsgenerators ohne weitere Angaben die Systemzeit.

Ein Hashcode ist ein numerischer Wert, über den Daten nahezu eindeutig identifiziert werden können.

Über die Methode `HitTest()` der Klasse `System.Windows.Media.VisualTreeHelper` können Sie bei der WPF prüfen, ob sich an einer bestimmten Position in einem bestimmten Objekt ein anderes Objekt befindet. Um auf die Eigenschaften des Objekts zuzugreifen, das getroffen wurde, müssen Sie das getroffene Objekt zunächst casten.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbene Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 2.1 Auf welchen Wert müssen Sie die Eigenschaft **WindowsStartupLocation** setzen, damit ein Fenster zentriert auf dem Bildschirm angezeigt wird?

- 2.2 Was geschieht, wenn Sie ein WPF-Steuerelement über die Toolbox einfügen und dann alle automatisch gesetzten Eigenschaften für die Positionierung im XAML-Code löschen? Wo wird das Steuerelement positioniert? Wie viel Platz nimmt es ein?

- 2.3 Mit welchen Eigenschaften ermitteln Sie die aktuelle Höhe und Breite eines Steuerelements?

- 2.4 Sie wollen eine generische Liste vom Typ `List` für `int`-Typen erstellen. Formulieren Sie eine entsprechende Anweisung. Der Bezeichner für die Liste ist beliebig.

- 2.5 Mit welcher Methode ergänzen Sie Werte am Ende einer generischen Liste?

- 2.6 Mit welcher Eigenschaft ermitteln Sie die Anzahl der Elemente in einer generischen Liste?

- 2.7 Wie können Sie die Initialisierung des Zufallsgenerators der Klasse `Random` steuern?

- 2.8 Wie ermitteln Sie den Hashcode eines Objekts?

- 2.9 Was liefert Ihnen die Methode `HitTest()` der Klasse `System.Windows.Media.VisualTreeHelper`?

- 2.10 Sie haben über die Methode `HitTest()` einen Treffer mit einem Objekt ermittelt. Formulieren Sie einen Ausdruck, um auf die Eigenschaft `Name` des getroffenen Objekts zuzugreifen.

3 Spielsteuerung und Bestenliste

In diesem Kapitel werden wir uns um die Spielsteuerung und die Bestenliste kümmern. Dabei setzen wir viele Techniken ein, die wir in ähnlicher Form bereits im Pong-Spiel verwendet haben.

Beginnen wir mit einigen Vorüberlegungen.

3.1 Vorüberlegungen

Beim Starten und Anhalten müssen wir fünf verschiedene Aktionen unterscheiden können:

- 1) Das Spiel wird das erste Mal neu gestartet.
- 2) Bei einem laufenden Spiel soll ein neues Spiel gestartet werden.
- 3) Das Spiel wird unterbrochen.
- 4) Das Spiel wird nach einer Unterbrechung fortgesetzt.
- 5) Das Spiel wird beendet, weil die Schlange mit der Spielfeldbegrenzung oder sich selbst kollidiert ist.

Die ersten vier Aktionen starten wir über das Menü **Spiel** mit den Einträgen **Neues Spiel** und **Pause**. Der Eintrag **Pause** soll wieder wie ein Schalter arbeiten – also das Spiel entweder anhalten, wenn es gerade läuft, oder das Spiel fortsetzen, wenn es aktuell unterbrochen wurde. Außerdem soll der Spieler die Funktion **Pause** über die Tastatur aufrufen können. Dazu erstellen wir später ein eigenes Command.

Über den Eintrag **Neues Spiel** soll nach einer Abfrage ein neues Spiel gestartet werden können – entweder direkt nach dem Aufruf des Programms oder während eines laufenden Spiels.

Für das Ende des Spiels erstellen wir eine eigene Methode, die wir bei einer Kollision der Schlange mit der Spielfeldbegrenzung oder sich selbst aufrufen.

So viel erst einmal zur Steuerung des Spiels.

Bei den Spieleinstellungen bieten wir lediglich eine Veränderung des Schwierigkeitsgrads an. Dazu ändern wir die Größe des Spielfelds. Je kleiner das Spielfeld ist, desto mehr Punkte bekommt der Spieler für einen „gefressenen“ Apfel. Die Auswahl nehmen wir über ein Menü ähnlich wie im Pong-Spiel vor.

Bei der Bestenliste greifen wir wieder auf unsere Klasse `Score` zurück, die wir ja bereits im Pong-Spiel verwendet haben. Änderungen nehmen wir vor allem bei der Ausgabe der Liste vor. Hier erstellen wir ein Grid, das wir mit den Daten aus dem Array der Bestenliste füllen.

Beginnen wir jetzt mit der praktischen Umsetzung.

3.2 Die Spielsteuerung

Vereinbaren Sie ein neues Feld `spielUnterbrochen` vom Typ `bool` in der Klasse der Anwendung. Es soll speichern, ob das Spiel aktuell unterbrochen ist oder nicht. Setzen Sie das Feld im Konstruktor der Anwendung auf `true`. Löschen Sie dann die Anweisungen zum Starten der beiden Timer aus dem Konstruktor.

Der vollständige Konstruktor für die Anwendung sollte jetzt so aussehen:

```
public MainWindow()
{
    InitializeComponent();
    //die Liste für die Schlange erzeugen
    schlange = new List<Schlangenteil>();

    //der Timer für die Schlange
    //die Instanz erzeugen
    timerSchlange = new DispatcherTimer();
    //das Intervall setzen
    timerSchlange.Interval = TimeSpan.FromMilliseconds(1000);
    //die Methode für das Ereignis zuweisen
    timerSchlange.Tick += new EventHandler(Timer_SchlangeBewegen);

    //der Timer für die Spielzeit
    //die Instanz erzeugen
    timerSpielzeit = new DispatcherTimer();
    //das Intervall setzen
    timerSpielzeit.Interval = TimeSpan.FromMilliseconds(1000);
    //die Methode für das Ereignis zuweisen
    timerSpielzeit.Tick += new EventHandler(Timer_Spielzeit);

    //das Spiel ist erst einmal angehalten
    spielUnterbrochen = true;
}
```

Code 3.1: Der erweiterte Konstruktor für die Anwendung

Legen Sie dann die Einträge **Neues Spiel** und **Pause** im Menü **Spiel** an. Als Bezeichner benutzen wir `spielPause` und `spielNeuesSpiel`. Wenn Sie möchten, können Sie unterhalb des Eintrags **Pause** auch noch eine Trennlinie setzen.

Für den Eintrag **Pause** setzen Sie bitte die Eigenschaft **IsCheckable** auf `True`, damit wir ihn mit einem Häkchen markieren können. Da wir die Methode zum Unterbrechen des Spiels später auch ohne das Anklicken eines Menüeintrags ausführen wollen, legen wir eine eigene Methode `SpielPause()` an. Diese Methode rufen wir dann beim Anklicken des Menüeintrags **Pause** auf.

Die Anweisungen für die Methode `SpielPause()` finden Sie im folgenden Code 3.2:

```
//die Methode unterbricht das Spiel oder setzt es fort
void SpielPause()
{
    //erst einmal prüfen wir den Status
    //läuft das Spiel?
    if (spielUnterbrochen == false)
```

```

    {
        //alle Timer anhalten
        timerSchlange.Stop();
        timerSpielzeit.Stop();
        //die Markierung im Menü einschalten
        spielPause.IsChecked = true;
        //den Text in der Titelleiste ändern
        Title = "Snake - Das Spiel ist angehalten!";
        spielUnterbrochen = true;
    }
    else
    {
        //alle Timer wieder starten
        timerSchlange.Start();
        timerSpielzeit.Start();
        //die Markierung im Menü abschalten
        spielPause.IsChecked = false;
        //den Text in der Titelleiste ändern
        Title = "Snake";
        spielUnterbrochen = false;
    }
}

```

Code 3.2: Die Methode SpielPause()

Je nach Wert des Felds `spielUnterbrochen` stoppen beziehungsweise starten wir die Timer und setzen die Anzeige des Häkchens sowie des Titels. Das kennen Sie ja bereits vom Pong-Spiel.

Die Methode für das Anklicken des Menüeintrags **Pause** sieht dann so aus:

```

private void SpielPause_Click(object sender, RoutedEventArgs e)
{
    SpielPause();
}

```

Code 3.3: Die Methode für das Klicken auf den Menüeintrag **Pause**

Sorgen Sie nun dafür, dass der Spieler in einer Spielpause nicht versehentlich die Richtung der Schlange verändern kann. Ergänzen Sie dazu vor den Abfragen für die Änderung der Richtung eine Abfrage, die die Methode wieder verlässt, wenn das Spiel unterbrochen ist.

Kommen wir zum Starten eines Spiels. Auch hier gehen wir ähnlich vor wie beim Pong-Spiel. Wir prüfen zunächst, ob wir ein laufendes Spiel unterbrechen müssen, und fragen dann über eine Methode `NeuesSpiel()` ab, ob ein neues Spiel gestartet werden soll. Diese Methode und die Methode für das Anklicken des Menüeintrags **Neues Spiel** finden Sie in den folgenden Codes:

```

//erzeugt einen Dialog zum Neustart und liefert das Ergebnis
//zurück
bool NeuesSpiel()
{
    bool ergebnis = false;
    //einen Dialog mit Ja/Nein erzeugen

```

```

//bitte in einer Zeile eingeben
MessageBoxResult abfrage = MessageBox.Show("Wollen Sie ein
neues Spiel starten?", "Neues Spiel", MessageBoxButtons.YesNo,
MessageBoxImage.Question);
//wurde auf Ja geklickt?
if (abfrage == MessageBoxResult.Yes)
{
    //dann rufen wir die Methode Start() auf
    Start();
    ergebnis = true;
}
return ergebnis;
}

```

Code 3.4: Die Methode NeuesSpiel()

```

private void SpielNeuesSpiel_Click(object sender, RoutedEventArgs
e)
{
    //läuft gerade ein Spiel?
    //dann halten wir es an
    if (spielUnterbrochen == false)
    {
        SpielPause();
        //den Dialog anzeigen
        NeuesSpiel();
        //und weiter spielen
        SpielPause();
    }
    //wenn kein Spiel läuft, starten wir ein neues, wenn im Dialog
    //auf Ja geklickt wurde
    else
        if (NeuesSpiel() == true)
    {
        SpielPause();
    }
}

```

Code 3.5: Die Anweisungen für das Anklicken des Menüeintrags **Neues Spiel**

Damit nicht direkt nach dem Start der Anwendung ein Apfel gezeichnet und ein Schlangenkopf erstellt wird, löschen Sie bitte den Aufruf der Methode `Start()` aus der Methode `Window_Loaded()` der Anwendung.

Hinweis:

Falls Sie die Balkenbreite in der Methode `Start()` setzen, verschieben Sie die Anweisung bitte in den Konstruktor. Sonst funktioniert gleich das Zeichnen nicht mehr.

Leeren Sie außerdem in der Methode `Start()` vor dem Erzeugen des Schlangenkopfes die Liste für die Schlange. Andernfalls würde eine alte Schlange verlängert beziehungsweise bei jedem Neustart ein weiterer Schlangenkopf angehängt. Für das Löschen verwenden Sie die Anweisung

```
schlange.Clear();
```

Entfernen Sie dann auch noch mit der Anweisung

```
spieldorf.Children.Clear();
```

alle Objekte aus dem Spielfeld. Sonst würden bei jedem Neustart ein zusätzlicher Apfel und ein zusätzlicher Schlangenkopf angezeigt.

Da beim Löschen auch die Spielfeldbegrenzungen verschwinden, lassen Sie sie mit der Methode `zeichneSpielfeld()` neu erstellen.

Hinweis:

Die Methode `zeichneSpielfeld()` wird beim ersten Start doppelt ausgeführt. Das nehmen wir aber bewusst in Kauf. Andernfalls bliebe das Spielfeld ja nach dem Start nahezu komplett leer.

Sorgen Sie anschließend dafür, dass die Funktion **Pause** nur dann aufgerufen werden kann, wenn ein Spiel gestartet wurde. Deaktivieren Sie dazu den Menüeintrag im Konstruktor der Anwendung und aktivieren Sie ihn, wenn ein Spiel gestartet wurde. Dazu setzen Sie die Eigenschaft **IsEnabled** auf `false` beziehungsweise auf `true`.

Jetzt fehlt nur noch das Beenden des Spiels, wenn die Schlange einen Kontakt mit einem Hindernis hatte. Dazu erstellen wir eine Methode `spielEnde()`, die eine Meldung anzeigt und dann über unsere Methode `NeuesSpiel()` abfragt, ob sofort ein neues Spiel gestartet werden soll. Die Methode finden Sie im folgenden Code 3.6:

```
void SpielEnde()
{
    //das Spiel anhalten
    SpielPause();
    //eine Meldung anzeigen
    //bitte in einer Zeile eingeben
    MessageBox.Show("Schade.", "Spielende", MessageBoxButtons.OK,
    MessageBoxIcon.Information);
    //Abfrage, ob ein neues Spiel gestartet werden soll
    if (NeuesSpiel() == true)
    {
        Start();
        //das Spiel "fortsetzen"
        SpielPause();
    }
    else
        //sonst beenden
        Close();
}
```

Code 3.6: Das Beenden des Spiels

Rufen Sie die Methode `spielEnde()` dann in der Methode `KollisionPruefen()` bei der Kollision der Schlange mit sich selbst oder einer Begrenzung auf.

Damit sind auch die Funktionen zum Starten und Anhalten des Spiels komplett. Kümmern wir uns nun um den Schwierigkeitsgrad.

Hier benutzen wir drei verschiedene Einstellungen – nämlich einfach, mittel und schwer. Je nach Einstellung verändern sich die Größe des Spielfelds und die Punkte, die ein Spieler für den Kontakt mit einem Apfel erhält. Die Einstellungen des Schwierigkeitsgrads erfolgen wie beim Pong-Spiel über ein Menü, das für jeden der drei Schwierigkeitsgrade einen eigenen Eintrag anbietet.

Vereinbaren Sie bitte zunächst ein neues Feld `punkteMehr` vom Typ `int` für die Klasse der Anwendung, damit wir die Punkte, die addiert werden, flexibel festlegen können. Setzen Sie den Wert dieses Felds im Konstruktor auf `10` für die Spieleinstellung **Mittel**. Passen Sie die Anweisung zum Verändern der Punkte in der Methode `KollisionPruefen()` dann so an, dass nicht mehr fest `10` Punkte addiert werden, sondern der Wert des Felds `punkteMehr`.

Erstellen Sie anschließend ein Menü **Einstellungen** mit einem Untermenü **Schwierigkeitsgrad**. Legen Sie in diesem Untermenü drei Einträge für die unterschiedlichen Schwierigkeitsgrade an und sorgen Sie dafür, dass die Einträge markiert werden können. Für den Eintrag **Mittel** setzen Sie bitte auch die Markierung.

Im letzten Schritt müssen Sie nun noch die Methoden für das Anklicken jedes einzelnen Menüeintrags erstellen. Hier schalten Sie zunächst die Markierung der anderen Menüeinträge ab, ändern die Spielfeldgröße und rufen dann die Methode `ZeichneSpielfeld()` auf. Um das Markieren des Menüeintrags müssen Sie sich – anders als beim Pong-Spiel – nicht selbst kümmern. Das erfolgt automatisch beim Anklicken des Eintrags.

Da sich die Anweisungen beim Anklicken der Menüeinträge vor allem durch die Werte, die gesetzt werden, unterscheiden, erstellen wir uns wieder eine Methode `SetzeEinstellungen()`, die die Werte als Argument erhält.

Die Methoden für das Ändern des Schwierigkeitsgrads finden Sie im Code 3.7:

```
private void SchwierigkeitEinfach_Click(object sender,
RoutedEventArgs e)
{
    //die Markierungen bei den anderen Einträgen abschalten
    schwierigkeitMittel.IsChecked = false;
    schwierigkeitSchwer.IsChecked = false;
    SetzeEinstellungen(1000, 800, 1);
}

private void SchwierigkeitMittel_Click(object sender,
RoutedEventArgs e)
{
    //die Markierungen bei den anderen Einträgen abschalten
    schwierigkeitEinfach.IsChecked = false;
    schwierigkeitSchwer.IsChecked = false;
    SetzeEinstellungen(800, 600, 10);
}

private void SchwierigkeitSchwer_Click(object sender,
RoutedEventArgs e)
{
    //die Markierungen bei den anderen Einträgen abschalten
    schwierigkeitEinfach.IsChecked = false;
```

```

        schwierigkeitMittel.IsChecked = false;
        SetzeEinstellungen(400, 300, 25);
    }

    void SetzeEinstellungen(int breite, int hoehe, int punkteNeu)
    {
        //die Größe des Fensters setzen
        Width = breite;
        Height = hoehe;
        //die Punkte setzen
        punkteMehr = punkteNeu;
        //Fenster neu positionieren
        Left = (SystemParameters.PrimaryScreenWidth - Width) / 2;
        Top = (SystemParameters.PrimaryScreenHeight - Height) / 2;
        //die Elemente im Spielfeld löschen
        spielfeld.Children.Clear();
        //das Spielfeld neu erstellen
        ZeichneSpielfeld();
    }
}

```

Code 3.7: Die Methoden für das Ändern des Schwierigkeitsgrads

Hinweis:

Die genauen Namen der Methoden für das Anklicken der Menüeinträge hängen von den Namen ab, die Sie in Ihrem Projekt verwenden.

Neu ist hier vor allem das Positionieren des Fensters nach dem Ändern der Größe. Bei der WPF verwenden wir dazu die Eigenschaften

`SystemParameters.PrimaryScreenWidth` beziehungsweise

`SystemParameters.PrimaryScreenHeight`. Sie liefern uns die Breite beziehungsweise die Höhe der aktuellen Bildschirmauflösung.

Damit die Spielsteuerung nicht durcheinandergerät, sollten Sie außerdem noch dafür sorgen, dass die Einstellungen nur dann geändert werden können, wenn gerade kein Spiel läuft. Dazu können Sie zum Beispiel die Einträge im Menü **Einstellungen** deaktivieren, wenn ein Spiel gestartet wurde.

Damit sind auch die Einstellungen komplett. Im nächsten Schritt binden wir die Bestenliste ein.

3.3 Die Bestenliste

Da unsere Bestenliste in der Klasse `Score` für eine Windows-Forms-Anwendung erstellt wurde, müssen wir einige Änderungen vornehmen. Sie betreffen das Beschaffen eines neuen Eintrags über einen Dialog und die Ausgabe der Liste.

Binden Sie im ersten Schritt bitte zunächst die Datei `Score.cs` mit der Klasse `Score` in das Projekt ein. Benutzen Sie dazu wie gewohnt die Funktion **Projekt/Vorhandenes Element hinzufügen...** Ändern Sie danach den Namensraum der Klasse in `Snake`.

Hinweis:

Für Klassen, die Sie häufig wiederverwenden wollen, sollten Sie einen neutralen Namensraum wählen beziehungsweise einen Namensraum, der zur Klasse passt. Dann müssen Sie den Namensraum im Code nicht ständig beim Einbinden in ein anderes Projekt anpassen. Bitte denken Sie dann aber daran, dass Sie den Namensraum beim Zugriff bekannt machen müssen.

Achten Sie bitte darauf, dass Sie die Version der Bestenliste verwenden, die die Liste in einer Datei verwaltet.

Der neue Dialog für das Beschaffen des Namens ist schnell erstellt. Erstellen Sie mit der Funktion **Projekt/Fenster hinzufügen...** ein neues Fenster mit dem Namen `NameDialog`. Fügen Sie in dieses Fenster ein Label, ein Eingabefeld und eine Schaltfläche ein. Passen Sie die Texte in den Steuerelementen so an wie auch im ursprünglichen Formular für die Windows-Forms-Anwendung. Für die Schaltfläche können Sie die Eigenschaft `IsDefault` aktivieren.

Setzen Sie außerdem die Eigenschaften für das Fenster wie in der Tab. 3.1.

Tab. 3.1: Die Eigenschaften für den Eingabedialog

Eigenschaft	Wert
<code>WindowStyle</code>	<code>ToolWindow</code>
<code>Title</code>	Herzlichen Glückwunsch!
<code>Topmost</code>	Ja
<code>WindowStartupLocation</code>	<code>CenterOwner</code>
<code>ResizeMode</code>	<code>NoResize</code>

Verkleinern Sie das Fenster dann noch ein wenig. Es sollte danach ungefähr so aussehen:

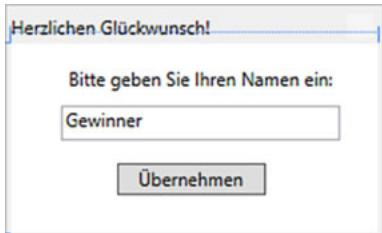


Abb. 3.1: Das Formular zur Eingabe des Namens

Beim Anklicken der Schaltfläche setzen Sie die Eigenschaft `DialogResult` für das Fenster auf `true`. Erstellen Sie außerdem noch eine öffentliche Methode `LiefereName()`, die den Inhalt des Eingabefelds liefert. Die beiden Methoden finden Sie im Code 3.8:

```
private void ButtonUebernehmen_Click(object sender,
RoutedEventArgs e)
{
    DialogResult = true;
}

public string LiefereName()
{
    return textBox.Text;
}
```

Code 3.8: Die Methoden für das Fenster NameDialog

Hinweis:

Die Bezeichner im vorigen Code müssen Sie an die Bezeichner in Ihrem Projekt anpassen.

Passen Sie dann die Anweisungen in der Methode `NeuerEintrag()` der Klasse `Score` so an, dass die Abfrage auf die Eigenschaft `DialogResult` des Dialogs erfolgt und der Eigentümer für den Dialog gesetzt wird. Den Eigentümer übergeben wir dabei über einen Parameter vom Typ `System.Windows.Window` an die Methode.

Die Anweisung zum Schließen des modalen Fensters können Sie löschen. Sie ist ja bei einer WPF-Anwendung nicht erforderlich. Die geänderte Methode sollte so aussehen:

```
//ist ein neuer Eintrag erreicht?
public bool NeuerEintrag(System.Windows.Window fenster)
{
    string tempName = string.Empty;
    //wenn die aktuelle Punktzahl größer ist als der letzte
    Eintrag
    //der Liste, wird der letzte Eintrag der Liste überschrieben
    //und die Liste neu sortiert
    if (punkte > bestenliste[anzahl - 1].GetPunkte())
    {
        //den Namen beschaffen
        NameDialog neuerName = new NameDialog();
        //den "Eigentümer" setzen
        neuerName.Owner = fenster;
        //den Dialog modal anzeigen
        neuerName.ShowDialog();
        if (neuerName.DialogResult == true)
            tempName = neuerName.LiefereName();
        bestenliste[anzahl - 1].SetzeEintrag(punkte, tempName);
        //neu sortieren
        Array.Sort(bestenliste);
        //und speichern
        SchreibePunkte();
    }
    return true;
}
```

```

    }
else
    return false;
}

```

Code 3.9: Die geänderte Methode NeuerEintrag()

Vereinbaren Sie dann in der Klasse für die Anwendung ein Feld vom Typ `Score` mit dem Namen `spielpunkte` und erzeugen Sie im Konstruktor der Klasse für die Anwendung über dieses Feld eine neue Instanz der Klasse `Score`. Dabei gibt es keine Besonderheiten.

Lassen Sie anschließend beim Kontakt der Schlange mit einem Apfel die Punkte über die Methode `VeraenderePunkte()` der Klasse `Score` anpassen. Ersetzen Sie dazu in der Methode `KollisionPruefen()` die Anweisung

```
punkte = punkte + punkteMehr;
```

durch die Anweisung

```
punkte = spielpunkte.VeraenderePunkte(punkteMehr);
```

Damit die Punkte bei einem Neustart korrekt gesetzt werden, rufen Sie außerdem in der Methode `Start()` die Methode `spielpunkte.LoeschePunkte()` auf.

Passen Sie dann noch die Anweisung zum Beschaffen des Dateipfads für die Datei an. Ersetzen Sie hier die Eigenschaft

`System.Windows.Forms.Application.StartupPath` durch

`System.AppDomain.CurrentDomain.BaseDirectory`⁴. Dieser Ausdruck beschafft ebenfalls den Ordner, in dem die Anwendung ausgeführt wird.

In der Methode `SpielEnde()` der Anwendung prüfen Sie nach dem Anzeigen der Meldung zum Ende des Spiels, ob ein neuer Eintrag in der Bestenliste möglich ist. Da das Ausgeben der Liste zahlreiche Änderungen erforderlich macht, lassen wir zunächst einfach eine Meldung anzeigen. Die zusätzlichen Anweisungen in der Methode `SpielEnde()` könnten also so aussehen:

```
//reicht es für einen Eintrag in der Bestenliste?
if (spielpunkte.NeuerEintrag(this) == true)
    MessageBox.Show("Kompliment.");
```

Übernehmen Sie jetzt alle Änderungen und kommentieren Sie die Methode zur Anzeige der Liste in der Klasse `Score` zunächst aus. Testen Sie die Bestenliste dann. Wenn Sie genügend Punkte sammeln, sollten Sie sich in die Liste eintragen können.

Kommen wir nun zum Anzeigen der Liste. Hier gehen wir einen anderen Weg als in der Windows-Forms-Anwendung. Wir erstellen ein eigenes Fenster für die Anzeige, die die Daten in einem Grid in Labels anzeigt. Für das Grid erstellen wir zunächst nur ein Gerüst mit zwei Spalten und einer Zeile. In dieser Zeile zeigen wir die Überschrift an. Die restlichen Zeilen mit den Einträgen aus der Bestenliste ergänzen wir dynamisch bei der Anzeige des Fensters.

4. *Base directory* bedeutet übersetzt so viel wie „Basisverzeichnis“.

Legen Sie bitte mit der Funktion **Projekt/Fenster hinzufügen...** ein neues Fenster an. Wir verwenden in unserem Beispiel den Namen **Bestenliste**. Setzen Sie außerdem die Eigenschaften für das Fenster wie in der folgenden Tab. 3.2.

Tab. 3.2: Die Eigenschaften für das Fenster der Bestenliste

Eigenschaft	Wert
Title	Bestenliste
Topmost	Ja
WindowStartupLocation	CenterOwner
ResizeMode	NoResize
Height	400
Width	250

Passen Sie das Grid im Fenster so an, wie im folgenden XAML-Code beschrieben.

```
<Grid x:Name="meinGrid">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <!-- bitte in einer Zeile eingeben -->
    <Label Grid.ColumnSpan="2" Content="Bestenliste"
        HorizontalAlignment="Center" FontSize="22 pt"
        FontWeight="Bold"/>
</Grid>
```

Code 3.10: Der XAML-Code für das Grid der Bestenliste

Hier vergeben wir zunächst einen Namen an das Grid, damit wir es gleich aus der Code-Behind-Datei ansprechen können. Danach definieren wir zwei Zeilen mit automatischer Höhe und zwei Spalten. Die erste Spalte soll $\frac{1}{3}$ des Platzes einnehmen und die zweite Spalte $\frac{2}{3}$. Zum Schluss setzen wir in die erste Zeile zentriert als Überschrift ein Label mit dem Text **Bestenliste**.

Das Fenster sollte nun ungefähr so aussehen:

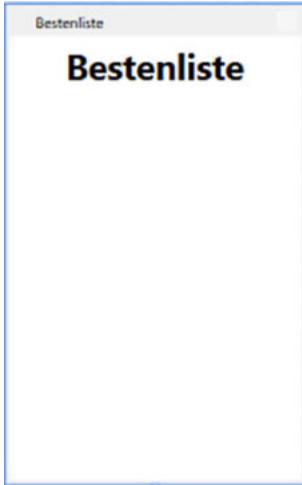


Abb. 3.2: Das Fenster für die Bestenliste

Damit wir bei der Ausgabe etwas flexibler sind, übergeben wir nicht einzelne Einträge aus der Bestenliste, sondern einfach eine Liste mit Zeichenketten. Diese Liste erstellen wir gleich vor der Anzeige der Bestenliste und reichen sie an den Konstruktor der Klasse `Bestenliste` weiter.

Im Konstruktor der Klasse `Bestenliste` erstellen wir für jeden Eintrag ein Label, das wir dann zum Grid hinzufügen. Nach jedem zweiten Eintrag legen wir außerdem eine neue Zeile an. Den kompletten Konstruktor finden Sie im Code 3.11:

```
public Bestenliste(List<string> eintraege)
{
    InitializeComponent();
    //zum Zählen der Zeilen und Spalten
    int zaehlerZeile = 1;
    int zaehlerSpalte = 0;
    //die Einträge in der Liste verarbeiten
    foreach (string zeichenkette in eintraege)
    {
        //ein neues Label mit der Zeichenkette erzeugen
        Label meinLabel = new Label();
        meinLabel.Content = zeichenkette;
        //und im Grid positionieren
        Grid.SetRow(meinLabel, zaehlerZeile);
        Grid.SetColumn(meinLabel, zaehlerSpalte);
        meinGrid.Children.Add(meinLabel);
        //die Spalte erhöhen
        zaehlerSpalte++;
        //haben wir zwei Spalten gefüllt?
        //dann fügen wir eine neue Zeile ein und fangen wieder von
        //vorne an
        if (zaehlerSpalte == 2)
        {
            zaehlerSpalte = 0;
```

```
        zaehlerZeile++;
        meinGrid.RowDefinitions.Add(new RowDefinition());
    }
}
```

Code 3.11: Der Konstruktor der Klasse Bestenliste

Mit den Anweisungen

```
Label meinLabel = new Label();
meinLabel.Content = zeichenkette;
Grid.SetRow(meinLabel, zahlerZeile);
Grid.SetColumn(meinLabel, zahlerSpalte);
meinGrid.Children.Add(meinLabel);
```

erzeugen wir das Label und setzen es in das Grid. Die Position wird dabei durch die Variablen `zaehlerZeile` und `zaehlerSpalte` und die Methoden `Grid.SetRow()` beziehungsweise `Grid.SetColumn()` bestimmt.

Nach dem Einfügen von zwei Labels setzen wir den Zähler für die Spalten wieder auf 0, erhöhen den Zähler für die Zeilen um 1 und legen mit der Anweisung

```
meinGrid.RowDefinitions.Add(new RowDefinition());
```

eine neue Zeile an. Dabei übergeben wir eine Instanz der Klasse `RowDefinition` mit den Standardwerten. Sie legt eine neue Zeile mit automatischer Höhe an.

Das Anzeigen der Liste lassen wir wieder durch die Methode `ListeAusgeben()` der Klasse `Score` erledigen. Sie sieht nach den Änderungen so aus:

```
public void ListeAusgeben(System.Windows.Window fenster)
{
    //wir erstellen uns der Einfachheit halber eine Liste vom Typ
    //string
    List<string> eintraege = new List<string>();
    for (int i = 0; i < anzahl; i++)
    {
        eintraege.Add(Convert.ToString(bestenliste[i].GetPunkte()));
        eintraege.Add(bestenliste[i].GetName());
    }
    //die Liste anzeigen
    Bestenliste listenAnzeige = new Bestenliste(eintraege);
    //den "Eigentümer" setzen
    listenAnzeige.Owner = fenster;
    //den Dialog modal anzeigen
    listenAnzeige.ShowDialog();
}
```

Code 3.12: Die geänderte Methode `ListeAusgeben()`

Hier erzeugen wir zunächst eine Liste vom Typ `string` und hängen die Einträge aus der Liste der Reihe nach am Ende an. Dann erzeugen wir eine Instanz für das Fenster mit der Bestenliste und übergeben dabei die Liste mit den Einträgen. Anschließend setzen wir den „Eigentümer“ für das Fenster und zeigen modal an. Dabei gibt es keine Besonderheiten.

Hinweis:

Denken Sie daran, dass sich die Klasse `List` im Namensraum `System.Collections.Generic` befindet. In der Klasse `Score` müssen Sie diesen Namensraum selbst einbinden.

Erstellen Sie jetzt noch einen Eintrag **Bestenliste** im Menü **Spiel** und lassen Sie die Liste beim Anklicken dieses Eintrags anzeigen. Denken Sie dabei daran, das Spiel vorher gegebenenfalls zu unterbrechen. Sorgen Sie außerdem dafür, dass die Bestenliste angezeigt wird, nachdem ein neuer Eintrag vorgenommen werden konnte.



Abb. 3.3: Die Bestenliste im Einsatz

Richtig spannend ist das Spiel bisher aber nicht – auch nicht bei der schwierigsten Einstellung. Die Schlange bewegt sich recht behäbig und strapaziert eher die Geduld des Spielers als seine Geschicklichkeit. Wir werden daher im nächsten Kapitel unter anderem dafür sorgen, dass die Schlange automatisch immer schneller wird.

Zusammenfassung

Sie können alle Elemente in einer Liste vom Typ `List` mit der Methode `Clear()` gleichzeitig löschen.

Sie können die Position eines Elements in einem Grid auch aus der Code-Behind-Datei festlegen.

Zeilen und Spalten in einem Grid lassen sich auch zur Laufzeit dynamisch erzeugen.

Aufgaben zur Selbstüberprüfung

- 3.1 Mit welchen Eigenschaften können Sie die Breite und die Höhe der aktuellen Bildschirmauflösung abfragen?

- 3.2 Mit welcher Eigenschaft können Sie sich den Ordner beschaffen, in dem eine WPF-Anwendung ausgeführt wird?

- 3.3 Sie wollen ein Label mit dem Namen `meinLabel` in einem Grid mit dem Namen `meinGrid` anzeigen. Es soll in die dritte Spalte in der fünften Zeile gesetzt werden. Formulieren Sie entsprechende C#-Anweisungen.

- 3.4 Sie wollen in einem Grid mit dem Namen `meinGrid` dynamisch zur Laufzeit eine neue Zeile mit den Standardeinstellungen ergänzen. Notieren Sie die entsprechende Anweisung.

4 Ein wenig Feinschliff

In diesem Kapitel verpassen wir unserem Spiel noch ein wenig Feinschliff. Wir sorgen dafür, dass die Schlange automatisch immer schneller wird, bauen eine Tastatursteuerung für die Pausenfunktion ein und erstellen einen Startbildschirm.

Beginnen wir mit der automatischen Beschleunigung der Schlange.

4.1 Automatische Beschleunigung der Schlange

Dazu setzen wir das Intervall für den Timer der Schlange nicht mehr fest, sondern regeln es über ein Feld. Den Wert des Felds reduzieren wir alle 50 Punkte um 100. Das führen wir so lange durch, bis wir 100 als Wert für das Intervall des Timers erreicht haben.

Vereinbaren Sie bitte im ersten Schritt in der Klasse der Anwendung ein Feld vom Typ int. Wir verwenden den Namen `geschwindigkeit`. Setzen Sie dieses Feld im Konstruktor und in der Methode `start()` auf den Wert 1000. Passen Sie dann die Anweisung für das Setzen des Intervalls für den Timer der Schlange so an, dass nicht mehr der feste Wert 1000 benutzt wird, sondern das Feld `geschwindigkeit`.

Die Änderung der Geschwindigkeit nehmen wir in der Methode `KollisionPruefen()` vor. Hier prüfen wir, ob sich die aktuelle Punktzahl ohne Rest durch 50 teilen lässt. Wenn das der Fall ist und der Wert von `geschwindigkeit` größer als 100 ist, ziehen wir von `geschwindigkeit` 100 ab. Die entsprechenden Anweisungen sind im folgenden Fragment fett markiert.

```
...
//war es ein Apfel oder das Rechteck eines Apfels?
if (name == "Apfel" || name == "Kollision")
{
    //die Punkte erhöhen und neu anzeigen
    punkte = spielpunkte.VeraenderePunkte(punkteMehr);
    punktAnzeige.Content = punkte;
    //die Geschwindigkeit erhöhen, wenn nicht schon das Maximum
    //erreicht ist
    if (punkte % 50 == 0 && geschwindigkeit > 100)
    {
        geschwindigkeit = geschwindigkeit - 100;
        //bitte in einer Zeile eingeben
        timerSchlange.Interval =
            TimeSpan.FromMilliseconds(geschwindigkeit);
    }
    //ein Teil hinten an die Schlange anhängen
    Schlangenteil sTeil = new Schlangenteil(new
        Point(schlange[ schlange.Count - 1 ].LiefereAltePosition().X,
        schlange[ schlange.Count - 1 ].LiefereAltePosition().Y +
        schlange[ schlange.Count - 1 ].LiefereGroesse()),
        Colors.Black);
}
...
```

Code 4.1: Das Erhöhen der Geschwindigkeit

4.2 Tastatursteuerung für die Pausenfunktion

Kommen wir nun zur Tastatursteuerung für die Pausenfunktion. Die einfachste Variante wäre es, die Tastaturabfrage in der Methode `Window_KeyDown()` so zu erweitern, dass beim Drücken der Taste das Spiel angehalten beziehungsweise fortgesetzt wird. Sie müssten lediglich darauf achten, dass Sie diese Abfrage direkt zu Beginn der Methode durchführen, damit Sie auch ein unterbrochenes Spiel wieder fortsetzen können.

Wir gehen in unserem Beispiel aber einen anderen Weg und erstellen ein eigenes Command. Das ist zwar deutlich aufwendiger, erlaubt es aber, dass wir die Funktion im Menü und die Funktion, die wir über die Tastatur aufrufen, sehr gut zusammenfassen können. Denn wir weisen beiden Aktionen dasselbe Command zu und müssen dann Änderungen bei Bedarf nur noch an einer Stelle vornehmen.

Um ein eigenes Command zu erstellen, gehen Sie grundsätzlich in den folgenden Schritten vor:

1. Sie legen für jedes Command ein statisches Feld vom Typ `RoutedCommand` an. Damit Sie auf dieses Feld im XAML-Code zugreifen können, muss es nach außen sichtbar sein.
2. Für jedes Command legen Sie über die Eigenschaften `CanExecute` und `Executed` die Methoden für die Steuerung beziehungsweise das Ausführen des Commands fest.
3. Sie legen die beiden Methoden für das Command an. Die Methode für die Eigenschaft `CanExecute` muss dabei die Eigenschaft `e.CanExecute` auf `true` setzen, wenn das Command ausgeführt werden kann.
4. Sie legen das Command bei der Eigenschaft `CommandBindings` eines Objekts fest.
5. Sie ordnen das Command einem Steuerelement zu.
6. Wenn Sie eine Taste für das Ausführen des Commands verwenden wollen, müssen Sie dem Command diese Taste außerdem noch über die Eigenschaft `InputBindings` eines Objekts zuweisen.

Einige dieser Schritte kennen Sie bereits von unserem selbst programmierten Browser. Neu sind vor allem das Erstellen des öffentlichen statischen Felds und die Zuordnung einer eigenen Taste.

Schauen wir uns die einzelnen Schritte jetzt in der Praxis in unserem Spiel an.

Vereinbaren Sie bitte im ersten Schritt in der Klasse der Anwendung ein statisches Feld vom Typ `RoutedCommand` und weisen Sie diesem Feld eine neue Instanz der Klasse `RoutedCommand` zu. Die entsprechende Anweisung kann so aussehen:

```
static RoutedCommand pause = new RoutedCommand();
```

Legen Sie dann eine öffentliche Eigenschaft an, die den Wert dieses Felds liefert. Die Vereinbarung dieser Eigenschaft könnte so aussehen:

```
public static RoutedCommand Pause
{
    get
    {
        return pause;
    }
}
```

Code 4.2: Die Eigenschaft für den Zugriff auf das Feld

Hinweis:

Es hat sich eingebürgert, für Eigenschaften, die lediglich dem Zugriff auf ein geschütztes Feld dienen, den Namen des Felds mit einem Großbuchstaben zu Beginn zu verwenden.

Sie könnten das Feld auch direkt mit der Sichtbarkeit `public` vereinbaren. Damit sparen Sie sich die Eigenschaft, verstößen aber auch gegen die Datenkapselung.

Damit wir gleich abfragen können, ob ein Spiel gestartet wurde, vereinbaren Sie außerdem in der Klasse der Anwendung ein Feld `spielGestartet` vom Typ `bool`. Setzen Sie dieses Feld im Konstruktor auf `false`. Beim Starten eines Spiels in der Methode `Start()` setzen Sie es dann auf `true`.

Die Anweisungen zum Aktivieren beziehungsweise Deaktivieren des Menüeintrags Pause können Sie löschen. Das Aktivieren beziehungsweise Deaktivieren erfolgt ja gleich automatisch über das Command.

Erstellen Sie dann in der Klasse der Anwendung die beiden Methoden für die Steuerung beziehungsweise das Ausführen des Commands. Sie finden sie im folgenden Code 4.3:

```
private void Pause_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    //gibt es das Spielfeld und läuft ein Spiel?
    if (spielfeld != null && spielGestartet == true)
        e.CanExecute = true;
}

private void Pause_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    //pausieren?
    SpielPause();
}
```

Code 4.3: Die Methoden für das Command

In der Methode `Pause_CanExecute()` prüfen wir, ob es ein Spielfeld gibt und ob aktuell ein Spiel läuft. Wenn das der Fall ist, setzen wir die Eigenschaft `e.CanExecute` auf `true`.

Hinweis:

Die Abfrage, ob ein Spiel läuft oder nicht, ist erforderlich, damit die Pausenfunktion nicht direkt nach dem Start aufgerufen werden kann. Das Feld `spielUnterbrochen` können wir für die Abfrage nicht benutzen. Dann könnten wir das Spiel zwar einmal unterbrechen, aber nicht mehr starten.

In der Methode `Pause_Executed()` rufen wir unsere Pausenfunktion auf. Das ist nichts Neues.

Im XAML-Code legen Sie das Command jetzt bei der Eigenschaft **CommandBindings** fest. Wir benutzen in unserem Beispiel wieder das Fenster. Die entsprechenden Anweisungen sehen so aus:

```
<Window.CommandBindings>
    <CommandBinding Command="{x:Static local:MainWindow.Pause}"
        CanExecute="Pause_CanExecute"
        Executed="Pause_Executed" />
</Window.CommandBindings>
```

Code 4.4: Die XAML-Anweisungen für das Setzen der Eigenschaft CommandBinding

Neu ist hier vor allem der Ausdruck `{x:Static local:MainWindow.Pause}` bei der Zuweisung des Commands. Wir verwenden hier eine statische Eigenschaft `MainWindow.Pause` aus dem lokalen Namensraum. Dieser lokale Namensraum wird mit der Anweisung

`xmlns:local="clr-namespace:Snake"`

bei den Eigenschaften des Fensters im XAML-Code festgelegt.

**Bitte beachten Sie:**

Sie müssen den Namen der Klasse hinter die Angabe `local:` und vor den Namen der Eigenschaft setzen. Sonst wird die Eigenschaft nicht gefunden.

Passen Sie dann die Eigenschaften für den Eintrag **Pause** im Menü **Spiel** so an, dass hier nicht mehr die Methode `spielPause_Click()` ausgeführt wird, sondern das Command `Pause`. Die komplette Anweisung im XAML-Code müsste nach der Änderung so aussehen:

```
<MenuItem x:Name="spielPause" Header="_Pause" IsCheckable="True"
    Command="{x:Static local:MainWindow.Pause}" />
```

Speichern Sie alle Änderungen und führen Sie einen Test durch. Geändert hat sich aber eigentlich nichts im Spiel. Denn es fehlt ja noch der Aufruf der Funktion über die Tastatur. Dazu müssen wir die Taste **P** über die Eigenschaft **InputBindings** mit dem Command verbinden. Wir nehmen die Bindung ebenfalls auf Ebene des Fensters vor.

Die entsprechenden XAML-Anweisungen finden Sie im folgenden Code 4.5:

```
<Window.InputBindings>
    <!-- bitte in einer Zeile eingeben -->
    <KeyBinding Command="{x:Static local:MainWindow.Pause}"
    Key="P"/>
</Window.InputBindings>
```

Code 4.5: Die XAML-Anweisungen für das Setzen der Eigenschaft InputBindings

Bei KeyBinding setzen Sie über die Eigenschaft Command das Command und über die Eigenschaft Key die Taste. Über die Eigenschaft Modifiers können Sie auch Tastenkombinationen angeben. Die Werte Modifiers= "Control" Key="P" stehen zum Beispiel für die Tastenkombination **Strg** + **P**.

Eine vollständige Liste der möglichen Werte für die Eigenschaft Modifiers finden Sie in der Hilfe unter dem Stichwort **ModifierKeys Enumeration**.



Damit die Taste jetzt auch im Menü hinter dem Eintrag angezeigt wird, setzen Sie bitte noch die Eigenschaft **InputGesture** für den Eintrag auf **P**. Sie finden diese Eigenschaft im Eigenschaftenfenster in der Gruppe **Inhalt**.

4.3 Ein Startbildschirm

Als Krönung versehen wir unser Spiel jetzt noch mit einem Startbildschirm.

Ein Startbildschirm wird im Fachjargon auch **Splash Screen** genannt.



Dazu fügen Sie die Grafik, die Sie als Startbildschirm verwenden wollen, zum Projekt hinzu. Bei den Eigenschaften der Datei ändern Sie dann den Eintrag für die Eigenschaft **Buildvorgang** in **SplashScreen**.

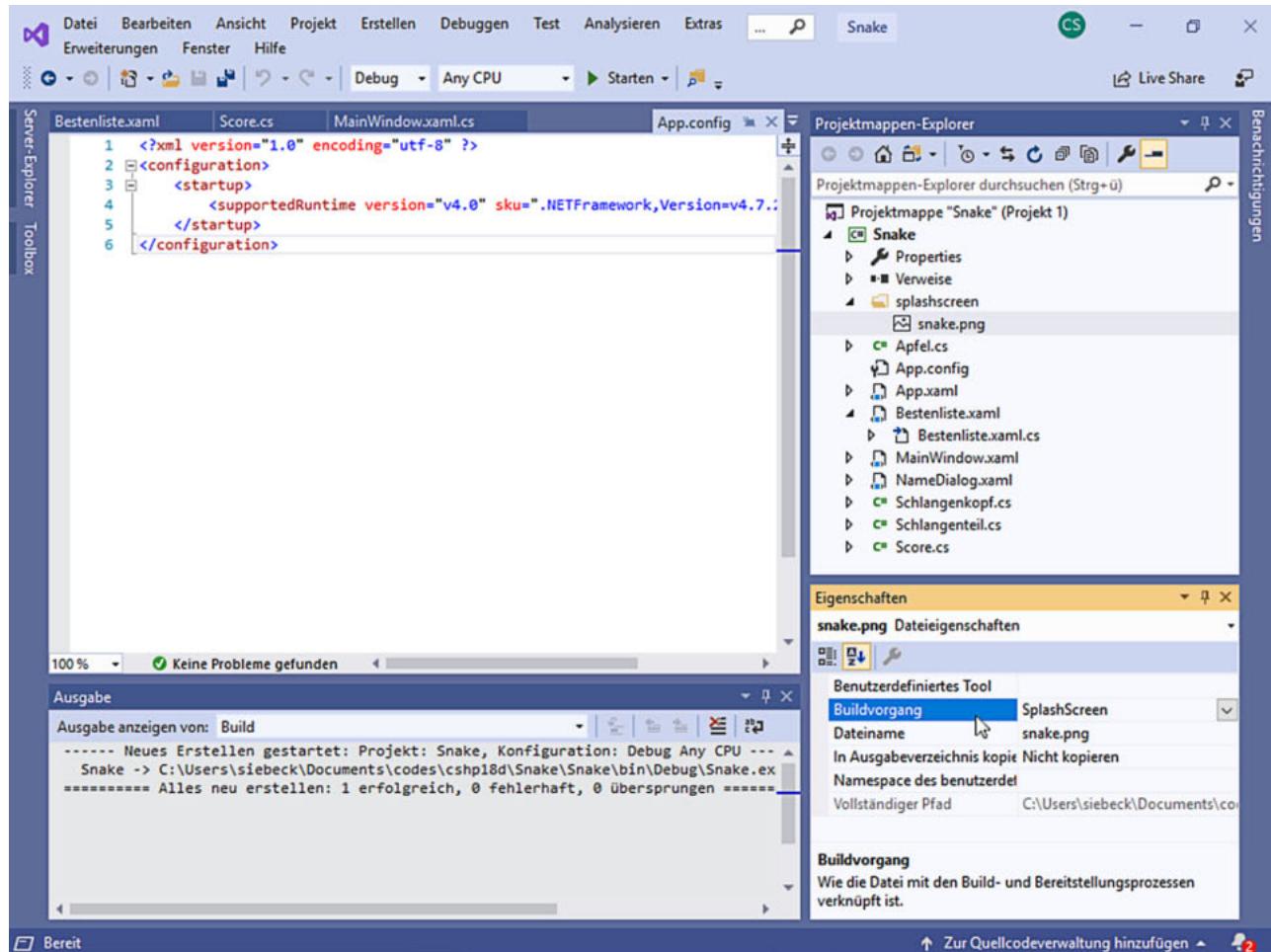


Abb. 4.1: Die Eigenschaft Buildvorgang für eine Datei (unten rechts am Mauszeiger)

Die Grafik wird dann direkt nach dem Start der Anwendung angezeigt und automatisch ausgeblendet, wenn die Anwendung komplett geladen ist. In unserem Fall passiert das allerdings so schnell, dass das Bild kaum zu sehen ist. Sie können das Starten aber künstlich verzögern, indem Sie zum Beispiel als erste Anweisung im Konstruktor die Methode `System.Threading.Thread.Sleep()` aufrufen. Die Wartezeit in Millisekunden geben Sie dabei über das Argument an.

Zusammenfassung

Eigene Commands erstellen Sie über die Klasse `RoutedCommands`.

Wenn Sie eine Eigenschaft für den Zugriff auf ein geschütztes Feld verwenden wollen, benutzen Sie für die Eigenschaft den Namen des Felds mit einem Großbuchstaben zu Beginn.

Über die Eigenschaft **Buildvorgang** können Sie festlegen, dass eine Grafik als Startbildschirm angezeigt werden soll.

Aufgaben zur Selbstüberprüfung

- 4.1 Beschreiben Sie kurz in eigenen Worten das Erstellen eines Commands. Es soll auch einer Taste zugeordnet werden.

- 4.2 Sie finden im XAML-Code eines Fensters bei der Eigenschaft **CommandBindings** folgenden Ausdruck

```
{x:Static local:MainWindow.Start}
```

Worauf wird mit dem Ausdruck zugegriffen?

- 4.3 Was geschieht, wenn Sie nur die Eigenschaft **InputGesture** für einen Menüeintrag setzen, es aber für die Taste kein Command gibt? Probieren Sie es aus.

- 4.4 Auf welchen Wert müssen Sie die Eigenschaft **Buildvorgang** für eine Grafik setzen, damit sie als Startbildschirm angezeigt wird?

- 4.5 Wie lange wird eine Grafik, die Sie als Startbildschirm benutzen, in der Standardeinstellung angezeigt?

Schlussbetrachtung

Herzlichen Glückwunsch! Sie haben die nächste – durchaus anspruchsvolle – Anwendung fertiggestellt. In dem Snake-Spiel haben wir auf der einen Seite neue Techniken wie das Verschieben der Schlange, das Erstellen einer generischen Liste und die Kollisionsprüfung eingesetzt, auf der anderen Seite aber auch auf Techniken zurückgegriffen, die Sie bereits in anderen Projekten eingesetzt haben.

Erweitern Sie auch dieses Spiel im Alleingang. Sie können zum Beispiel die Kollisionsabfrage verfeinern oder neue Früchte benutzen, die als Bonus dienen. Sie könnten für die Schlange auch mehrere „Leben“ vorsehen – also das Spiel nicht direkt bei der ersten Kollision beenden.

Einige Erweiterungen warten aber auch wie gewohnt bei den Einsendeaufgaben auf Sie.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 2

2.1 Damit ein Fenster zentriert auf dem Bildschirm angezeigt wird, müssen Sie die Eigenschaft auf den Wert `CenterScreen` setzen.

2.2 Das Steuerelement wird oben links positioniert und nimmt den gesamten freien Platz ein.

2.3 Die aktuelle Höhe und Breite ermitteln Sie über die Eigenschaften `ActualHeight` und `ActualWidth`.

2.4 Die Anweisung könnte so aussehen:

```
List<int> listeInt = new List<int>();
```

Der Bezeichner `listeInt` kann auch abweichen.

2.5 Die Methode zum Einfügen von Werten am Ende einer generischen Liste heißt `Add()`.

2.6 Die Anzahl der Elemente in einer generischen Liste erhalten Sie über die Eigenschaft `Count`.

2.7 Sie übergeben einen Wert an den Konstruktor der Klasse.

2.8 Um den Hashcode zu ermitteln, benutzen Sie die Methode `GetHashCode()`.

2.9 Die Methode `HitTest()` liefert ein Ergebnis vom Typ `HitTestResult`. Es enthält entweder Angaben zu dem Objekt, das gefunden wurde, oder den Wert `null`, wenn nichts gefunden wurde.

2.10 Der Ausdruck für den Zugriff auf die Eigenschaft `Name` könnte so aussehen:

```
((Shape)(treffer.VisualHit)).Name
```

Der Bezeichner `treffer` kann auch abweichen.

Kapitel 3

- 3.1 Um die Bildschirmauflösung abzufragen, verwenden Sie die Eigenschaften `SystemParameters.PrimaryScreenWidth` beziehungsweise `SystemParameters.PrimaryScreenHeight`.
- 3.2 Den Ordner können Sie sich mit der Eigenschaft `System.AppDomain.CurrentDomain.BaseDirectory` beschaffen.
- 3.3 Die Anweisungen könnten so aussehen:

```
Grid.SetRow(meinLabel, 4);  
Grid.SetColumn(meinLabel, 2);  
meinGrid.Children.Add(meinLabel);
```

- 3.4 Die Anweisung könnte so aussehen:

```
meinGrid.RowDefinitions.Add(new RowDefinition());
```

Kapitel 4

- 4.1 Für das Erstellen eines eigenen Commands gehen Sie in den folgenden Schritten vor:
 1. Sie legen für das Command ein statisches Feld vom Typ `RoutedCommand` an. Das Feld muss nach außen sichtbar sein.
 2. Sie legen über die Eigenschaften **CanExecute** und **Executed** die Methoden für die Steuerung beziehungsweise das Ausführen des Commands fest.
 3. Sie legen die beiden Methoden für das Command an. Die Methode für die Eigenschaft **CanExecute** muss dabei die Eigenschaft `e.CanExecute` auf `true` setzen, wenn das Command ausgeführt werden kann.
 4. Sie legen das Command bei der Eigenschaft **CommandBindings** eines Objekts fest.
 5. Sie ordnen das Command einem Steuerelement zu.
 6. Sie weisen dem Command die Taste über die Eigenschaft **InputBindings** eines Objekts zu.
- 4.2 Mit dem Ausdruck wird auf eine statische Eigenschaft `windowMain.Start` im lokalen Namensraum zugegriffen.
- 4.3 Es wird lediglich die Taste im Menü angezeigt. Sie hat aber keinerlei Wirkung.
- 4.4 Sie müssen die Eigenschaft **Buildvorgang** auf den Wert `SplashScreen` setzen.
- 4.5 Die Grafik wird in der Standardeinstellung so lange angezeigt, bis das Programm komplett geladen wurde.

B. Glossar

Canvas	Das Steuerelement Canvas ist ein Layout-Container, in dem untergeordnete Elemente über absolute Koordinaten positioniert werden können.
Casting	Siehe Typecasting.
Code-Behind-Dateien	Die Code-Behind-Dateien sind die Quelltextdateien für die Programmlogik einer WPF-Anwendung oder einer Universal Windows Platform App.
Generic	Ein Generic ist eine Methode, Klasse, Struktur und so weiter, die Platzhalter für die Typen benutzt, die von ihm verarbeitet werden. Die konkreten Typen für die Platzhalter werden erst zur Laufzeit des Programms eingesetzt.
Generika	Generika ist eine andere Bezeichnung für Generics. Der Begriff wird auch für Nachbildungen von Medikamenten benutzt.
Hashcode	Ein Hashcode ist ein numerischer Wert, über den Daten nahezu eindeutig identifiziert werden können. So haben unterschiedlicher Instanzen einer Klasse in der Regel auch unterschiedliche Hashcodes.
Modales Fenster	Ein modales Fenster steht immer im Vordergrund und kann auch nicht minimiert werden. Mit der eigentlichen Anwendung kann erst dann weitergearbeitet werden, wenn das modale Fenster wieder geschlossen wird. Typische Beispiele für modale Fenster sind unter anderem die Öffnen- und Speicherndialoge von Windows.
Splash Screen	Splash Screen ist ein anderer Ausdruck für einen Startbildschirm.
Standardkonstruktor	Ein Standardkonstruktor ist ein Konstruktor ohne Parameter.
Startbildschirm	Ein Startbildschirm wird direkt nach dem Aufruf eines Programms angezeigt. Er soll Wartezeiten überbrücken oder Informationen anzeigen.
Substitution	Bei der Substitution kann eine abgeleitete Klasse überall dort benutzt werden, wo auch die übergeordnete Klasse verwendet werden kann.
Timer	Ein Timer löst in einem frei definierbaren Zeitabstand immer wieder ein Ereignis aus, in dem Sie Anweisungen verarbeiten lassen können. Ein Timer ermöglicht damit die zeitgesteuerte Verarbeitung von Anweisungen.

Typecasting Typecasting bezeichnet das Umwandeln eines Wertes in einen anderen Datentyp.

Überschreiben Beim Überschreiben erstellen Sie mehrere Methoden mit demselben Namen und derselben Parameterliste in unterschiedlichen Klassen.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Auflage. Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 4. Auflage. Bonn: Rheinwerk.

Speziell mit dem Thema WPF beschäftigt sich das folgende Buch:

Huber, T.C. (2019). *Windows Presentation Foundation. Das umfassende Handbuch*. 5. Auflage. Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Das Snake-Spiel	4
Abb. 2.1	Das Fenster mit dem Spielfeld	6
Abb. 2.2	Das Spielfeld	9
Abb. 2.3	Die Schlange	18
Abb. 2.4	Die Schlange lässt sich lenken	19
Abb. 2.5	Ein Apfel im Spielfeld	21
Abb. 2.6	Mehrere Äpfel auf dem Spielfeld	22
Abb. 2.7	Eine Kollision zwischen zwei Objekten	24
Abb. 2.8	Die Rechtecke für die Kollision	26
Abb. 2.9	Die Schlange wird immer länger	28
Abb. 2.10	Snake in der „Rohversion“	30
Abb. 3.1	Das Formular zur Eingabe des Namens	41
Abb. 3.2	Das Fenster für die Bestenliste	45
Abb. 3.3	Die Bestenliste im Einsatz	47
Abb. 4.1	Die Eigenschaft Buildvorgang für eine Datei	54

E. Tabellenverzeichnis

Tab. 3.1	Die Eigenschaften für den Eingabedialog	41
Tab. 3.2	Die Eigenschaften für das Fenster der Bestenliste	44

F. Codeverzeichnis

Code 2.1	Die Zeilen für das Grid	5
Code 2.2	Das Gerüst für das Menü	5
Code 2.3	Die Felder für das Spiel	7
Code 2.4	Die Methode Start()	7
Code 2.5	Die Methoden ZeichneRechteck() und ZeichneSpielfeld()	8
Code 2.6	Die Klasse Schlangenteil	12
Code 2.7	Die Klasse Schlangenkopf	14
Code 2.8	Das Erzeugen einer Testschlange	15
Code 2.9	Die Methode für den Timer	16
Code 2.10	Das Erzeugen und Starten des Timers	17
Code 2.11	Die Methode Window_KeyDown()	18
Code 2.12	Die Klasse Apfel	20
Code 2.13	Das Erzeugen der Äpfel	21
Code 2.14	Die neue Methode zum Liefern der Position	23
Code 2.15	Die Abfrage auf einen Treffer	23
Code 2.16	Das Erzeugen des Rechtecks für die Kollision	25
Code 2.17	Die Reaktion auf eine Kollision	27
Code 2.18	Das Erzeugen eines neuen Apfels bei der Kollision	29
Code 2.19	Der Timer für die Spielzeit	30
Code 3.1	Der erweiterte Konstruktor für die Anwendung	35
Code 3.2	Die Methode SpielPause()	36
Code 3.3	Die Methode für das Klicken auf den Menüeintrag Pause	36
Code 3.4	Die Methode NeuesSpiel()	37
Code 3.5	Die Anweisungen für das Anklicken des Menüeintrags Neues Spiel.....	37
Code 3.6	Das Beenden des Spiels	38
Code 3.7	Die Methoden für das Ändern des Schwierigkeitsgrads	40
Code 3.8	Die Methoden für das Fenster NameDialog	42
Code 3.9	Die geänderte Methode NeuerEintrag()	43
Code 3.10	Der XAML-Code für das Grid der Bestenliste	44
Code 3.11	Der Konstruktor der Klasse Bestenliste	46
Code 3.12	Die geänderte Methode ListeAusgeben()	46
Code 4.1	Das Erhöhen der Geschwindigkeit	49
Code 4.2	Die Eigenschaft für den Zugriff auf das Feld	51
Code 4.3	Die Methoden für das Command	51

Code 4.4	Die XAML-Anweisungen für das Setzen der Eigenschaft CommandBinding	52
Code 4.5	Die XAML-Anweisungen für das Setzen der Eigenschaft InputBindings	53

G. Sachwortverzeichnis

A

Anwendung
 Ordner der, beschaffen 43

B

Bildschirmauflösung
 ermitteln 40

C

Command
 eigenes erstellen 50
 Taste für ein, setzen 52

E

Eigenschaft
 Topmost 5
 WindowsStartupLocation 5

G

Generic 10
Generika 10

H

Hashcode 22

K

Klasse
 List 10
Kollisionsabfrage 22
Konstruktor
 der Basisklasse aufrufen 14

M

Menüleiste 5

O

Objekt
 aus einer grafischen Oberfläche
 löschen 38

S

Schlange
 Bewegen der 16
Schwierigkeitsgrad 38
Spielfeld 6
 erstellen 5
Spielregeln 3
Spielzeit
 Anzeige der 30
Splash Screen 53
Startbildschirm 53
Starten und Anhalten 34
Steuerelement
 aktuelle Breite und Höhe ermitteln 8
Substitution 11

Z

Zufallsgenerator
 initialisieren 22

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH18D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Schicken Sie für die Lösungen bitte das vollständige Projekt mit allen Unterordnern und Dateien ein. Markieren Sie im Quelltext deutlich, zu welcher Aufgabe die jeweiligen Anweisungen gehören – zum Beispiel durch Kommentare.

Beschreiben Sie bei allen Aufgaben zusätzlich, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Änderungen Sie vornehmen beziehungsweise welche Steuerelemente Sie verwenden.

Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.

1. Programmieren Sie eine Beschleunigungsfunktion für die Schlange. Diese Funktion soll beim Drücken der Leertaste die Schlange auf das Doppelte ihrer aktuellen Geschwindigkeit beschleunigen. Ein erneutes Drücken der Leertaste soll die Geschwindigkeit wieder auf den alten Wert zurücksetzen.

Die Beschleunigungsfunktion soll aber nicht beliebig oft aufgerufen werden können. Für jede 100 Punkte, die der Spieler hat, soll er die Funktion eine Sekunde lang nutzen können – bei 200 Punkten also zwei Sekunden, bei 300 Punkten drei Sekunden und so weiter. Maximal soll der Spieler 10 Sekunden für die Beschleunigungsfunktion sammeln können.

Wenn der Spieler die Funktion nutzt, soll die genutzte Zeit von der zur Verfügung stehenden Zeit abgezogen werden. Für jede 100 Punkte wird dann wieder eine Sekunde aufaddiert, bis wieder maximal 10 Sekunden zur Verfügung stehen.

Zeigen Sie die zur Verfügung stehende Zeit mit einem geeigneten Steuerelement an – zum Beispiel einem ProgressBar. Hier können Sie über die Eigenschaft **Value** den aktuellen Wert setzen und über die Eigenschaft **Maximum** den maximalen Wert.

40 Pkt.

2. Erweitern Sie das Snake-Spiel um mindestens einen weiteren Level. Dieser Level muss sich vom bereits vorhandenen Level unterscheiden – zum Beispiel durch Hindernisse. Den Level können Sie zum Beispiel beim Erreichen einer bestimmten Punktzahl wechseln.

Blenden Sie vor dem Wechsel in einen anderen Level eine Meldung ein. Im neuen Level darf der Spieler wieder mit einer Schlange beginnen, die nur aus dem Kopf besteht. Alle anderen Werte wie die Geschwindigkeit der Schlange oder die noch verfügbare Dauer der Beschleunigung sollen sich aber nicht verändern.

Achten Sie bitte darauf, dass Sie beim Positionieren der Äpfel dafür sorgen müssen, dass sie nicht auf oder unter einem Hindernis liegen. Wie Sie das umsetzen, bleibt Ihnen überlassen. Dokumentieren Sie Ihre Lösung aber bitte ausreichend.

40 Pkt.

3. Erstellen Sie ein Tastaturkommando, um ein neues Snake-Spiel zu starten. Benutzen Sie dafür ein Command. Es soll über die Taste **N** aufgerufen werden.

20 Pkt.

insges. 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Die Planungsphase
UML-Grundlagen

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1119N01

CSHP19D

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Die Planungsphase UML-Grundlagen

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Die Planungsphase UML-Grundlagen

Inhaltsverzeichnis

Einleitung	1
1 Aufgaben und Ergebnisse der Planungsphase	3
Zusammenfassung	5
2 Das Lastenheft	6
2.1 Sinn und Zweck des Lastenhefts	6
2.2 Der Aufbau eines Lastenhefts	7
2.3 Informationssammlung für das Lastenheft	8
2.4 Das Lastenheft für die Autovermietung	12
Zusammenfassung	15
3 Aufwands- und Kostenschätzung	17
3.1 Faktoren für den Aufwand	17
3.2 Das „Teufelsquadrat“	18
3.3 Schätzmethoden	20
Zusammenfassung	22
4 Die Function-Point-Methode	24
4.1 Ermittlung der Function Points	24
4.2 Ermittlung der Entwicklungsdauer	27
Zusammenfassung	31
5 Projektmanagement	33
5.1 Das grundsätzliche Vorgehen	33
5.2 Meilensteine, Aktivitäten und Arbeitspakete	34
5.3 Die Termin- und Kapazitätplanung	35
5.4 Balkendiagramme und Netzpläne	37
5.5 Ein praktisches Beispiel	39
Zusammenfassung	42

6	Die Entscheidung	44
7	UML-Grundlagen	45
7.1	Die Geschichte der UML	45
7.2	Die Diagramme der UML	46
7.3	UML-Werkzeuge	49
	Zusammenfassung.....	50
	Schlussbetrachtung	52
	Anhang	
A.	Lösungen der Aufgaben zur Selbstüberprüfung	53
B.	Glossar	56
C.	Literaturverzeichnis.....	62
D.	Abbildungsverzeichnis	63
E.	Tabellenverzeichnis	64
F.	Sachwortverzeichnis	65
G.	Einsendeaufgabe	67

Einleitung

Ab diesem Studienheft beginnen wir konkret mit der Arbeit an unserem großen Beispielprojekt – einer Software für eine Autovermietung.

Diese Autovermietung hat sich grundsätzlich entschlossen, gemeinsam mit einem fiktiven Unternehmen SoftFix GmbH eine Software für die verschiedenen Prozesse im Unternehmen zu entwickeln. In einem ersten Schritt geht es jetzt darum, zu klären, ob diese Software nun tatsächlich programmiert wird oder nicht. Es muss eine **Machbarkeits- oder Durchführbarkeitsstudie** erstellt werden.

Am Ende dieses Studienhefts beschäftigen wir uns noch mit den Grundlagen der *Unified Modeling Language* (UML).

Im Einzelnen lernen Sie in diesem Studienheft:

- welche Dokumente die Machbarkeitsstudie umfasst,
- was der Sinn und Zweck eines Lastenhefts ist,
- wie Sie ein Lastenheft strukturieren können,
- wie Sie an die Informationen für das Lastenheft kommen,
- welche Faktoren den Aufwand bei der Software-Entwicklung bestimmen,
- wie diese Faktoren zusammenhängen,
- welche Methoden zur Aufwandsschätzung eingesetzt werden können,
- wie Sie die Function-Point-Methode einsetzen,
- wie Sie bei der Projektplanung grundsätzlich vorgehen,
- was sich hinter Meilensteinen, Aktivitäten und Arbeitspaketen verbirgt,
- wie Sie Termine planen,
- wie Sie Informationen aus einem Projektplan über Diagramme darstellen,
- wie die UML entstanden ist,
- welche Modelle in der UML grob unterschieden werden,
- welche Diagramme in der UML eingesetzt werden und
- welche Werkzeuge Sie bei der Arbeit mit der UML unterstützen können.

Hinweis:

Wir beschäftigen uns zunächst mit dem klassischen Vorgehen bei der Software-Entwicklung, bei dem mehrere Schritte aufeinander aufbauen und jeder Schritt Ergebnisse für den folgenden Schritt liefert. Später stellen wir Ihnen aber auch noch kurz agile Methoden wie Extreme Programming und Scrum vor. Sie wollen sehr viel schneller und mit weniger Aufwand zu greifbaren Ergebnissen kommen.

Christoph Siebeck

1 Aufgaben und Ergebnisse der Planungsphase

In diesem Kapitel stellen wir Ihnen zunächst die Aufgaben und die Ergebnisse der Planungsphase im Überblick vor.

Am Anfang eines Software-Lebenszyklus steht immer die **Idee** – „Ich will eine Software haben“ oder auch „Ich muss eine Software haben“. Diese Initialzündung kann dabei ganz unterschiedliche Ausgangspunkte haben:

- Die Software soll bisher manuell durchgeführte Abläufe erleichtern, schneller oder sicherer machen. Sie wird komplett neu erstellt.
- Jemand möchte mit der Software Geld verdienen – zum Beispiel durch die Entwicklung von Standardprogrammen für einen großen Markt.
- Eine bereits eingesetzte Software kann nicht weiterverwendet werden und muss überarbeitet werden. Das kann zum Beispiel dann der Fall sein, wenn das Unternehmen gewachsen ist und die Software den neuen Anforderungen nicht mehr gerecht wird oder wenn sich die Software nicht an Änderungen in der Umwelt anpassen lässt. Zu solchen Änderungen in der Umwelt gehört zum Beispiel die Einführung einer neuen Währung oder die Weiterentwicklung der eingesetzten Systemsoftware. Denken Sie hier nur an den Euro sowie die neuen Versionen von Windows oder Linux-Distributionen.

Egal woher die Initialzündung nun stammt: Sie muss in nahezu allen Fällen konkretisiert werden. Im ersten Schritt muss also zunächst einmal detailliert festgelegt werden, was die Software überhaupt machen soll beziehungsweise was genau geändert werden soll. Diese Anforderungen werden in einem **Lastenheft** festgehalten.

Ein Lastenheft findet sich nicht nur bei der Software-Entwicklung, sondern auch vielen anderen Projekten. Allgemein ausgedrückt beschreibt ein Lastenheft die Gesamtheit der Forderungen eines Auftraggebers an die Lieferungen und Leistungen eines Auftragnehmers.^{a)}



a) Quelle: DIN 69905.

Ausgehend vom Lastenheft können dann Aussagen über die wahrscheinliche Entwicklungszeit („Wie lange brauchen wir?“) und damit auch über die wahrscheinlichen Kosten („Wie teuer wird die Software?“) gemacht werden. Diese Ergebnisse werden in einem ersten **Projektplan** und einer groben **Kalkulation** festgehalten.

Zusammengefasst bilden das Lastenheft, der Projektplan und die Kalkulation die **Machbarkeitsstudie** – auch **Durchführbarkeitsstudie** genannt.

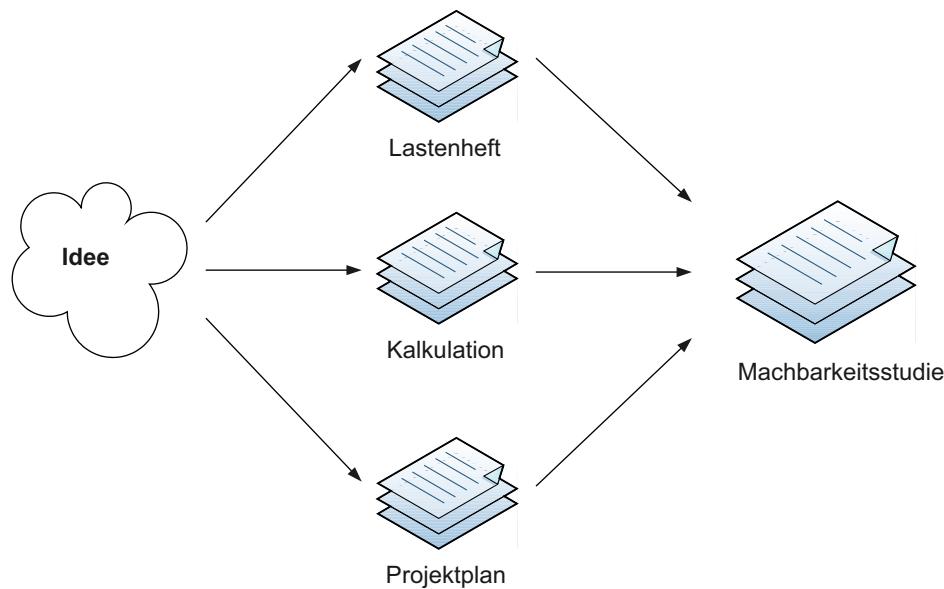


Abb. 1.1: Von der Idee zur Machbarkeitsstudie

Die Machbarkeitsstudie muss Antworten auf die folgenden Fragen geben können:

- Was soll die Software grundsätzlich leisten?
- Sind alle Anforderungen weitgehend vollständig erfasst?
- Widersprechen sich die Anforderungen nicht?
- Wie lange dauert die Entwicklung wahrscheinlich?
- Kann die Entwicklung mit dem vorhandenen Personal durchgeführt werden? Sind ausreichend Mitarbeiter vorhanden? Sind die Mitarbeiter genügend qualifiziert?
- Was kostet die Entwicklung?
- Lohnt sich die Erstellung überhaupt? Gibt es vielleicht kostengünstigere Alternativen?
- Welche Risiken können möglicherweise auftreten? Wie soll mit diesen Risiken umgegangen werden?

Denken Sie bitte daran, dass die Ergebnisse der Planungsphase die Grundlage für alle weiteren Schritte bilden. Das heißt: Wenn das fertige Produkt stimmen soll, muss auch die Planung stimmen. Das gilt nicht nur für die Software-Entwicklung. Wenn Sie zum Beispiel ein Bild aufhängen wollen und das Loch irgendwo bohren oder irgendeinen Haken nehmen, dürfen Sie sich nachher nicht wundern, dass das Bild zu hoch, zu tief, zu weit links hängt oder – im schlimmsten Fall – herunterfällt und dabei zerbricht.



Fehler, die Ihnen in der Planungsphase unterlaufen, lassen sich später häufig nur mit sehr großem Aufwand und hohen Kosten korrigieren.

In den folgenden Kapiteln werden wir uns im Detail ansehen, wie Sie die verschiedenen Dokumente der Planungsphase erstellen.

Zusammenfassung

Die Machbarkeitsstudie legt die Anforderungen an die Software fest und soll Antwort auf die Fragen nach der Entwicklungsdauer und den Kosten geben.

Sie bildet die Grundlage für alle weiteren Schritte.

Aufgabe zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Welche Dokumente gehören zur Machbarkeitsstudie? Beschreiben Sie auch kurz die Aufgaben der einzelnen Dokumente.

2 Das Lastenheft

In diesem Kapitel beschäftigen wir uns mit dem Lastenheft. Sie lernen, wie ein Lastenheft aufgebaut sein kann und woher die Informationen für das Lastenheft kommen können. Abschließend erstellen wir ein Lastenheft für die Software der Autovermietung.



Noch einmal zur Wiederholung:

Das Lastenheft legt fest, **was** die Software grundsätzlich machen soll. Das Wie – also die technische Umsetzung – spielt im Lastenheft keine Rolle.

2.1 Sinn und Zweck des Lastenhefts

Schauen wir zunächst einmal, **warum** überhaupt ein Lastenheft erstellt werden muss.

Ein häufiges Problem bei der Software-Entwicklung besteht darin, dass der Kunde und der Hersteller aneinander vorbeireden. Der Kunde formuliert zwar in der Regel seine Wünsche und Anforderungen – häufig aber so diffus, dass der Hersteller nicht weiß, was er denn nun genau entwickeln soll, oder aber die Aufgabenstellung völlig falsch versteht. Der Hersteller dagegen benutzt oft ein technisches Vokabular, das vom Kunden nicht oder falsch verstanden wird.



Beispiel 2.1:

Unsere Autovermietung hat die Anforderungen an das Programm so formuliert:

„Wir bieten unterschiedliche Autos zur Miete an – unter anderem Pkws und Transporter. Diese Autos können von unseren Kunden für einen bestimmten Zeitraum angemietet werden. Dazu muss ein Mietvertrag abgeschlossen werden. Wenn die Mietdauer beendet ist, kommt das Auto wieder zurück. Der Vertrag kann aber auch verlängert werden.“

Das Programm soll mindestens die Funktionen Kundenverwaltung und Vermietung haben.“

Hinweis: Bevor Sie weiterlesen ...

Versuchen Sie einmal selbst, aus dieser Produktbeschreibung den detaillierten Leistungsumfang der Software abzuleiten. Schreiben Sie sich dazu die wesentlichen Aussagen auf ein Blatt Papier und entwickeln Sie dann dazu die Funktionen, die benötigt werden.

Die Funktion „Kundenverwaltung“ ist vielleicht noch einigermaßen klar. Hier könnten zum Beispiel neue Kunden angelegt werden, bestehende Kundendaten verändert oder auch gelöscht werden. Offen bleiben dann aber die Fragen: Was für Daten werden für einen Kunden benötigt? Und wer ist überhaupt ein Kunde?

Und was verbirgt sich im Detail hinter der Funktion „Vermietung“? Was genau muss bei der Vermietung berücksichtigt werden? Sie sehen selbst: Fragen über Fragen.

Eine Möglichkeit wäre es nun natürlich, dass Sie als Entwickler diese Fragen selbst beantworten. Wenn Sie allerdings nicht zufällig einmal bei einer Autovermietung gearbeitet haben, werden Sie kaum über genügend Wissen für präzise Antworten verfügen. Im Endergebnis entwickeln Sie wahrscheinlich ein Programm, das möglicherweise völlig an den eigentlichen Anforderungen des Kunden vorbeigeht.

Einen Ausweg aus diesem Dilemma bietet nur eine möglichst genaue Beschreibung der Anforderungen **aus Sicht des Kunden** im Lastenheft. Das heißt, der Kunde muss so klar und eindeutig wie eben möglich definieren, was er von der Software erwartet.

Das Lastenheft beschreibt die Anforderungen aus Kundensicht, **nicht** aus Entwicklersicht. Das bedeutet: Der Kunde muss genau verstehen, was im Lastenheft beschrieben ist. Und: Kunde und Entwickler erarbeiten das Lastenheft in der Regel **gemeinsam**.



Schauen wir uns nun an, wie ein Lastenheft aufgebaut werden kann.

2.2 Der Aufbau eines Lastenhefts

Damit das Lastenheft möglichst einfach nachzuvollziehen und auch zu prüfen ist, sollten Sie es inhaltlich strukturieren. Wie Sie die Struktur aufbauen, ist dabei beliebig. Feste Vorgaben gibt es nicht.

Balzert empfiehlt aber zum Beispiel folgenden grundsätzlichen Aufbau:¹

1. Visionen und Ziele
2. Rahmenbedingungen
3. Kontext und Überblick
4. Funktionale Anforderungen
5. Qualitätsanforderungen

Schauen wir uns die einzelnen Punkte genauer an:

Bei den **Visionen und Zielen** werden die Visionen und Ziele dargestellt, die mit dem Einsatz des Produkts erreicht werden sollen. Die Angabe kann dabei durchaus recht allgemein erfolgen – zum Beispiel „Die Autovermietung soll mit der Software ihre Kunden, Autos und die Vermietung verwalten können.“

Bei den **Rahmenbedingungen** werden die **Anwendungsbereiche** und die **Zielgruppen** beschrieben – also, wer arbeitet wo mit dem Produkt.

Beim **Kontext und Überblick** beschreiben Sie das Umfeld des Produkts – zum Beispiel, welche Schnittstellen zu anderen Systemen verwendet werden oder wo und wie lange das System genau verfügbar sein soll.

Bei den **funktionalen Anforderungen** werden die wichtigsten Funktionen aus Anwendersicht beschrieben. Dabei spielt es zunächst einmal keine Rolle, ob diese Funktionen überhaupt durch die Software umgesetzt werden sollen. Die **Qualitätsanforderungen**

1. Balzert, H. (2009), S. 94.

beschreiben besondere Qualitätsmerkmale des Produkts – zum Beispiel, dass es besonders anwenderfreundlich oder leicht zu ändern sein muss. Die Beschreibung kann entweder sprachlich oder als Tabelle zum Ankreuzen erfolgen.

Tab. 2.1: Tabelle zum Festlegen der Qualitätsanforderungen

Qualitätsanforderung	sehr gut	gut	normal	unwichtig
Zuverlässigkeit				
Robustheit				
Benutzbarkeit				
Wiederverwendbarkeit				
...				

Um in späteren Phasen eindeutig auf eine Beschreibung zurückgreifen zu können, sollten die einzelnen Beschreibungen durchnummiert werden – zum Beispiel als LF10, LF20 oder LV 10² und so weiter.

Tipp:

Nummerieren Sie die Funktionen nicht in 1er-Schritten – zum Beispiel LF1, LF2 und so weiter. Stellen Sie bei dieser Art der Nummerierung später fest, dass Sie einen wichtigen Teil hinter der Funktion LF1 vergessen haben, müssen Sie nach dem Einfügen auch alle anderen nachfolgenden Beschreibungen neu nummerieren. Verwenden Sie dagegen 10er-Schritte, können Sie die neue Beschreibung zum Beispiel ohne Schwierigkeiten mit der Nummer LF15 einschieben.

Wie ein Lastenheft konkret aussehen kann, zeigen wir Ihnen gleich beim Lastenheft für die Autovermietung. Jetzt wollen wir uns zunächst einmal ansehen, wie Sie die nötigen Informationen für das Lastenheft beschaffen können.

2.3 Informationssammlung für das Lastenheft

Wenn wir noch einmal die möglichen Initialzündungen für das Erstellen von Software betrachten, lassen sich vier verschiedene Ausgangssituationen für das Erstellen des Lastenhefts unterscheiden:

1. Der Kunde liefert Ihnen bereits ein Lastenheft.
2. Sie wollen eine Standard-Software erstellen.
3. Sie wollen eine bestehende Software überarbeiten.
4. Sie wollen eine komplett neue Software für einen bestimmten Kunden entwickeln.

Der erste Fall – der Kunde liefert ein Lastenheft – findet sich zum Beispiel häufiger bei der Auftragsvergabe über Ausschreibungen. Der Kunde legt detailliert seine Anforderungen fest. Sie müssen dann „nur noch“ ein entsprechendes Angebot erstellen.

2. Das LF steht für Lastenheft Funktionale Anforderung. Das LV steht für Lastenheft Visionen und Ziele. Die Darstellung orientiert sich ebenfalls an Balzert.

Im zweiten Fall – bei der Entwicklung von Standard-Software – können Sie für das Lastenheft zum Beispiel **Marktanalysen** durchführen. Befragen Sie mögliche Kunden, welche Funktionen die Software haben sollte, oder sehen Sie sich entsprechende Produkte von Mitbewerbern an. Aus diesen Daten können Sie dann das Lastenheft entwickeln – allerdings weitestgehend im Alleingang. Eine konkrete Abstimmung mit dem Kunden ist allein schon deshalb schwierig, weil gar nicht klar ist, wer denn nun genau Ihr Kunde ist.

Im dritten Fall – bei der Überarbeitung einer bestehenden Software – sollten Sie zunächst einmal eine **Istanalyse** durchführen – also möglichst detailliert festhalten,

- welche Funktionen die bisherige Software unterstützt,
- welche dieser Funktionen überarbeitet werden müssen,
- welche dieser Funktionen übernommen werden können und
- welche Funktionen fehlen.

Mit den Daten der Istanalyse können Sie dann gezielt mit dem Kunden festlegen, was genau gemacht werden soll.

Der vierte Fall – das Entwickeln einer komplett neuen Software – ist in der Regel mit dem meisten Aufwand verbunden. Hier müssen Sie versuchen, durch möglichst viele Informationen die oft vagen Beschreibungen des Kunden zu präzisieren.

Eine Möglichkeit dazu bieten **Fragebögen**, die Sie aus der Kundenbeschreibung entwickeln können. Solch ein Fragebogen könnte so aussehen:

Welche Daten müssen genau für einen Kunden vom System verarbeitet werden können?		
Name und Vorname	ja	nein
Anschrift	ja	nein
eindeutige Kennzeichnung – zum Beispiel eine Kundennummer	ja	nein
...		
Erweitern Sie die Liste bitte um fehlende Daten.		
<hr/> <hr/> <hr/>		

Abb. 2.1: Musterfragebogen

Achten Sie dabei darauf, dass die Fragen möglichst präzise sind und nach Möglichkeit mit Ja oder Nein beantwortet werden können. Fragen, die eine freie Antwort ermöglichen, liefern oft keine eindeutigen Informationen.



Fragen, auf die vor allem mit Ja oder Nein geantwortet werden kann, werden auch **geschlossene Fragen** genannt. Fragen, die auf eine ausführliche Antwort zielen, heißen **offene Fragen**.

Eine andere Möglichkeit ist das Erstellen von Stellenbeschreibungen und Stellenprofilen. Bitten Sie die Mitarbeiter des Unternehmens, möglichst detailliert zu beschreiben, welche Tätigkeiten sie an ihrem Arbeitsplatz ausführen und welche Randbedingungen dabei beachtet werden müssen. Um den Mitarbeitern das Erstellen der Beschreibungen möglichst einfach zu machen, können Sie ein Formblatt entwickeln, das zum Beispiel folgende Fragen enthält:

- Welche Tätigkeit führen Sie aus? (Bitte vergeben Sie nach Möglichkeit einen kurzen beschreibenden Namen)
- Welche Daten benötigen Sie für die Ausführung?
- Woher stammen die Daten?
- Welches Ergebnis erzeugen Sie?
- Wohin wird das Ergebnis weitergeleitet?

Sehr hilfreich ist auch das Abfragen periodischer Tätigkeiten, wie:

- Schreiben Sie bitte Ihre täglichen Tätigkeiten auf.
- Schreiben Sie bitte Ihre wöchentlichen Tätigkeiten auf.

und so weiter.

Tipp:

In vielen Unternehmen gibt es bereits fertige Stellenbeschreibungen. Fragen Sie gegebenenfalls in der Personalabteilung nach.

Oft fällt es allerdings Mitarbeitern schwer, besonders Prozesse eindeutig zu beschreiben. Lassen Sie dann einfache Skizzen erstellen, die die einzelnen Schritte in der richtigen Reihenfolge und die Beteiligten darstellen. So entsteht eine Art Drehbuch – im Fachjargon auch **Storyboard**³ genannt.

3. *Storyboard* ist die englische Übersetzung für „Drehbuch“.

Für den Prozess „Vermietung“ in der Autovermietung könnte diese Skizze zum Beispiel so aussehen:

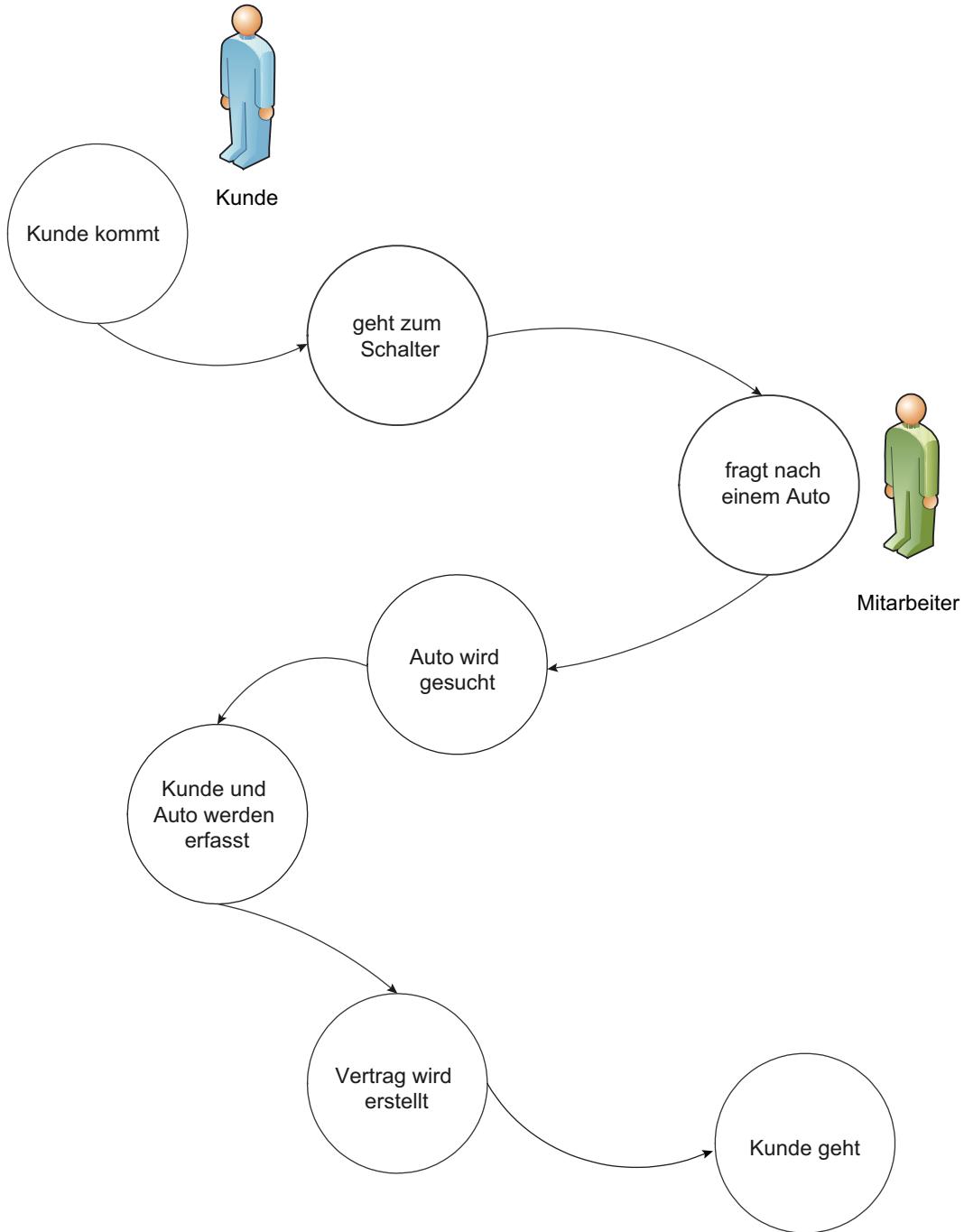


Abb. 2.2: Das Storyboard für den Prozess „Vermietung“

Tipp:

Fragen Sie auch hier nach, ob es im Unternehmen möglicherweise bereits Beschreibungen von Prozessen gibt.

Weitere Möglichkeiten, Informationen zu beschaffen, sind:

- Interviews mit Mitarbeitern,
- Brainstorming-Workshops,
- teilnehmende Beobachtung oder
- die direkte Mitarbeit als eine Art Praktikant.



Brainstorming^{a)} ist eine Technik, bei der spontane Einfälle zu einem Thema gesammelt werden. Die Strukturierung erfolgt in der Regel erst nach dem Sammeln der Einfälle.

Bei der **teilnehmenden Beobachtung** nimmt ein Dritter als Zuschauer an einem Arbeitsschritt teil. Er wirkt nicht aktiv mit, sondern versucht, den Arbeitsschritt möglichst exakt festzuhalten – zum Beispiel durch Notizen.

- a) Wörtlich übersetzt bedeutet *brainstorm* so viel wie „Geistesblitz“.

Grundsätzlich sollten Sie versuchen, so viele Informationen wie möglich zu beschaffen. Fragen Sie daher nicht nur die Mitarbeiter, sondern zum Beispiel auch mögliche Kunden des Unternehmens oder andere Beteiligte nach Wünschen und Verbesserungsvorschlägen.



Für die Beteiligten wird auch der Begriff **Stakeholder^{a)}** verwendet. Er umfasst sämtliche Personen, die von der Entwicklung und dem Einsatz des Systems betroffen sind.

- a) *Stake* bedeutet übersetzt so viel wie „Anteil, Beteiligung“. *Holder* bedeutet übersetzt so viel wie „Inhaber“ oder „Halter“.

2.4 Das Lastenheft für die Autovermietung

Nehmen wir jetzt einmal an, die Informationssammlung für unsere Autovermietung ist abgeschlossen und hat folgende Ergebnisse gebracht:

Die Autovermietung vermietet Fahrzeuge. Dazu gehören normale Pkws, Cabrios und Transporter. Jeder Fahrzeugtyp hat eine eindeutige Kennzeichnung.

Für die Vermietung werden der Name, die Anschrift und eine eindeutige laufende Nummer des Kunden festgehalten.

Der Kunde meldet sich mit seinem Mietwunsch bei einem Mitarbeiter. Dabei muss er den Fahrzeugtyp sowie das Datum und die Dauer der Anmietung mitteilen. Diese Daten werden gemeinsam mit den Daten des Kunden im Mietvertrag erfasst. Zusätzlich werden im Vertrag die Kosten für die Anmietung hinterlegt. Sie werden über einen Tagesmietpreis berechnet, der vom Fahrzeug abhängt.

Das angemietete Fahrzeug wird von Aushilfen von einem zentralen Sammelplatz zur Mietstation gefahren und dem Kunden übergeben.

Bei der Rückgabe werden die Daten des Kunden und des zurückgegebenen Fahrzeugs erfasst. Damit endet der Mietvertrag. Die zurückgegebenen Fahrzeuge werden durch die Aushilfen wieder an den zentralen Sammelplatz gefahren.

Fahrzeuge, die vermietet sind, kann ein anderer Kunde für sich reservieren.

An einer Stelle der Mietstation sollen Kunden Listen abrufen können, welche Fahrzeuge grundsätzlich im Bestand der Autovermietung sind und welche Fahrzeuge zurzeit vermietet werden können.

Damit sind die wichtigsten Prozesse und Leistungen beschrieben. Das Lastenheft könnte jetzt so aussehen:

Lastenheft „Autovermietung 2030“⁴

1. Visionen und Ziele

Die Software Autovermietung 2030 soll die Autovermietung bei der Verwaltung der Kunden und Fahrzeuge sowie bei der Vermietung und der Rückgabe der Fahrzeuge unterstützen. Außerdem soll sie den Kunden einen Überblick über die vorhandenen und ausleihbaren Fahrzeuge verschaffen können.

2. Rahmenbedingungen

Das Produkt wird in den Räumen der Autovermietung eingesetzt. Es wird von Mitarbeitern und Aushilfen in der Vermietung, der Rückgabe, der Verwaltung und von Kunden bedient.

3. Kontext und Überblick

Das System soll jeden Werktag mindestens 12 Stunden zur Verfügung stehen.

4. Funktionale Anforderungen

Die wesentlichen funktionalen Anforderungen werden im Folgenden immer mit einer eindeutigen Nummer, dem Namen und einer Kurzbeschreibung der Funktion dargestellt:

LF10 Pflege der Kundendaten

Beschreibung: Daten zum Kunden werden neu erfasst, geändert oder gelöscht.

LF20 Pflege der Fahrzeugdaten

Beschreibung: Daten zu den Fahrzeugen werden erfasst, geändert oder gelöscht.

LF30 Vermietung

Beschreibung: Der Kunde sucht ein Fahrzeug aus. Die Daten zum Kunden und zum Fahrzeug werden erfasst und zusammen mit der Mietdauer und den Kosten im Mietvertrag festgehalten. Das Einlesen der Daten im System soll automatisiert erfolgen. Dazu werden Strichcodes und entsprechende Lesegeräte verwendet.

4. Autovermietung 2030 soll der Name unserer Beispielsoftware sein.

LF40 Überführung eines gemieteten Fahrzeugs

Beschreibung: Das vermietete Fahrzeug wird von einer Aushilfe zur Mietstation gefahren und an den Kunden übergeben.

LF50 Rückgabe

Beschreibung: Der Kunde gibt ein Fahrzeug zurück. Die Daten zum Kunden und zum zurückgegebenen Fahrzeug werden erfasst. Auch hier soll das Einlesen der Daten über Strichcodes und entsprechende Lesegeräte automatisiert werden.

LF60 Rückführung der zurückgegebenen Fahrzeuge

Beschreibung: Die zurückgegebenen Fahrzeuge aus LF40 werden wieder an den Sammelplatz gefahren.

LF70 Reservierung

Beschreibung: Der Kunde reserviert ein aktuell vermietetes Fahrzeug für sich selbst.

LF80 Listen

Beschreibung: Der Kunde kann Listen zu den verfügbaren Fahrzeugen erzeugen. Dabei können alle Fahrzeuge oder nur die aktuell ausleihbaren Fahrzeuge angezeigt werden.

5. Qualitätsanforderungen

Qualitätsanforderung	sehr gut	gut	normal	unwichtig
Zuverlässigkeit			X	
Robustheit			X	
Benutzbarkeit	X			
Wiederverwendbarkeit			X	
Änderbarkeit			X	
Portierbarkeit				X

Die Benutzbarkeit muss sehr gut sein, da die Aushilfen ständig wechseln. Umfangreiche Schulungen können daher nicht erfolgen.

So viel zum Lastenheft für die Software Autovermietung 2030.

Abschließend wollen wir Ihnen noch einige allgemeine Tipps zum Erstellen von Lastenheften geben:

Achten Sie darauf, dass das Lastenheft für den Kunden verständlich ist.

Denken Sie daran: Der Kunde ist in der Regel kein Computerexperte. Vermeiden Sie daher – wann immer möglich – Fachbegriffe. Wenn es gar nicht anders geht, erklären Sie Fachbegriffe in einem gesonderten Glossar.

In unserem Lastenheft könnte beispielsweise der Begriff Strichcode so erklärt werden:

„**Strichcode** – Über einen Strichcode können Informationen sehr leicht in maschinenlesbarer Form dargestellt werden. Dazu werden unterschiedliche breite Striche und Lücken abgedruckt, die für die Informationen stehen. Ein typisches Beispiel für einen Strichcode finden Sie auf vielen Lebensmittelverpackungen.“

Achten Sie auf eindeutige Darstellungen.

Begriffe, die möglicherweise missverständlich sein könnten, sollten Sie ebenfalls im Glossar beschreiben.

Halten Sie das Lastenheft kurz.

In der Regel genügen einige Seiten. Andernfalls besteht die Gefahr, dass das Lastenheft nur noch überflogen wird. Kaum jemand macht sich die Mühe, zu diesem sehr frühen Zeitpunkt der Entwicklung Hunderte von Seiten zu studieren.

Überprüfen Sie das Lastenheft sorgfältig auf Tipp- und Rechtschreibfehler.

Benutzen Sie zum Beispiel die Rechtschreibprüfung Ihrer Textverarbeitung und kontrollieren Sie den Text auch noch einmal selbst sorgfältig. Ein Lastenheft, in dem es von Tippfehlern wimmelt, kann im Extremfall dazu führen, dass der Kunde an Ihren Qualitäten zweifelt – egal, ob das Lastenheft nun fachlich in Ordnung ist.

Verwenden Sie ein einheitliches, einfaches und klares Layout.

Benutzen Sie am besten nur eine einzige Schriftart und gehen Sie sparsam mit Textformatierungen wie fett und kursiv um. Unterstreichungen sollten Sie gar nicht verwenden. Sie machen einen Text schwer lesbar.

Und – ganz wichtig aus kaufmännischer Sicht:

Achten Sie darauf, dass das Lastenheft vor einem Vertrag erstellt wird. Machen Sie das Lastenheft – wann immer möglich – zum Bestandteil des Vertrags beziehungsweise bauen Sie den Vertrag auf dem Lastenheft auf. Sollte das nicht möglich sein, legen Sie im Vertrag fest, dass die Funktionalität durch ein noch zu erstellendes Lastenheft definiert wird.

So viel zum Lastenheft. Im nächsten Kapitel werden wir uns mit der Aufwands- und Kostenschätzung beschäftigen.

Zusammenfassung

Das Lastenheft beschreibt die Anforderungen an das System aus Sicht des Anwenders. Es muss klar, eindeutig und nachvollziehbar sein.

Das Lastenheft sollte eine klare inhaltliche Struktur haben.

Aufgaben zur Selbstüberprüfung

- 2.1 In welche wesentlichen Punkte können Sie ein Lastenheft gliedern?

- 2.2 Sie sollen eine Software komplett neu erstellen. Nennen Sie einige Techniken, wie Sie die Informationen für das Lastenheft sammeln können.

- 2.3 Warum sollten Sie bei der Nummerierung im Lastenheft keine 1er-Schritte verwenden?

3 Aufwands- und Kostenschätzung

In diesem Kapitel lernen Sie, wie Sie eine Aufwands- und Kostenschätzung für die Entwicklung von Software durchführen. Es geht also darum, die Fragen „Wie lange brauchen wir für die Entwicklung?“ und „Was kostet die Entwicklung?“ zu beantworten.

Zuerst erfahren Sie, wovon der Aufwand überhaupt abhängt. Danach stellen wir Ihnen einige Schätzmethoden vor.

3.1 Faktoren für den Aufwand

Schauen wir uns zuerst einmal an, was sich hinter dem Begriff Aufwand verbirgt:

In der betriebswirtschaftlichen Kosten- und Leistungsrechnung bezeichnet der Aufwand den gesamten Werteverzehr eines Unternehmens in einer Rechnungsperiode – also zum Beispiel Wareneinkäufe, Kosten für Dienstleistungen, Abgaben an den Staat und so weiter. Wir betrachten hier aber vor allem den zeitlichen Aufwand – also die Dauer der Entwicklung.

Denn ein Großteil der Kosten bei der Software-Entwicklung entsteht durch die Personalkosten. Und das heißt: Je länger die Entwicklung dauert, desto höher sind auch die Kosten.

Hinweis:

Aus Sicht der Kostenrechnung ist diese Einschränkung falsch, da ja zum Beispiel auch alleine für den Betrieb des Unternehmens Kosten entstehen – zum Beispiel für Miete, Heizung und so weiter. Diese Kosten fallen bei der Beschränkung auf die Personalkosten einfach unter den Tisch. Wir wollen hier aber keine exakte Kostenrechnung durchführen, sondern nur eine grobe Kalkulation. Und das ist – wie Sie gleich sehen werden – schon schwierig genug.

Generell hängt der Aufwand für die Erstellung einer Software von zwei wesentlichen Faktoren ab:

- der Quantität und
- der Qualität.

Die **Quantität** beschreibt dabei den Umfang der Software – also welche und wie viele Funktionen umgesetzt werden sollen. Grundsätzlich lässt sich sagen: Je vielfältiger und komplexer die Funktionen sind, desto umfangreicher wird die Software – desto höher ist die Quantität.

Ähnliches gilt für die **Qualität**. Je höher die Qualitätsanforderungen, desto höher auch der Aufwand. Oder andersherum ausgedrückt: Wenn Aspekte wie Benutzerfreundlichkeit, Wiederverwendbarkeit oder Zuverlässigkeit keine Rolle spielen, lässt sich ein Programm „schlampiger“ und damit auch schneller erstellen.

Das heißt also: Die Dauer der Entwicklung, die Kosten, die Quantität und die Qualität hängen direkt zusammen und bestimmen sich gegenseitig.

3.2 Das „Teufelsquadrat“

Grafisch lässt sich diese Abhängigkeit in einem Quadrat darstellen. Die vier Ecken stehen dabei für die Qualität, die Quantität, die Dauer sowie die Kosten und das gesamte Quadrat für den Aufwand.

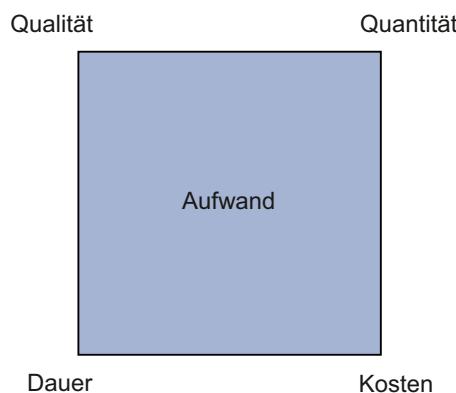


Abb. 3.1: Abhängigkeiten bei der Software-Entwicklung

Wenn Sie jetzt zum Beispiel die Quantität erhöhen – also die entsprechende Ecke nach außen ziehen –, wandern auch die anderen drei Ecken weiter nach außen. Das Quadrat wird größer – der Aufwand nimmt zu.

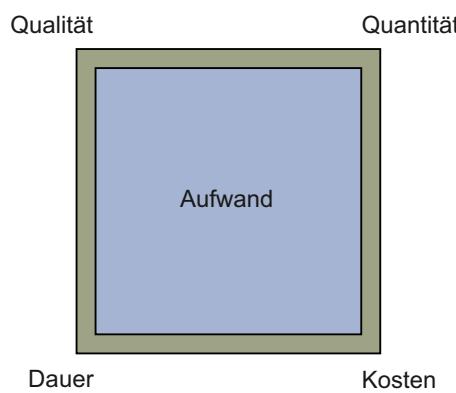


Abb. 3.2: Veränderung nach dem Erhöhen der Quantität (das grüne Quadrat außen)

Das Gleiche geschieht, wenn Sie einen Faktor verringern – also eine Ecke nach innen ziehen. Auch hier würden die anderen Ecken automatisch mitwandern. Das Quadrat und damit der Aufwand werden kleiner.

Dieses sehr einfache Modell berücksichtigt aber nicht, dass für die Entwicklung normalerweise nur begrenzte Kapazitäten zur Verfügung stehen. So lässt sich zum Beispiel die Produktivität der Programmierer nur mit viel Aufwand erhöhen. Sie müssten entweder neue Mitarbeiter einstellen, die dann aber erst einmal eingearbeitet werden müssen, oder bessere Entwicklungswerkzeuge einsetzen.

Ergänzt man jetzt in dem Modell von oben die Produktivität durch ein weiteres Quadrat, entsteht das „Teufelsquadrat“ von Sneed.

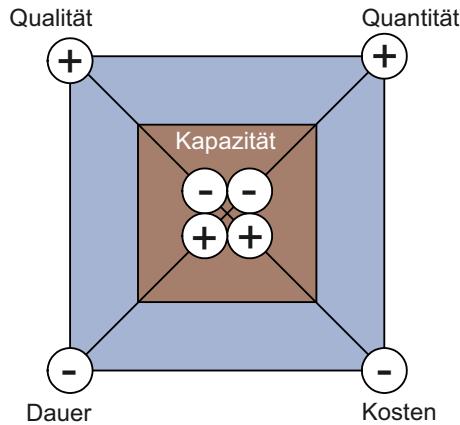


Abb. 3.3: Das „Teufelsquadrat“ von Sneed

Die vier Ecken der beiden Quadrate stehen dabei wieder für die vier wesentlichen Faktoren Quantität, Qualität, Dauer und Kosten. Die Pluszeichen an den Ecken stehen für eine Erhöhung eines Faktors, die Minuszeichen entsprechend für eine Verringerung der Wertigkeit.

Änderungen werden jetzt nicht mehr an den Ecken des äußeren Quadrats vorgenommen, sondern an den Ecken des inneren Quadrats. Die Fläche des inneren Quadrats bleibt dabei aber unverändert. Dieses Quadrat steht ja für die Produktivität, die sich insgesamt nicht verändern kann.

Wenn Sie zum Beispiel die Ecke für die Kosten nach außen ziehen, muss auch eine der anderen drei Ecken automatisch nach innen wandern, damit die Fläche des Quadrats gleich bleibt. Das heißt: Eine Reduzierung der Kosten führt automatisch dazu, dass sich entweder die Quantität oder die Qualität verringert beziehungsweise die Dauer erhöht.

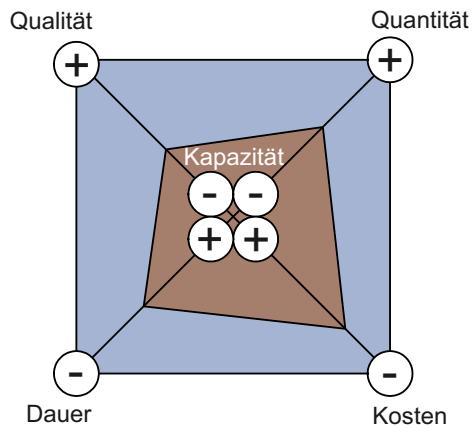


Abb. 3.4: Auswirkung der verringerten Kosten (hier hat sich die Qualität verringert)

Das Gleiche geschieht, wenn Sie zum Beispiel die Qualität erhöhen – also die entsprechende Ecke des inneren Quadrats nach außen ziehen. Diese Änderung führt automatisch dazu, dass sich entweder die Quantität verringert oder die Kosten beziehungsweise die Dauer erhöhen.

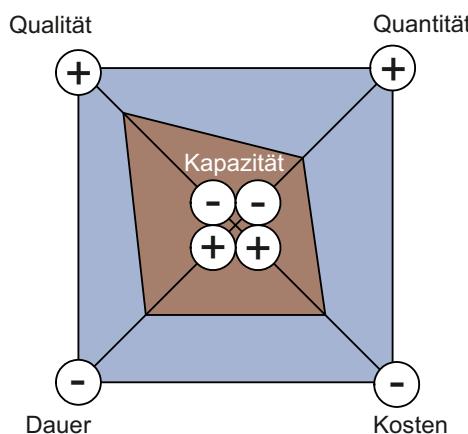


Abb. 3.5: Auswirkungen der erhöhten Qualität (hier hat sich die Quantität verringert)

Noch deutlicher werden die Auswirkungen, wenn Sie Veränderungen an einem oder mehreren Faktoren ausschließen wollen – also die entsprechenden Ecken des inneren Quadrats quasi fixieren. Auch dazu ein Beispiel:



Beispiel 3.1:

Die Entwicklungsdauer soll verkürzt werden, ohne dass sich die Qualität und die Quantität verändern dürfen. Damit wandert die Ecke für die Kosten automatisch nach innen, wenn Sie die Ecke für die Dauer nach außen ziehen – die Kosten steigen.

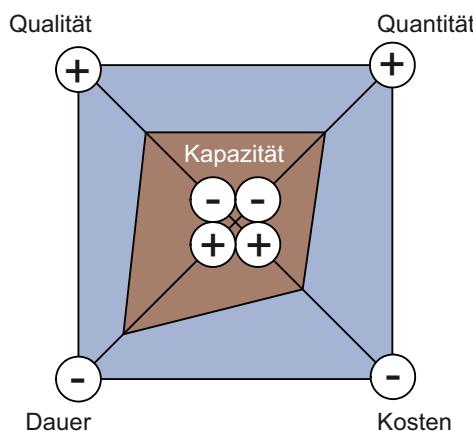


Abb. 3.6: Steigende Kosten bei verkürzter Entwicklungsdauer

So viel zum „Teufelsquadrat“.

3.3 Schätzmethoden

Schauen wir uns nun einige Schätzmethoden an. Beginnen wir mit der **Analogiemethode**.

Hier werden die Anforderungen aus dem aktuellen Projekt mit bereits durchgeführten Projekten verglichen. Daraus kann dann mehr oder weniger nach Gefühl der Aufwand für das aktuelle Projekt geschätzt werden. Diese Methode setzt allerdings einen sehr er-

fahrenden Schätzer voraus. Außerdem müssen bereits ähnliche Projekte unter ähnlichen Rahmenbedingungen durchgeführt worden sein. Daher sind die Ergebnisse in der Regel nur sehr grob und liegen nicht selten deutlich unter dem tatsächlichen Aufwand.

Nach einem ähnlichen Prinzip arbeiten **Prozentsatzmethoden**. Hier werden die prozentualen Anteile einer Entwicklungsphase aus ähnlichen Projekten herangezogen.

Beispiel 3.2:

Aus einem bereits abgeschlossenen Projekt wissen Sie, dass die Analyse der Anforderungen 20 Prozent der gesamten Projektzeit beansprucht hat. Wenn Sie jetzt ein vergleichbares Projekt durchführen, können Sie davon ausgehen, dass die Analysephase ebenfalls 20 Prozent der gesamten Zeit beanspruchen wird, und dann die gesamte Dauer entsprechend hochrechnen.



Wenn Sie also im aktuellen Projekt 30 Tage für die Analyse benötigt haben, werden Sie für das gesamte Projekt nach der Prozentsatzmethode 150 Tage benötigen.

Die Rechnung in Detail:

20 Prozent entsprechen einem Fünftel. $30 \text{ Tage} \cdot 5$ ergibt dann 150 Tage.



Die Prozentsatzmethode funktioniert allerdings nur, wenn mindestens die erste Phase des aktuellen Projekts bereits abgeschlossen ist. Außerdem steht und fällt diese Methode ebenfalls mit der Erfahrung des Schätzers, der ja zunächst einmal feststellen muss, ob überhaupt ein vergleichbares Projekt durchgeführt wurde.

Weitere Methoden für die Schätzung sind **LOC-Varianten**. Sie basieren auf der Anzahl der Quellcodezeilen – im Englischen *Lines of Code*. Im einfachsten Fall erfolgt die Schätzung nach der Formel

Umfang des Systems in LOC/Produktivität der Entwickler in LOC pro Zeiteinheit

Als Zeiteinheit werden dabei häufig Personentage – kurz PT – benutzt. Ein Personentag entspricht dabei der Arbeitsleistung, die **eine Person pro Tag** erbringen kann. Die Personentage werden weiter zu Personenmonaten (PM) und Personenjahren (PJ) zusammengefasst. Ein Personenmonat entspricht dabei ungefähr 20 Personentagen und ein Personenjahr ungefähr 10 Personenmonaten.

Ein Personenmonat entspricht nicht 30 Personentagen, da ja zum Beispiel am Wochenende eigentlich nicht gearbeitet wird. Außerdem können auch Ausfälle durch Krankheiten oder Urlaub entstehen. Deshalb geht auch das Personenjahr nicht von 12 Monaten aus, sondern lediglich von 10.



Hinweis:

Statt der Bezeichnung Personentag, Personenmonat und Personenjahr finden sich zum Teil auch noch die Bezeichnung Manntag (MT), Mannmonat (MM) und Mannjahr (MJ). Diese Bezeichnungen waren früher üblich, wurden von weiblichen Entwicklern aber als diskriminierend empfunden.

Eine Beispielrechnung könnte dann so aussehen:

Der Umfang des Systems wird auf 50000 Codezeilen geschätzt. Pro Personentag erstellen die Programmierer durchschnittlich 100 Zeilen.

$$50000 \text{ LOC} / 100 \text{ LOC pro PT} = 500 \text{ PT}$$

500 Personentage entsprechen knapp 25 Personenmonaten oder 2,5 Personenjahren. Ein Entwickler wäre also mit der Programmierung gut und gerne zwei Kalenderjahre beschäftigt. Zwei Entwickler bräuchten gemeinsam ungefähr ein Kalenderjahr.



Bitte beachten Sie:

Ein Personentag entspricht nicht immer einem Kalendertag. Wenn zum Beispiel fünf Entwickler gemeinsam einen Tag an einem Problem arbeiten, vergeht nur ein Kalendertag. Der Aufwand beträgt aber fünf Personentage, da ja fünf Personen benötigt werden. Wenn ein Entwickler sich dagegen nur die Hälfte seiner Arbeitszeit um das Projekt kümmert und die andere Hälfte seiner Arbeitszeit anderen Projekten widmet, entspricht ein Personentag für das Projekt zwei Arbeitstagen.

Auch wenn LOC-Methoden auf den ersten Blick durchaus brauchbar erscheinen mögen, liefern sie trotzdem fast nie wirklich passende Ergebnisse. Zum einen ist es extrem schwierig, die Anzahl der Codezeilen für das gesamte System vorherzusagen.

Zum anderen wird nicht berücksichtigt, dass komplexe Aufgaben zum Teil mit einigen wenigen Zeilen gelöst werden können, trotzdem aber sehr viel Zeit in Anspruch nehmen. Die Umsetzung komplizierter mathematischer Berechnung erfordert zum Beispiel nur wenig Quelltext, aber sehr viel Zeit. Eingabemasken dagegen lassen sich relativ schnell erstellen, führen aber in der Regel zu sehr viel Quelltext.

Einen anderen Ansatz als die bisher vorgestellten Verfahren verfolgen die **Gewichtungsmethoden**. Sie gehen nicht nur vom Umfang des zu erstellenden Systems aus, sondern berücksichtigen vor allem auch die Komplexität. Dazu werden verschiedene Faktoren über mathematische Formeln verknüpft.

Eine dieser Gewichtungsmethoden – die Function-Point-Methode – werden wir uns im nächsten Kapitel ein wenig genauer ansehen.

Zusammenfassung

Der Aufwand für ein Software-Produkt wird im Wesentlichen von der Quantität und der Qualität bestimmt.

Die Dauer, die Kosten, die Quantität und die Qualität hängen direkt zusammen und bestimmen sich gegenseitig.

In der Regel steht für die Entwicklung nur eine begrenzte Kapazität zur Verfügung.

Für die Aufwandsschätzung gibt es verschiedene Verfahren.

Aufgaben zur Selbstüberprüfung

- 3.1 Welche Kosten bestimmen im Wesentlichen die Entwicklungskosten für eine Software?

- 3.2 Was stellt das Teufelsquadrat dar?

- 3.3 Welche drei Schätzmethoden für den Aufwand kennen Sie bisher? Beschreiben Sie die Methoden bitte kurz.

- 3.4 Was verstehen Sie unter einem Personenmonat? Entspricht ein Personenmonat einem Kalendermonat?

4 Die Function-Point-Methode

In diesem Kapitel beschäftigen wir uns mit der Function-Point-Methode⁵.

4.1 Ermittlung der Function Points

Bei der Function-Point-Methode wird der Umfang des zu erstellenden Systems – die **Functional Size** – abgeschätzt und in **Function Points** (fp) festgehalten. Den Ausgangspunkt der Schätzung bilden dabei die **Elementarprozesse**. Dabei handelt es sich um kleine, in sich abgeschlossene Prozesse, die aus Sicht des Anwenders oder eines anderen Programms für die Arbeit mit der Anwendung sinnvoll sind. Für unsere Anwendung Autovermietung 2030 sind zum Beispiel das Anlegen von Kunden oder das Anlegen von Fahrzeugen solche Elementarprozesse.

Grundsätzlich werden bei den Elementarprozessen drei Typen unterschieden:

Eingabe (external input, EI)

Hier werden vor allem die Daten berücksichtigt, die das System einlesen muss. Dazu gehören nicht nur Informationen, die per Hand eingegeben werden, sondern zum Beispiel auch Daten, die über einen Scanner oder aus einer Datei gelesen werden.



Ein Scanner ist ein Eingabegerät für das maschinelle Einlesen von Daten. Scanner arbeiten ähnlich wie Fotokopierer.

Bei unserer Autovermietung würden zum Beispiel die Daten zu den Kunden und auch die Daten zu den Fahrzeugen zu den Eingabedaten gehören.

Ausgabe (external output, EO)

Hier werden die Daten berücksichtigt, die das System ausgeben muss. Dazu zählen nicht nur Ausgaben auf dem Bildschirm, sondern auch Druckausgaben und so weiter.

Bei der Autovermietung würden zum Beispiel die Listen für die verschiedenen Fahrzeuge bei den Ausgabedaten aufgezählt.

Abfrage (external inquiry, EQ)

Zu den Abfragen gehören alle Prozesse, die auf die gespeicherten Daten zugreifen, aber keine Veränderungen an den Daten vornehmen. Ein typisches Beispiel für eine Abfrage ist die Suche nach einem Kunden. Hier wird auf die Kundendaten zugegriffen, aber keine Veränderung an den Daten vorgenommen.

Neben den Elementarprozessen werden außerdem noch die Daten betrachtet. Hier wird zwischen **internen Datenbeständen** (*internal logical file*, ILF) und **externen Datenbeständen** (*external interface file*, EIF) unterschieden. Zu den internen Datenbeständen gehören Daten, die vom System angelegt oder verändert werden – zum Beispiel Daten für einen Kunden oder – für unsere Autovermietung – die Daten zu einem bestimmten

5. Übersetzt bedeutet Function-Point so viel wie „Funktionspunkt“.

Auto. Zu den externen Datenbeständen gehören Daten, die nicht direkt vom System selbst angelegt und verändert werden, aber vom System benötigt werden. Das ist zum Beispiel dann der Fall, wenn ein Buchhaltungsprogramm auf Preise in einer Artikeldatenbank zugreift und die Preise für die Artikel nicht in dem Buchhaltungsprogramm selbst gepflegt werden.

Für jede einzelne Kategorie gibt es einen minimalen, mittleren und maximalen Punktwert. Über diesen Punktwert wird beschrieben, wie komplex die Anforderungen sind.

Tab. 4.1: Punktwerte

Elementarprozesse/Datenbestände	Minimal	mittel	maximal
Eingabe	3	4	6
Ausgabe	4	5	7
Abfrage	3	4	6
interner Datenbestand	7	10	15
externer Datenbestand	5	7	10

Die Zuordnung der Punkte zu den einzelnen Elementarprozessen und Datenbeständen der Anwendung erfolgt anhand von **Komplexitätsregeln**.

Die Komplexitätsregeln legen unter anderem fest, wie viele Daten wie verarbeitet werden müssen. Wie die Vergabe genau erfolgt, wollen wir uns hier aber nicht weiter ansehen.



In der Praxis werden aber vielfach vereinfachte Verfahren eingesetzt. Das Rapid-Verfahren ordnet zum Beispiel einer Eingabe 4 Function Points zu, einer Ausgabe 5 Function Points, einer Abfrage 4 Function Points, einem internen Datenbestand 7 Function Points und einem externen Datenbestand 5 Function Points.

Im letzten Schritt werden die Function Points aller Elementarprozesse und Datenbestände summiert.

Schauen wir uns die einzelnen Schritte jetzt einmal am Beispiel der Autovermietung an. Grundlage sind dabei die Funktionen aus dem Lastenheft.

Zu den **Eingaben** zählen zum Beispiel

- das Anlegen eines Kunden,
- das Ändern eines Kunden,
- das Anlegen eines Fahrzeugs,
- das Ändern eines Fahrzeugs,
- das Eingeben der Kundendaten bei der Vermietung,
- das Einlesen der Fahrzeugdaten bei der Vermietung und
- das Eingeben der Fahrzeugdaten für die Reservierung.

Das Eingeben der Kundendaten und das Einlesen der Fahrzeugdaten bei der Rückgabe zählen wir nicht noch einmal auf, da es sich genauso umsetzen lässt wie das Eingeben beziehungsweise Einlesen der entsprechenden Daten bei der Vermietung. Das Eingeben der Fahrzeugdaten bei der Reservierung müssen wir dagegen einzeln zählen, da ja hier die Fahrzeuge gar nicht vorhanden sind. Die entsprechenden Daten könnten also nicht maschinell eingelesen werden, sondern müssten per Hand erfasst werden.



Wenn Ihnen nicht mehr genau klar ist, was sich hinter den einzelnen Begriffen verbirgt und wie die Funktionen umgesetzt werden sollen, sehen Sie sich bitte noch einmal das Lastenheft der Autovermietung an. Sie finden es in Kapitel 2.4.

Damit hätten wir also insgesamt sieben Eingaben. Für jede Eingabe setzen wir nach dem Rapid-Verfahren 4 Function Points an. Damit hätten wir dann insgesamt für diesen Bereich 28 Function Points. Tabellarisch lassen sich die Ergebnisse so darstellen:

Tab. 4.2: Die Function Points für die Eingabe

Elementarprozess	Bereich	Function Points
Anlegen eines Kunden	Eingabe (EI)	4
Ändern eines Kunden	Eingabe (EI)	4
Anlegen eines Fahrzeugs	Eingabe (EI)	4
Ändern eines Fahrzeugs	Eingabe (EI)	4
Eingeben der Kundendaten bei der Vermietung	Eingabe (EI)	4
Einlesen der Fahrzeugdaten bei der Vermietung	Eingabe (EI)	4
Eingeben der Fahrzeugdaten bei der Reservierung	Eingabe (EI)	4
Summe		28

Nach dem gleichen Verfahren würden jetzt die Werte für die anderen Kategorien ermittelt. Bei den Ausgaben werden zum Beispiel die Anzeige von Kunden- und Fahrzeugdaten, die Liste und der Mietvertrag erfasst, bei den Abfragen das Suchen nach Kunden- und Fahrzeugdaten und bei den internen Datenbeständen die Kunden-, Fahrzeug-, Ausleih- und Reservierungsdaten. Externe Datenbestände sind in unserem Beispiel nicht vorhanden.

Die Tabelle mit den Function Points könnte dann – etwas vereinfacht – so aussehen:

Tab. 4.3: Die Function Points für die Autovermietung

Elementarprozess/Datenbestand	Bereich	Function Points
Anlegen eines Kunden	Eingabe (EI)	4
Ändern eines Kunden	Eingabe (EI)	4
Anlegen eines Fahrzeugs	Eingabe (EI)	4
Ändern eines Fahrzeugs	Eingabe (EI)	4

Elementarprozess/Datenbestand	Bereich	Function Points
Eingeben der Kundendaten bei der Vermietung	Eingabe (EI)	4
Einlesen der Fahrzeugdaten bei der Vermietung	Eingabe (EI)	4
Eingeben der Fahrzeugdaten bei der Reservierung	Eingabe (EI)	4
Anzeigen der Kundendaten	Ausgabe (EO)	5
Anzeigen der Fahrzeugdaten	Ausgabe (EO)	5
Anzeigen des Mietvertrags	Ausgabe (EO)	5
Anzeigen der Liste	Ausgabe (EO)	5
Suchen nach Kundendaten	Abfrage (EQ)	4
Suchen nach Fahrzeugdaten	Abfrage (EQ)	4
Kundendaten	Interne Datenbestände (ILF)	7
Fahrzeugdaten	Interne Datenbestände (ILF)	7
Ausleihdaten	Interne Datenbestände (ILF)	7
Reservierungsdaten	Interne Datenbestände (ILF)	7
Summe		84

Insgesamt liefert die Bewertung also 84 Function Points.

4.2 Ermittlung der Entwicklungsdauer

Jetzt stellt sich die Frage: „Wie finde ich über die Function Points die Entwicklungsdauer heraus?“ Denn darum ging es ja ursprünglich.

Antwort auf diese Frage gibt ein Vergleich der Function Points für das geplante Projekt mit bereits abgeschlossenen Projekten. Im einfachsten Fall können Sie dann in einer Tabelle nachsehen, wie lange die Entwicklung der neuen Software voraussichtlich dauern wird. Dieser Vergleich ist aber nur dann möglich, wenn für jedes bereits abgeschlossene Projekt sowohl die Function Points als auch die Dauer in Personenmonaten oder Personenjahren festgehalten wird.

Nehmen wir einmal an, unsere Beispieldfirma SoftFix GmbH hat bereits zahlreiche Projekte durchgeführt und entsprechende Aufzeichnungen zur Dauer und den Function Points gemacht. Die Tabelle mit diesen Daten könnte dann so aussehen:

Tab. 4.4: Zuordnung der Function Points zur Dauer

Function Points	Dauer in Personenmonaten
50	4
100	7
200	12
250	15
500	31

Die Berechnung der Function Points für die Autovermietung hat den Wert 84 ergeben. Nach der vorigen Tabelle würde sich damit eine ungefähre Entwicklungsdauer zwischen fünf und sechs Personenmonaten ergeben.

Die Zuordnung kann auch grafisch in Form einer Kurve erfolgen:

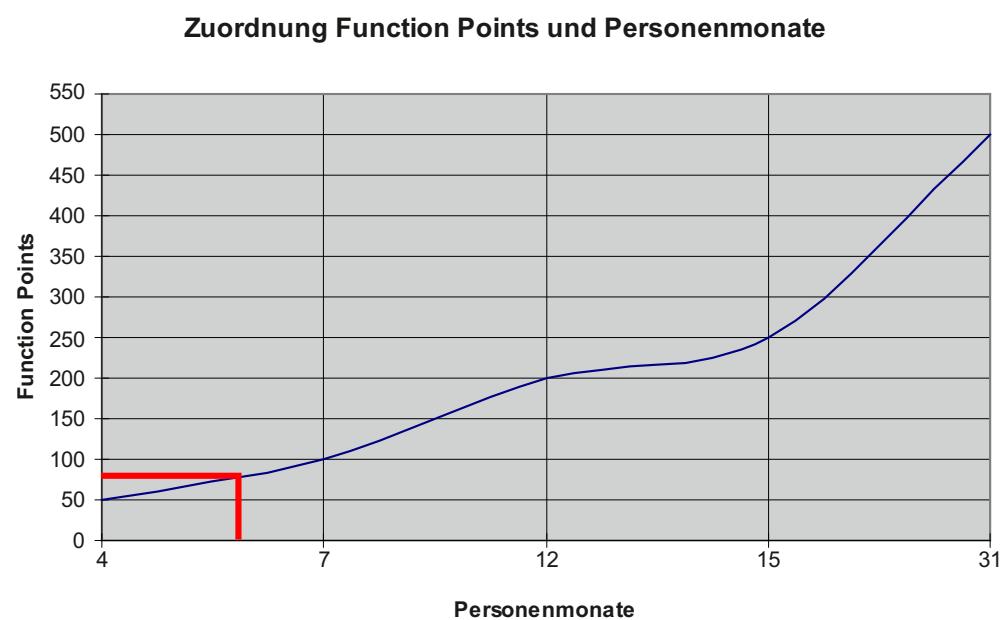


Abb. 4.1: Zuordnung der Function Points zu den Personenmonaten durch eine Kurve

Dazu ziehen Sie von der Achse mit den Funktion Points (links im Diagramm) eine waagerechte Linie bis zur Kurve. Am Schnittpunkt der Linie mit der Kurve ziehen Sie dann eine senkrechte Linie nach unten zur Achse für die Personenmonate. In der vorherigen Abbildung sind diese Linien für unsere Autovermietung rot eingetragen.

Schwierig wird es, wenn überhaupt keine Aufzeichnungen vorliegen oder wenn es sich um das allererste Projekt handelt, das ein Unternehmen durchführt. Dann gibt es ja keinerlei Vergleichswerte.

In solchen Fällen können Sie auf Tabellen von anderen Unternehmen zugreifen – zum Beispiel von IBM. In dieser Tabelle finden sich unter anderem folgende Zuordnungen:

Tab. 4.5: Zuordnung der Function Points zur Dauer nach IBM⁶

Function Points	Dauer in Personenmonaten
50	5
100	8
150	11
200	14
250	17
500	36
1000	76
2500	245

Grafisch dargestellt sieht die Tabelle von IBM so aus:

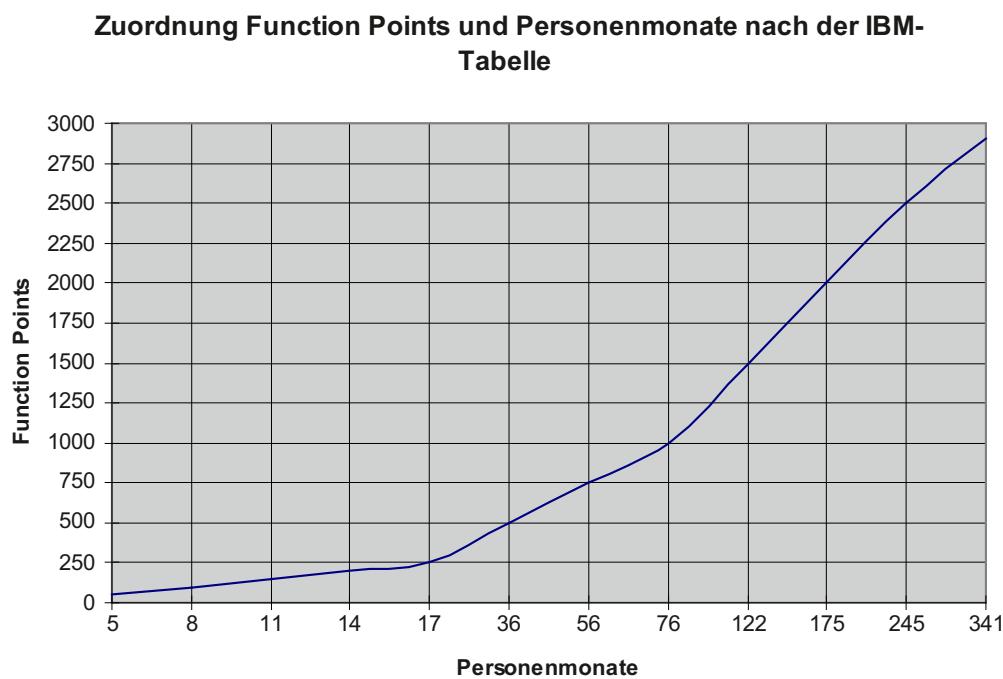


Abb. 4.2: Die Kurve zur IBM-Tabelle

Die abweichenden Werte in den vorigen Tabellen ergeben sich durch den Einsatz unterschiedlicher Werkzeuge, unterschiedlicher Methoden und auch durch die unterschiedliche Produktivität eines Teams. Die Tabellen spiegeln also immer auch die Rahmenbedingungen wider.

6. Nach Balzert.

Fassen wir die wesentlichen Schritte der Function-Point-Methode noch einmal zusammen:

1. Ermitteln Sie die Elementarprozesse und die Datenbestände.
2. Bewerten Sie die einzelnen Prozesse und Datenbestände.
3. Zählen Sie die Werte zusammen. Sie erhalten dann die Function Points.
4. Ordnen Sie die Function Points über eine Tabelle oder eine Kurve den Personenmonaten zu.

Wie Sie selbst gesehen haben, ist die Ermittlung der Function Points bereits bei vergleichsweise kleinen Projekten sehr aufwendig. Außerdem liefert die Methode auch nur dann wirklich verlässliche Ergebnisse, wenn die Anforderungen genau feststehen. Und das ist in dieser frühen Phase ja oft noch gar nicht der Fall. Das heißt, auch bei der Function-Point-Methode hat wieder die Erfahrung des Schätzers Einfluss auf das Ergebnis.

Und hier zeigt die Erfahrung: In der Regel sind die ersten Einschätzungen zu niedrig – gleich, welche Methode nun zum Einsatz kommt. Hinterfragen Sie daher Ihre ersten Einschätzungen kritisch und rechnen Sie – wenn Sie ganz sichergehen wollen – noch zusätzliche Zeit hinzu.

Eine weitere Variante, um zu optimistische Planungen zu vermeiden, ist die Ermittlung über den optimistischen Wert, den wahrscheinlichen Wert und den pessimistischen Wert. Rechnen Sie diese Werte zusammen und teilen Sie das Ergebnis durch 3.



Beispiel 4.1:

Für die Entwicklung der Software für die Autovermietung werden als optimistischer Wert 5 Personenmonate angenommen, als wahrscheinlicher Wert 6 Personenmonate und als pessimistischer Wert 9 Personenmonate – zusammen also 20. Geteilt durch 3 ergibt sich dann ein Wert von 6,6 Personenmonaten.

Neben der Function-Points-Methode gibt es auch noch andere Verfahren, um den Aufwand zu schätzen. Dazu gehören zum Beispiel die Data-Point- und die Object-Point-Methode. Hier stehen nicht die Funktionen einer Anwendung im Mittelpunkt, sondern die Daten beziehungsweise die Objekte. Weitere Verfahren sind COCOMO (**Constructive Cost Model**) und COCOMO II. Damit wollen wir uns hier aber nicht im Detail beschäftigen.

Exkurs: Die optimale Entwicklungsdauer

Wenn man bei der Entwicklungsdauer nur die Personenmonate berücksichtigt, lässt sich leicht folgende Rechnung aufstellen:

Ich benötige für ein Projekt 100 Personenmonate. Ein Mitarbeiter braucht also für die Umsetzung ungefähr 100 Monate und 100 Mitarbeiter können das Produkt in einem Monat fertigstellen. Je mehr Mitarbeiter ich einsetze, desto schneller liegt das Endprodukt vor.

Dabei handelt es sich aber um eine „Milchmädchenrechnung“. Denn: Je mehr Mitarbeiter eingesetzt werden, desto höher ist der Aufwand für die Organisation. Auf das Beispiel von oben bezogen: Es ist kaum möglich, die Arbeit von 100 Mitarbeitern in einem Monat so zu koordinieren, dass kein großes Durcheinander entsteht.

Daher gibt es Formeln für die Berechnung der **optimalen Entwicklungsdauer**. Diese optimale Entwicklungsdauer berücksichtigt, dass sich Entwicklungszeiten eben nicht beliebig durch weitere Mitarbeiter verkürzen lassen. Für einen Aufwand von 100 Personenmonaten ergibt sich zum Beispiel eine optimale Entwicklungsdauer von knapp elf Monaten.

Wie die Formeln im Detail aussehen, wollen wir Ihnen hier nicht weiter vorstellen.

Aus der optimalen Entwicklungsdauer können Sie dann in einem weiteren Schritt die durchschnittliche Teamgröße errechnen. Die entsprechende Formel lautet:

$$\text{Teamgröße} = \text{Aufwand in PM} / \text{optimale Entwicklungsdauer}$$

Für unser Beispiel wäre die durchschnittliche Teamgröße dann neun Mitarbeiter (100 / 11; ganz exakt wären es 9,09 Mitarbeiter).

So viel zur optimalen Entwicklungsdauer.

Im nächsten Kapitel werden wir uns ansehen, wie Sie die Entwicklung im Detail über das Projektmanagement planen.

Zusammenfassung

Die Function-Point-Methode gewichtet unterschiedliche Faktoren bei der Entwicklung. Sie berücksichtigt nicht nur allein den Umfang, sondern auch die Komplexität.

Über Tabellen oder Kurven kann aus den Function Points der Aufwand in Personenmonaten abgelesen werden.

Aufgaben zur Selbstüberprüfung

- 4.1 Wie wird der Umfang des zu erstellenden Systems bei der Function-Points-Methode genannt?

- 4.2 Über welche fünf Kategorien werden die Function Points dargestellt?

- 4.3 Wie lässt sich die durchschnittliche Teamgröße berechnen?

5 Projektmanagement

In diesem Kapitel unternehmen wir einen Ausflug in das Projektmanagement.

Bitte beachten Sie:

Wir konzentrieren uns hier vor allem auf einen Teilbereich des Projektmanagements – auf die Projektplanung. Vollständige Einführungen in das Projektmanagement finden Sie zum Beispiel in den Büchern im Literaturverzeichnis.



Nachdem jetzt aus den bisherigen Schritten klar ist, welche Anforderungen an die Software gestellt werden, und eine – wenn auch vielleicht sehr grobe – Schätzung des Aufwands erfolgt ist, müssen die einzelnen Tätigkeiten im Detail geplant werden. Hier geht es um die Frage: „Wer macht wann was?“

Aus der Aufwandsschätzung allein lässt sich diese Frage nicht beantworten. Hier ist zwar möglicherweise klar, bis wann die Software grundsätzlich fertiggestellt werden kann. Es finden sich aber keinerlei Aussagen, wie dieser Termin erreicht werden kann und wie die einzelnen Teilschritte dabei koordiniert werden müssen. Diese Aussagen sind nur über **Projektpläne** möglich.

Grundsätzlich sollten Sie Software immer als Projekt betrachten und auch die entsprechenden Methoden zur Planung einsetzen. Auch bei einem sehr kleinen Projekt, das Sie im Alleingang durchführen, benötigen Sie einen Rahmen für die Orientierung. Größere Projekte mit mehreren Personen lassen sich ohne vernünftiges Projektmanagement gar nicht abwickeln. Stellen Sie sich einfach nur einmal vor, Sie müssen die Arbeit von zehn Personen koordinieren. Das geht nur mit sorgfältiger Planung.



5.1 Das grundsätzliche Vorgehen

Die Planung eines Projekts erfolgt grundsätzlich vom Groben zum Feinen. Zunächst einmal legen Sie fest, welche Phasen im Projekt durchlaufen werden müssen, dann definieren Sie die groben Ziele jeder einzelnen Phase und anschließend legen Sie fest, wie Sie diese Ziele erreichen wollen. Die Detailplanung für die einzelnen Phasen erfolgt dabei in der Regel erst dann, wenn die Phase begonnen wird, beziehungsweise kurz vor dem Beginn der Phase.

Diese Form der Planung wird auch **rollierende Korridorplanung** genannt.



Der erste Schritt – die Festlegung der verschiedenen Phasen – ist bei der Software-Entwicklung vergleichsweise einfach. Hier können Sie für nahezu jedes Projekt eine grobe Einteilung in die Phasen Planung, Analyse, Entwurf, Implementierung, Test sowie Betrieb und Wartung vornehmen.

Im nächsten Schritt definieren Sie dann die **groben Ziele** für die Phasen. Ein solches Grobziel kann zum Beispiel die Machbarkeitsstudie sein. Der Weg zum Erreichen eines Grobziels wird über Meilensteine, Aktivitäten und Arbeitspakete gesteuert. Was sich genau hinter diesen Begriffen verbirgt, erfahren Sie gleich.

Danach erfolgt dann die **Termin- und Kapazitätsplanung**. Hier werden für die einzelnen Aktivitäten der Beginn, die Dauer, das Ende und der oder die Verantwortlichen festgelegt. Dabei müssen auch Abhängigkeiten zwischen den einzelnen Aktivitäten berücksichtigt werden.

Als Endergebnis halten Sie einen Projektplan in den Händen, der sowohl den zeitlichen Ablauf als auch die Abhängigkeiten der einzelnen Aktivitäten abbildet.

Schauen wir uns nun die einzelnen Schritte im Detail an.

5.2 Meilensteine, Aktivitäten und Arbeitspakete

Die **Meilensteine** stellen wichtige Zwischenergebnisse in einer Phase dar. Um das Grobziel Machbarkeitsstudie zu erreichen, müssen zum Beispiel das Lastenheft, die Kalkulation und auch der erste grobe Projektplan erstellt werden. Diese drei Bestandteile könnten also die Meilensteine auf dem Weg zur Machbarkeitsstudie bilden.



Ob und wie viele Meilensteine Sie in einer Phase festlegen, lässt sich nicht generell sagen. So kann eine Phase auch nur aus einem einzigen Meilenstein bestehen – dem Grobziel der Phase. Sie müssen immer im Einzelfall entscheiden, welche Meilensteine für Ihr Projekt wichtig sind.

Das Erreichen eines Meilensteins muss messbar sein. Ein Meilenstein wie „Das Lastenheft soll nahezu fertig sein“ ist zum Beispiel nur sehr eingeschränkt zu gebrauchen. Hier fehlt der Maßstab „Was ist nahezu fertig?“. Reichen 70 Prozent? Oder muss das Lastenheft zu 80 Prozent fertiggestellt sein?

Die **Aktivitäten** einer Phase sind die Arbeiten, die zum Erreichen eines Meilensteins zwingend erforderlich sind. Sie lassen sich in der Regel direkt aus den Meilensteinen ableiten.

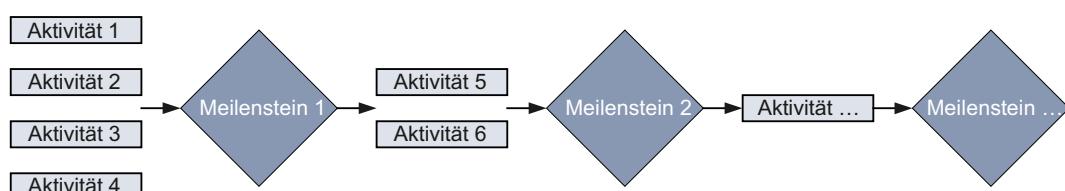


Abb. 5.1: Meilensteine und Aktivitäten

Damit der Meilenstein „Lastenheft erstellt“ erreicht wird, müssten zum Beispiel folgende Aktivitäten komplett ausgeführt worden sein:

- Fragebögen entwickeln und verschicken,
- Fragebögen auswerten,
- Brainstorming-Workshops durchführen und auswerten,

- Storyboards entwerfen und auswerten,
- Interviews vorbereiten und durchführen,
- Interviews auswerten,
- Stellenbeschreibungen auswerten,
- Informationen zusammenfassen und
- funktionale Anforderungen definieren.

Da die Planung auf Ebene der einzelnen Aktivitäten in der Regel aber gar nicht erforderlich ist, können Sie verschiedene Aktivitäten zu **Arbeitspaketen** zusammenfassen. Mögliche Arbeitspakete für unser Beispiel wären

- alle Aktivitäten, die sich mit dem Sammeln von Informationen beschäftigen,
- alle Aktivitäten, die sich mit dem Auswerten von Informationen beschäftigen, und
- das Erstellen des Lastenhefts selbst.

Eine andere Variante, Arbeitspakete zu bilden, ist die Zuordnung zu einem bestimmten Mitarbeiter beziehungsweise zu einer bestimmten Rolle.

Über **Rollen** werden bei der Software-Entwicklung die Aufgaben, Verantwortlichkeiten und Kompetenzen einer Person definiert. Eine Rolle kann von mehreren Personen eingenommen werden. Eine typische Rolle wäre zum Beispiel Programmierer. Andere Rollen sind Kunde, Manager oder Tester.



Wichtig ist vor allem, dass das Arbeitspaket eine in sich geschlossene Aufgabenstellung darstellt. Ein Arbeitspaket „Informationen und Lastenheft“ wäre für das Projekt Autovermietung 2030 zum Beispiel zu grob, da es zwei Aufgabenstellungen zusammenfasst.

Aktivitäten und Arbeitspakete werden häufig auch als **Vorgang** bezeichnet.



Hinweis:

Neben dem zu groben Planen werden Projekte häufig auch zu detailliert geplant. In unserem Beispiel wäre eine einzelne Planung für die verschiedenen Aktivitäten der Informationssammlung viel zu zeit- und arbeitsaufwendig. Häufig lohnt sich eine zu detaillierte Planung auch gar nicht. So ist es für unsere Autovermietung vor allem wichtig, dass die Informationen zu einem bestimmten Zeitpunkt vorliegen. Welche Aktivität dabei genau wann durchgeführt wird, spielt keine besondere Rolle.

5.3 Die Termin- und Kapazitätsplanung

Nachdem Sie Meilensteine, Aktivitäten und Arbeitspakete festgelegt haben, können Sie jetzt mit der Terminplanung beginnen. Dabei stehen folgende Fragen im Mittelpunkt:

- Wie lange dauert der Vorgang?
- Wann kann die Arbeit frühestens begonnen werden?
- Wann muss die Arbeit spätestens beendet sein?

Genau wie bei der Aufwandsschätzung für das gesamte Projekt ist auch die Zeitplanung für einzelne Vorgänge nicht ganz einfach – vor allem dann, wenn Sie keine Erfahrungen haben. Häufig wird auch hier die Dauer viel zu optimistisch geschätzt – also zu kurz. Im Endergebnis hat dann die gesamte Planung mit der Realität nichts mehr zu tun.

Tipp:

Für eine realistische Zeitplanung können Sie ähnliche Techniken benutzen, die wir Ihnen auch bei der Aufwandsschätzung vorgestellt haben. Verwenden Sie Vergleichswerte aus anderen Projekten, befragen Sie Experten oder berechnen Sie die Dauer aus dem optimistischen Wert, dem realistischen Wert und dem pessimistischen Wert.

Im einfachsten Fall lässt sich die Zeitplanung nach folgender Formel durchführen:

$$\text{frühester Beginn} + \text{Dauer} = \text{Ende}$$

Beispiel 5.1:



Für den Vorgang „Informationen sammeln“ werden zehn Tage angesetzt. Die Arbeit kann am 15. des Monats begonnen werden und wäre dann am 25. des Monats beendet.

Der Beginn des Projekts ist dabei nicht automatisch auch der frühestmögliche Start für sämtliche Vorgänge des Projekts. Denn häufig bestehen Abhängigkeiten zwischen den einzelnen Vorgängen.

In unserem Projekt Autovermietung 2030 könnte zum Beispiel die Auswertung der Informationen natürlich erst dann erfolgen, wenn die Informationen vorliegen. Das heißt, der Vorgang „Informationen sammeln“ muss erst abgeschlossen sein, bevor mit dem Vorgang „Informationen auswerten“ begonnen werden kann. Genauso kann auch erst dann mit dem Erstellen des Lastenhefts begonnen werden, wenn die Informationen ausgewertet sind.

Der Vorgang „Informationen auswerten“ hat also einen **Vorgänger** – das Sammeln der Informationen – und einen **Nachfolger** – das Erstellen des Lastenhefts.



Abb. 5.2: Der Vorgänger und der Nachfolger für das Auswerten der Informationen

Der Starttermin für den Vorgang „Informationen auswerten“ hängt damit direkt vom Endtermin seines Vorgängers ab. Gleichzeitig bestimmt das Ende des Vorgangs den frühesten Beginn seines Nachfolgers.

Hinweis:

Ein Vorgang kann auch mehrere Vorgänger und Nachfolger haben. Das ist vor allem bei umfangreichen Projekten eher die Regel als die Ausnahme.

Neben der **Normalfolge**, bei der das Ende eines Vorgangs den Starttermin für die Nachfolger bestimmt, gibt es unter anderem auch noch die **Anfangsfolge** und die **Endfolge**. Bei der Anfangsfolge müssen die Vorgänge gleichzeitig begonnen werden, bei der Endfolge dagegen müssen die Vorgänge gleichzeitig beendet werden.

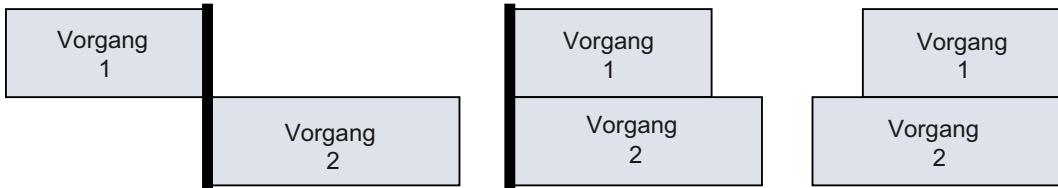


Abb. 5.3: Normalfolge (links), Anfangsfolge (Mitte) und Endfolge (rechts)

Vorgänge, von denen andere Vorgänge abhängen, werden **kritische Vorgänge** genannt. Sie müssen pünktlich beendet werden, damit sich die nachfolgenden Vorgänge und damit unter Umständen das Projektende nicht verschieben.



Alle kritischen Vorgänge zusammen bilden den **kritischen Pfad** durch das Projekt.

Bei kritischen Vorgängen sollten Sie die Terminplanung besonders sorgfältig durchführen und nach Möglichkeit auch **Pufferzeiten** einplanen. Diese Pufferzeiten sind Reservezeiten, in denen der Vorgang noch abgeschlossen werden kann, ohne dass es zu Verzögerungen im Projekt kommt.

Ein weiterer Punkt, der bei der Terminplanung unbedingt berücksichtigt werden muss, sind die zur Verfügung stehenden **Kapazitäten**. In der Regel steht Ihnen ja nur eine beschränkte Anzahl Mitarbeiter zur Verfügung. Und ein Mitarbeiter, der seine gesamte Arbeitszeit einem bestimmten Vorgang widmet, kann natürlich nicht gleichzeitig für einen anderen Vorgang eingeplant werden. Sie müssen dann entweder die Kapazitäten des Mitarbeiters auf die beiden Vorgänge verteilen oder aber die beiden Vorgänge nacheinander planen.

Hinweis:

Beim Verteilen der Kapazität eines Mitarbeiters auf mehrere Vorgänge verlängert sich automatisch die Dauer dieser Vorgänge. Der Mitarbeiter kann ja seine Arbeitsleistung nicht verdoppeln. Ein Personentag bleibt immer ein Personentag.

Neben der **Vorwärtsplanung** für ein Projekt, die über einen vorgegebenen Starttermin den Endtermin ermittelt, gibt es auch noch die **Rückwärtsplanung**. Hier wird der Endtermin fest vorgegeben und die Projektplanung ermittelt dann den Termin, an dem spätestens mit dem Projekt begonnen werden muss. Eine Rückwärtsplanung kann zum Beispiel dann erforderlich sein, wenn der Kunde einen festen Abgabetermin für die Software vorgibt.

5.4 Balkendiagramme und Netzpläne

Wie Sie gerade gesehen haben, ist bereits die Planung von kleineren Projekten oder Projektphasen nicht ganz einfach. Damit der Überblick nicht verloren geht, werden Vorgänge und ihre Abhängigkeiten häufig auch über Diagramme dargestellt.

Das **Balkendiagramm** – auch Gantt-Chart genannt – stellt dabei vor allem die zeitliche Reihenfolge der einzelnen Vorgänge dar.

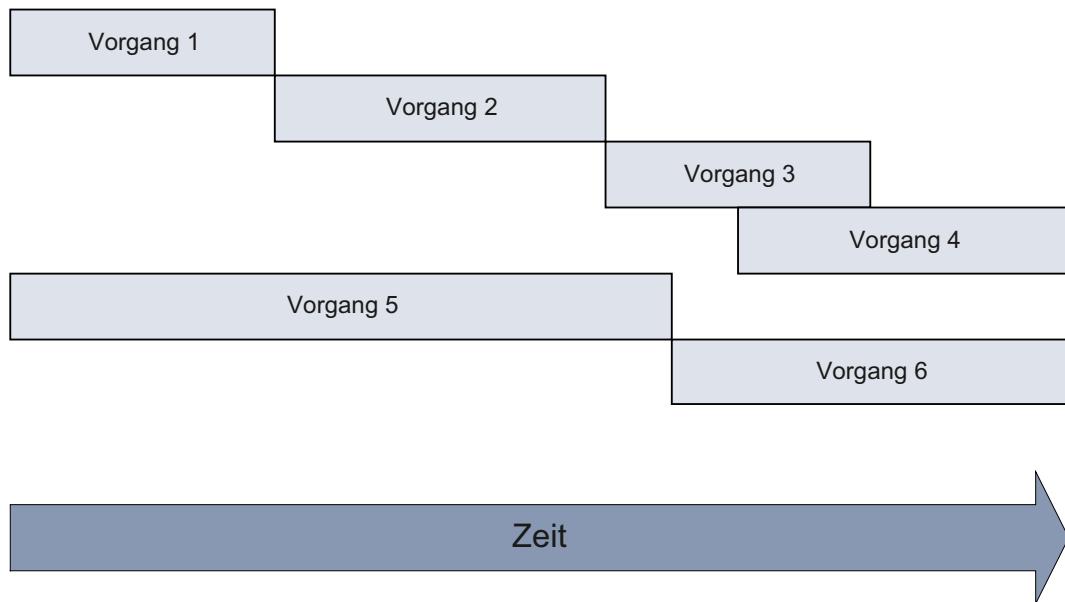


Abb. 5.4: Schematisiertes Balkendiagramm

Der Netzplan dagegen konzentriert sich auf die Darstellung der Abhängigkeiten.

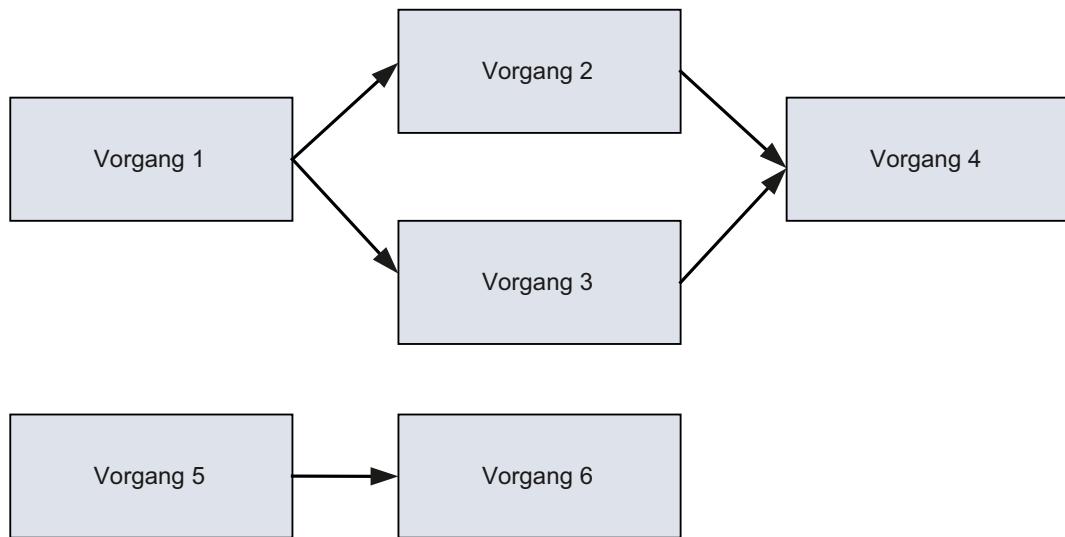


Abb. 5.5: Schematisierter Netzplan

Tipp:

Das manuelle Erstellen und Pflegen von Projektplänen kann bereits bei sehr kleinen Projekten sehr aufwendig und auch sehr fehlerträchtig werden. Wenn Sie beispielsweise die Dauer eines Vorgangs verändern, müssen Sie auch sämtliche Vorgänge auf dem kritischen Pfad anpassen. Vergessen Sie auch nur einen einzigen Vorgang, ist die gesamte Planung falsch.

Sie sollten daher auch bei kleinen Projekten eine Projektmanagement-Software wie zum Beispiel Project von Microsoft oder die Open Source Software ProjectLibre einsetzen. Diese Programme aktualisieren bei Änderungen automatisch auch alle anderen betroffenen Vorgänge und bieten Ihnen zusätzlich zahlreiche Möglichkeiten für die grafische Darstellung. Größere Projekte lassen sich ohne Software-Unterstützung kaum mit vertretbarem Aufwand planen.

5.5 Ein praktisches Beispiel

Schauen wir uns die Projektplanung jetzt am praktischen Beispiel für unser Projekt Autovermietung 2030 an. Damit das Ganze übersichtlich bleibt, beschränken wir uns hier nur auf die Planungsphase.

Im ersten Schritt legen wir die einzelnen Vorgänge, die geplante Dauer und den zuständigen Mitarbeiter fest.

Tab. 5.1: Die Vorgänge

Nummer	Vorgang	Dauer	Mitarbeiter
1	Informationen sammeln	10	Meier, Müller
2	Informationen auswerten	2	Meier
3	Lastenheft erstellen	1	Meier
4	Aufwandsschätzung durchführen	2	Geschäftsführung
5	Projektplan erstellen	1	Geschäftsführung
6	Machbarkeitsstudie erstellen	1	Geschäftsführung

Als Meilenstein dient das Erstellen der Machbarkeitsstudie – in der Tabelle durch die fett gedruckte Nummer markiert. Dieser Meilenstein bildet gleichzeitig das Grobziel der Phase.

Als Projektbeginn nehmen wir den 1. April. Dieser Termin ist damit der Starttermin für den ersten Vorgang. Die Starttermine für die weiteren Vorgänge werden durch die Abhängigkeiten bestimmt. Sie sehen für unser Projekt so aus:

Tab. 5.2: Die Abhängigkeiten

Nummer	Vorgang	abhängig von
1	Informationen sammeln	
2	Informationen auswerten	1
3	Lastenheft erstellen	2
4	Aufwandsschätzung durchführen	3
5	Projektplan erstellen	
6	Machbarkeitsstudie erstellen	3, 4, 5

Grafisch lassen sich diese Abhängigkeiten so darstellen:

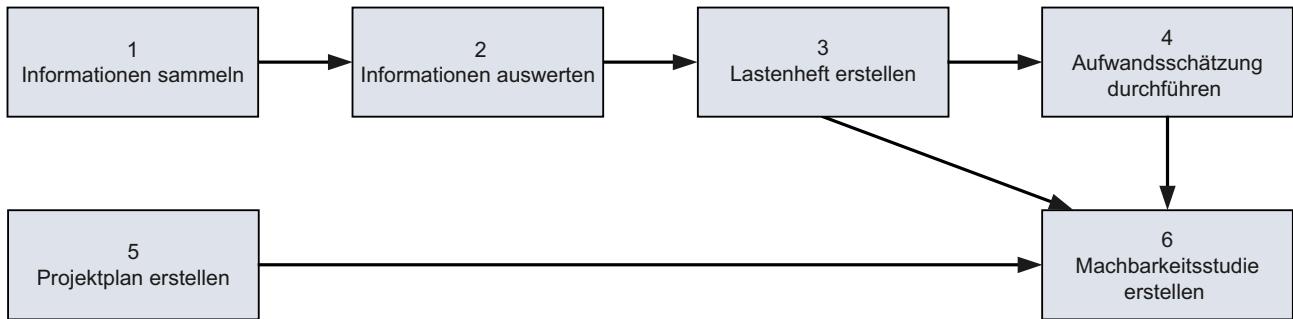


Abb. 5.6: Die Abhängigkeiten der Planungsphase

Aus den Abhängigkeiten ergeben sich dann die Termine in der folgenden Tabelle. Dabei sind die arbeitsfreien Wochenenden bereits berücksichtigt – eine Arbeit, die Ihnen zum Beispiel eine Projektmanagement-Software automatisch abnimmt.

Hinweis:

Die genauen Termine hängen von den Wochenenden ab und ändern sich daher von Jahr zu Jahr.

Tab. 5.3: Die Termine

Nummer	Vorgang	Start	Dauer	Ende
1	Informationen sammeln	1.4.	10	14.4.
2	Informationen auswerten	15.4.	2	18.4.
3	Lastenheft erstellen	19.4.	1	19.4.
4	Aufwandsschätzung durchführen	20.4.	2	21.4.
5	Projektplan erstellen	1.4.	1	1.4.
6	Machbarkeitsstudie erstellen	22.4.	1	22.4.

Gleichzeitig starten können also nur die Vorgänge 1 und 5. Bei allen anderen Vorgängen wird der Starttermin durch die Abhängigkeiten bestimmt.

Die Vorgänge und ihre Abhängigkeiten können in einer Projektmanagement-Software zum Beispiel so aussehen:

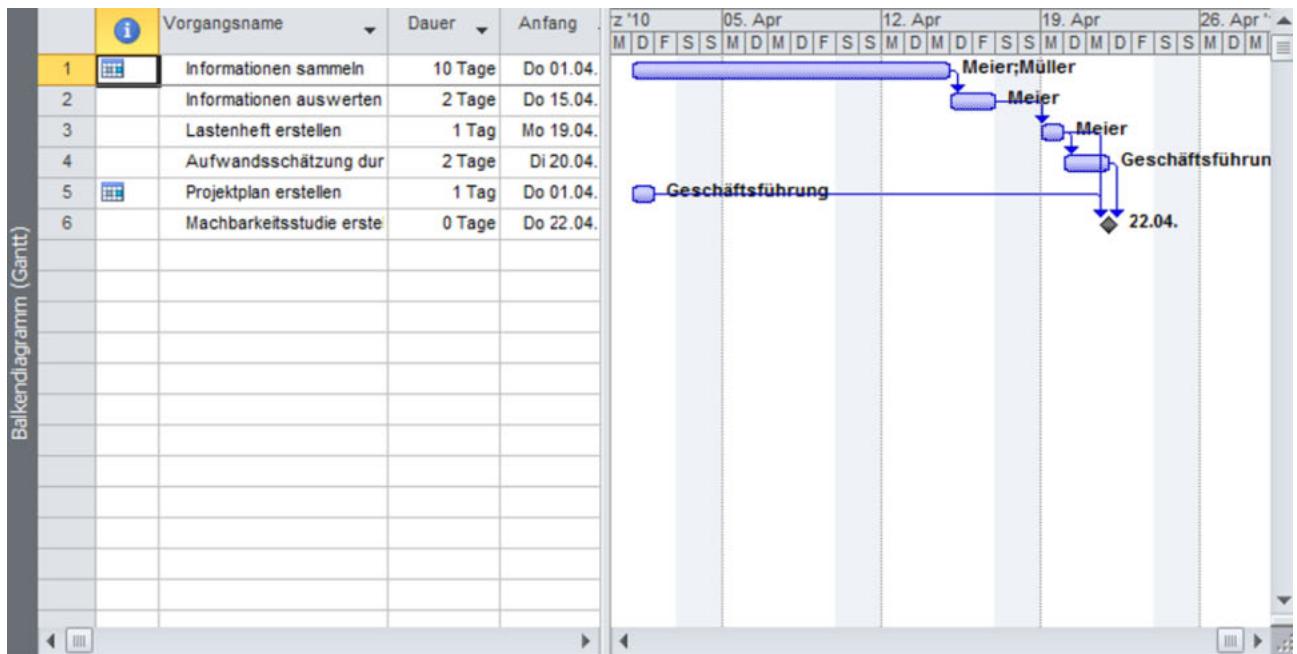


Abb. 5.7: Darstellung in einer Projektmanagement-Software (hier: Microsoft Project)

Links finden Sie die einzelnen Vorgänge mit Dauer, Anfangstermin und so weiter. Rechts daneben wird das Balkendiagramm dargestellt. Das Symbol ♦ rechts unten im Diagramm steht dabei für den Meilenstein. Rechts neben den einzelnen Vorgängen im Diagramm werden auch die eingeplanten Mitarbeiter angezeigt.

Der Netzplan, den die Projektmanagement-Software erzeugt, könnte so aussehen:

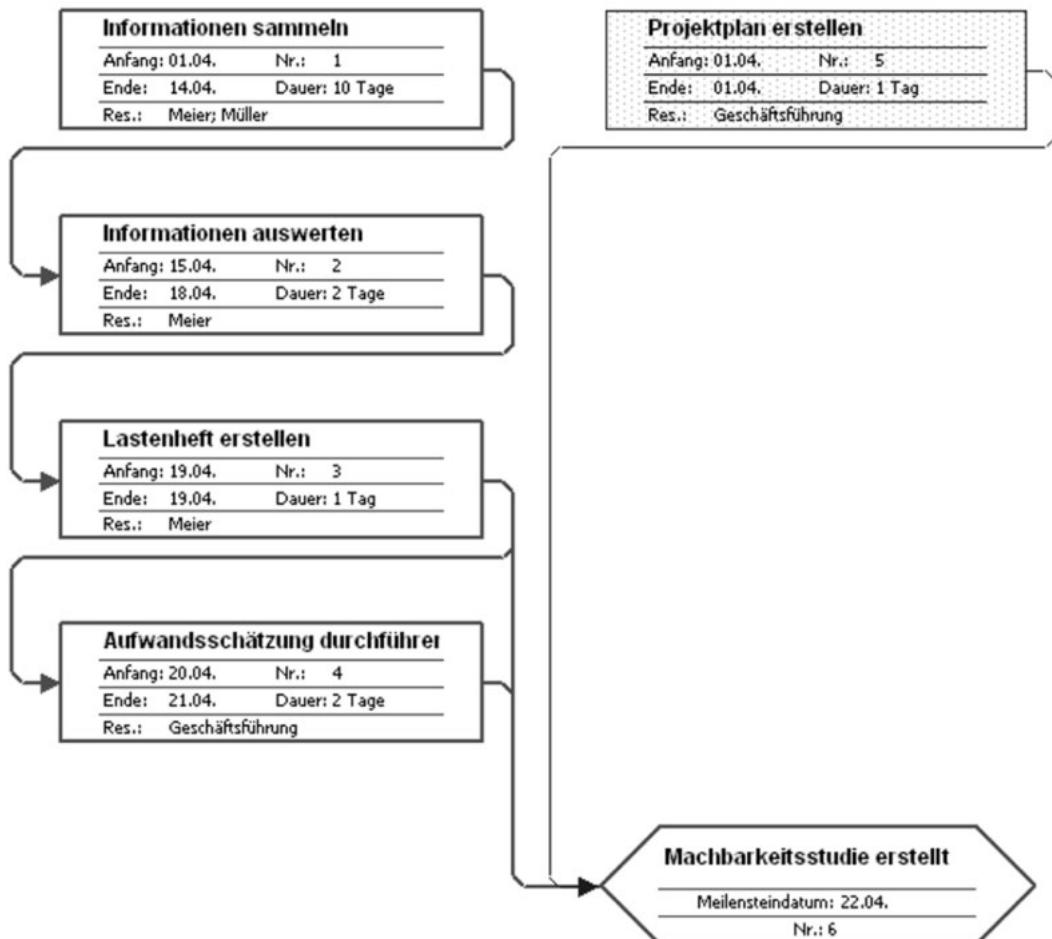


Abb. 5.8: Der Netzplan für Autovermietung 2030 (Quelle: Microsoft Project)

Für das konkrete Projektmanagement gibt es verschiedene Methoden, wie zum Beispiel PRINCE2 (*Projects in Controlled Environments*), PMBOK (*Project Management Body of Knowledge*) und ICB (*IPMA Competence Baseline*). Damit wollen wir uns hier aber nicht weiter beschäftigen.

Abschließend noch ein Hinweis:

Die Projektplanung ist kein einmaliger Vorgang, der nur zu Beginn des Projekts durchgeführt wird. Die Planung muss ständig aktualisiert und auch verfeinert werden. Nur so können Sie verlässliche Aussagen zu den Terminen machen. Bei großen und sehr großen Projekten wird das Projektmanagement daher oft von Spezialisten übernommen, die sich um nichts anderes kümmern.

Zusammenfassung

Die Projektplanung legt im Detail fest, wann einzelne Tätigkeiten durchgeführt werden müssen. Die Planung erfolgt phasenweise.

Zuerst werden die groben Ziele festgelegt. Anschließend werden Meilensteine, Aktivitäten und Arbeitspakete definiert.

Dann werden die Termine und Kapazitäten für die einzelnen Vorgänge festgelegt. Dabei müssen Abhängigkeiten zwischen den einzelnen Vorgängen berücksichtigt werden.

Aufgaben zur Selbstüberprüfung

- 5.1 Warum ist der Starttermin für ein Projekt nicht auch automatisch der Starttermin für sämtliche Vorgänge im Projekt?

- 5.2 Welche Aufgaben übernimmt ein Netzplan bei der Projektplanung?

- 5.3 Mit welcher Diagrammart wird der zeitliche Ablauf der Vorgänge dargestellt?

- 5.4 Was ist ein Meilenstein? Welche Anforderungen an einen Meilenstein müssen Sie beachten?

- 5.5 Welche Abhängigkeiten zwischen Vorgängen kennen Sie? Beschreiben Sie die einzelnen Abhängigkeiten bitte kurz.

6 Die Entscheidung

Nachdem wir jetzt die Machbarkeitsstudie für unser Projekt Autovermietung 2030 erstellt haben, können wir noch einmal die Fragen vom Beginn des Studienhefts als eine Art Checkliste der Reihe nach durchgehen.

Die Antwort auf die Frage „Was soll die Software grundsätzlich leisten?“ findet sich im Lastenheft. Ob die Anforderungen weitgehend vollständig erfasst sind, lässt sich leider nicht prüfen. Wir gehen aber einmal davon aus, dass wir nichts Wesentliches vergessen haben.

Die Anforderungen sind konsistent, widersprechen sich also nicht.

Als Entwicklungsdauer gehen wir von sechs Personenmonaten aus. Diesen Wert haben wir über die Function-Point-Methode ermittelt. Um ein wenig Reserve zu haben, rechnen wir noch einen Personenmonat dazu. Insgesamt beträgt die Entwicklungsdauer also wahrscheinlich sieben Personenmonate.

Die Entwicklungskosten berechnen wir aus den Personalkosten. Wir nehmen hier einen Wert von 5000 € pro Personenmonat an – insgesamt also 35000 €.

Hinweis:

Der Wert für die Personalkosten ist fiktiv. Die tatsächlichen Kosten pro Personenmonat hängen von sehr vielen Faktoren ab und können sowohl niedriger als auch deutlich höher sein.

Die Entwicklung lässt sich mit dem vorhandenen Personal der SoftFix GmbH durchaus durchführen. Allerdings ist das Personal auch einer der Risikofaktoren. Da die SoftFix GmbH ein vergleichsweise kleines Unternehmen ist, kann der Ausfall eines Mitarbeiters unter Umständen das gesamte Projekt verzögern. Daher hat der Geschäftsführer beschlossen, zwei weitere Entwickler als freie Mitarbeiter in Alarmbereitschaft zu versetzen. Die freien Mitarbeiter sollen im Bedarfsfall einspringen.

Bleibt noch eine Frage offen: „Lohnt sich die Erstellung überhaupt?“

Grundsätzlich lassen sich die Anforderungen auch mit vergleichsweise einfachen Datenbankprogrammen wie zum Beispiel Microsoft Access umsetzen. Da hier weniger Programmierarbeit anfällt, würde die Entwicklung wahrscheinlich kostengünstiger. Allerdings führt der Einsatz eines solchen Datenbankprogramms auch dazu, dass mögliche Weiterentwicklungen zwingend mit diesem einen Programm durchgeführt werden müssen. Stoßen Sie dabei an die Grenzen des Programms, ist automatisch auch die Entwicklung beendet – das System lässt sich nicht mehr weiter ausbauen.

Die Autovermietung hat sich daher entschlossen, auf eine möglichst flexible Lösung zurückzugreifen – auch wenn diese Lösung kurzfristig erst einmal teurer ist.

Damit ist der Startschuss für die Entwicklung gefallen und die eigentliche Arbeit am System kann beginnen. Mit dem ersten Schritt – der objektorientierten Analyse – werden wir uns im weiteren Verlauf auseinandersetzen.

7 UML-Grundlagen

Zum Abschluss dieses Studienhefts stellen wir Ihnen die Unified Modeling Language – kurz UML – im Überblick vor. Sie erfahren, was die UML überhaupt ist, und lernen auch einige Diagramme kennen.

Unternehmen wir zunächst einmal einen kleinen Ausflug in die Geschichte der UML.

7.1 Die Geschichte der UML

Objektorientierte Ansätze sind schon vergleichsweise alt – zumindest gemessen an den Maßstäben der Informationstechnik. Bereits seit 1970 existierten die ersten Versuche, die immer weiter ausgebaut, geändert oder auch komplett überarbeitet wurden. Das führte dazu, dass es um 1995 zahlreiche unterschiedliche Ansätze gab, die sich zum Teil nur sehr gering unterschieden, zum Teil aber auch völlig anders vorgingen.

Die verschiedenen Methoden benutzten auch unterschiedliche Notationen, die ebenfalls nicht sehr viel gemeinsam hatten. Kurz und gut: Es gab ein großes Durcheinander und vor allem keinen einheitlichen Standard. Missverständnisse waren an der Tagesordnung und die Entwickler mussten – je nach eingesetzter Methode – unter Umständen erst mühselig umlernen.

Das änderte sich, als im Jahre 1994 Jim Rumbaugh – einer der Väter der *Object Modeling Technique*⁷ (OMT) – und Grady Booch – der Vater des *Object Oriented Design*⁸ (OOD) – bei der Firma Rational zusammentrafen. Sie beschlossen, die Notationen ihrer beiden Methoden zur *Unified Method*⁹ (UM) zusammenzuführen.

Kurze Zeit später stieß auch noch Ivar Jacobson – der Vater des *Object Oriented Software Engineering*¹⁰ (OOSE) – dazu. Er brachte ebenfalls Elemente seiner Notation mit ein. So entstand die UML Version 0.9.

Die drei Herren Rumbaugh, Booch und Jacobson werden in Fachkreisen auch als die „drei Amigos“ bezeichnet.



Da die drei wesentlichen Ideengeber der objektorientierten Ansätze nun gemeinsam an einer Standardnotation arbeiteten, kamen auch immer mehr Unternehmen und andere Organisationen dazu. Im Jahre 1997 wurde schließlich die Version 1.1 der UML von der *Object Management Group* (OMG) als Standard verabschiedet.

Seitdem wurde die UML immer weiterentwickelt. Zurzeit ist die Version 2.5 aktuell.

7. *Object Modeling Technique* bedeutet übersetzt so viel wie „Objektmodellierungs-Technik“.

8. *Object Oriented Design* bedeutet übersetzt so viel wie „Objektorientierter Entwurf“.

9. *Unified Method* bedeutet übersetzt so viel wie „Vereinheitlichte Methode“.

10. *Object Oriented Software Engineering* bedeutet übersetzt so viel wie „Objektorientierte Software-Entwicklung“.

**Bitte beachten Sie:**

Die UML ist keine Methode für die Entwicklung von Software-Systemen, sondern eine standardisierte Notation – also ein System zur Darstellung von Informationen über Symbole und Texte. Wenn Sie so wollen, ist die UML eine formalisierte Sprache für die Beschreibung von objektorientierten Systemen. Sie bietet damit eine gemeinsame Basis bei der Entwicklung und Umsetzung.

7.2 Die Diagramme der UML

Die Modellierung mit der UML untergliedert sich grob in zwei Modelle – das statische Modell und das dynamische Modell.

Im **statischen Modell** wird vor allem die Struktur eines Systems dargestellt – also zum Beispiel, welche Objekte und Klassen es überhaupt gibt und wie Klassen untereinander in Beziehung stehen.

Im **dynamischen Modell** wird vor allem das Verhalten eines Systems beschrieben. Dazu gehört zum Beispiel die Zusammenarbeit der Klassen bei der Lösung eines konkreten Problems.

Beide Modelle arbeiten mit speziellen Diagrammen. Zum statischen Modell gehören unter anderem

- das Klassendiagramm,
- das Komponentendiagramm,
- das Paketdiagramm und
- das Verteilungsdiagramm.

Zum dynamischen Modell gehören zum Beispiel

- das Anwendungsfalldiagramm – auch *Use-Case-Diagramm*¹¹ genannt,
- das Aktivitätsdiagramm,
- das Zustandsdiagramm und
- das Sequenzdiagramm.

Die einzelnen Diagramme müssen nicht zwangsläufig für jedes System erstellt werden, sondern werden nur dann eingesetzt, wenn sie auch tatsächlich benötigt werden.

Neben diesen grundlegenden Diagrammen kennt die UML noch weitere Diagrammformen wie zum Beispiel das Zeitverlaufs- oder Timing-Diagramm zur Darstellung zeitlicher Abläufe. Mit diesen Diagrammen wollen wir uns in diesem Lehrgang aber nicht weiter beschäftigen.

Schauen wir uns einige dieser Diagramme am Beispiel eines stark vereinfachten Geldautomaten an.

11. *Use Case* ist der englische Begriff für „Anwendungsfall“.

Hinweis:

Wir geben Ihnen hier zunächst nur einen groben Überblick über die Diagramme, damit Sie schon einmal einen Vorgeschmack und einen ersten Eindruck bekommen. Später werden wir dann unsere Autovermietung im Detail mit den verschiedenen Diagrammformen darstellen.

Beginnen wir mit dem **Klassendiagramm**.

Wie Sie ja bereits wissen, wird es eingesetzt, um die Klassen in einem System und ihre Beziehungen untereinander zu beschreiben. Die Klassen werden durch ein Rechteck dargestellt. Oben steht der Name der Klasse, darunter folgen die Attribute und Methoden. Die Beziehungen zwischen Klassen werden durch verschiedene Linien dargestellt.

Zur Erinnerung:

Die Attribute werden in C# Felder genannt.



In stark vereinfachter Form haben wir diese Klassendiagramme bereits mehrfach in diesem Lehrgang benutzt – zum Beispiel bei der Vorstellung von Klassen und der Vererbung.

Das Klassendiagramm für unseren Geldautomaten könnte zum Beispiel so aussehen:

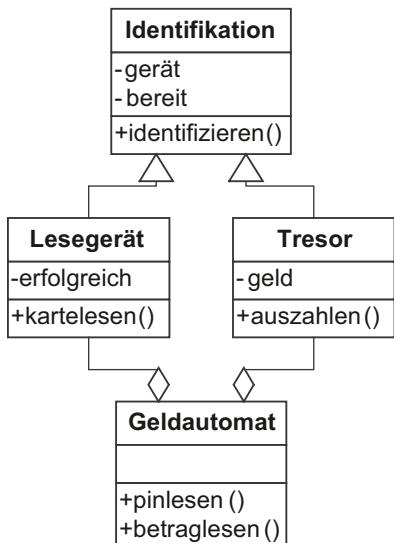


Abb. 7.1: Das Klassendiagramm des Geldautomaten

Oben in dem Diagramm befindet sich eine allgemeine Klasse `Identifikation`. Sie verfügt über das Attribut `gerät` zur eindeutigen Kennzeichnung des Geräts, für das eine Identifikation durchgeführt wird, und über das Attribut `bereit`, das festhält, ob die Identifikation möglich ist. Außerdem verfügt die Klasse über die Methode `identifizieren()`.

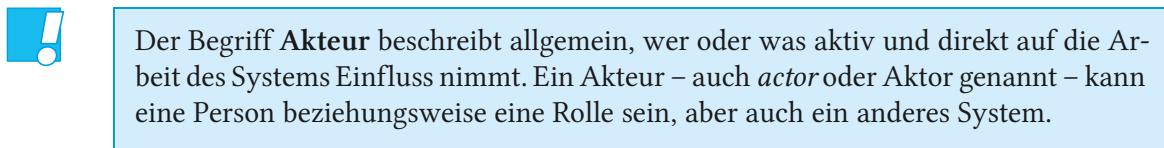
Darunter befinden sich dann die beiden Klassen `Lesegerät`, `Tresor` sowie die Klasse für den Geldautomaten.

Die Linie mit dem nicht ausgefüllten Dreieck an einem Ende kennzeichnet eine **Vererbung**. In unserem Beispiel erben die Klassen Lesegerät und Tresor zum Beispiel von der Klasse Identifikation.

Die Linien mit den nicht ausgefüllten Rauten dagegen stehen für eine **Aggregation**. Die Aggregation gibt an, dass eine Klasse in einer anderen enthalten ist. So sind zum Beispiel das Lesegerät und der Tresor Bestandteil eines Geldautomaten.

Schauen wir uns noch eine weitere Diagrammform an – und zwar das **Anwendungsfalldiagramm**.

Es stellt Beziehungen zwischen Akteuren und Prozessen dar. Die Akteure werden dabei als „Strichmännchen“ abgebildet und die Prozesse als Ellipsen.



Für den Geldautomaten könnte das Anwendungsfalldiagramm zum Beispiel so aussehen:

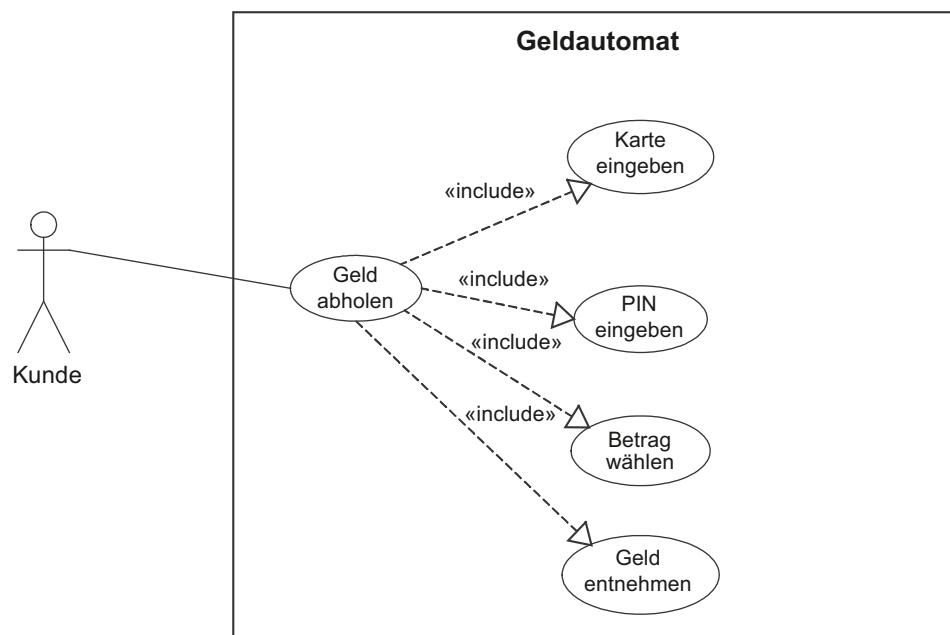


Abb. 7.2: Das Anwendungsfalldiagramm des Geldautomaten

Insgesamt werden in dem Diagramm fünf Anwendungsfälle und ein Akteur dargestellt. Die Beziehung zwischen dem Akteur und dem Anwendungsfall, an dem er beteiligt ist, wird durch eine Linie dargestellt.

Neben der einfachen Beziehung gibt es noch besondere Beziehungen – zum Beispiel **include**. Diese Beziehung drückt aus, dass ein Anwendungsfall auf einen anderen zugreift beziehungsweise dass ein Anwendungsfall Teil eines anderen Anwendungsfalls ist. In unserem Beispiel finden Sie diese Art der Beziehung rechts im Diagramm. Der Anwendungsfall „Geld abholen“ greift auf die vier anderen Anwendungsfälle zu. Dadurch wird auch die Beziehung zum Akteur hergestellt – wenn auch indirekt.

Kommen wir abschließend noch zum **Aktivitätsdiagramm**.

Das Aktivitätsdiagramm wird – wie der Name schon sagt – für die Darstellung von Aktivitäten in einem Prozess benutzt. Die Aktivitäten werden in Aktionen unterteilt. Diese Aktionen werden durch Rechtecke mit gerundeten Ecken dargestellt und durch Pfeile verbunden. So lässt sich auch die Reihenfolge der Aktionen ablesen.

Das Aktivitätsdiagramm für den Geldautomaten könnte zum Beispiel so aussehen:

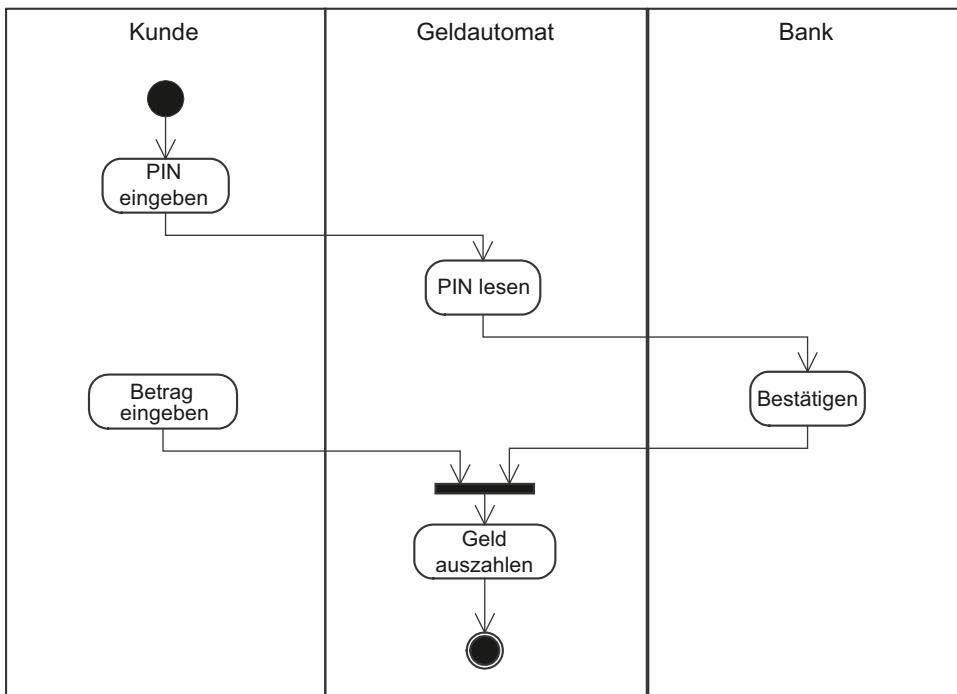


Abb. 7.3: Das Aktivitätsdiagramm des Geldautomaten

Dargestellt werden hier nur die Aktionen von der Eingabe der PIN – der *Personal Identification Number* oder persönlichen Identifizierungsnummer – bis zur Auszahlung des Geldes. Der Startpunkt befindet sich oben links im Diagramm. Im ersten Schritt gibt der Kunde seine PIN ein. Diese PIN wird vom Geldautomaten gelesen und dann bei der Bank bestätigt.

Unten im Diagramm treffen die Aktionen **Bestätigen** von der Bank und **Betrag eingeben** vom Kunden zusammen. Nur wenn diese Aktionen abgeschlossen sind, wird das Geld ausgezahlt. Danach ist der Vorgang beendet.

So viel zu den Diagrammen. Abschließend möchten wir Ihnen noch ein Werkzeug zum Arbeiten mit der UML vorstellen.

7.3 UML-Werkzeuge

UML-Diagramme per Hand zu erstellen, kann schnell zu einem mühseligen und aufwendigen Unterfangen werden. Stellen Sie zum Beispiel kurz vor der Fertigstellung fest, dass Ihnen ein Fehler unterlaufen ist, müssen Sie unter Umständen die ganze Arbeit noch einmal neu beginnen.

Auf dem Markt sind aber mittlerweile viele Programme, die Ihnen sehr viel Arbeit und Aufwand nehmen können – zum Beispiel Microsoft Visio. Es bietet Ihnen vorgefertigte Symbole für die unterschiedlichen UML-Diagramme an und ermöglicht auch eine sehr komfortable Beschriftung über Dialoge.

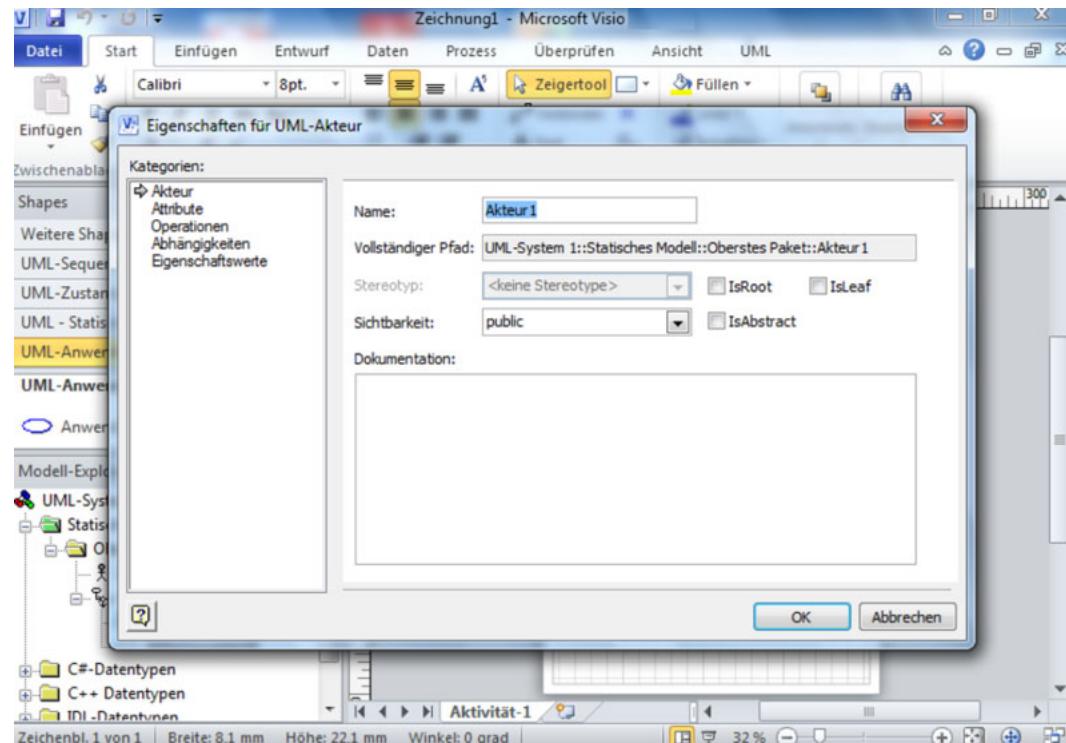


Abb. 7.4: Erstellen von Klassendiagrammen mit Microsoft Visio

Tipp:

Die meisten UML-Werkzeuge sind ausgesprochen teuer. Sie können aber zum Beispiel über die Internetseiten von Microsoft eine zeitlich befristete Testversion von Visio herunterladen.

Zusammenfassung

Für die Modellierung mit der UML werden unterschiedliche Diagramme eingesetzt – zum Beispiel das Anwendungsfalldiagramm.

Es gibt zahlreiche Programme, die die Arbeit mit der UML unterstützen.

Aufgaben zur Selbstüberprüfung

- 7.1 Handelt es sich bei UML um eine Methode zur objektorientierten Entwicklung?

- 7.2 Welche Modelle unterscheidet die UML? Was stellen die Modelle dar?

- 7.3 Nennen Sie bitte zu jedem Modell jeweils ein Diagramm.

Schlussbetrachtung

In diesem Studienheft haben Sie gelernt, welche Bedeutung eine gute Vorbereitung für die Software-Entwicklung hat. Sie wissen jetzt, wie Sie die Anforderungen an das Produkt im Lastenheft fixieren, wie Sie – zumindest eine grobe – Kostenschätzung durchführen und wie Sie die Entwicklung im Detail planen können.

Eine Methode zur Aufwandsschätzung – die Function-Point-Methode – haben Sie im Detail kennengelernt.

Wie Sie selbst gesehen haben, ist das Erstellen einer Machbarkeitsstudie recht aufwendig. Sie sollten sich aber auf keinen Fall verleiten lassen, die Studie nebenbei zu erstellen – auch wenn Sie möglichst schnell mit der Umsetzung beginnen wollen. Verlassen Sie sich nicht auf lose schwammige Absprachen mit dem Kunden und auch nicht auf Schätzungen „Pi mal Daumen“. Andernfalls ist Ärger vorprogrammiert: In der Regel wird der Aufwand nämlich unterschätzt, nur sehr selten überschätzt.

Denken Sie daran: Die Machbarkeitsstudie sichert Sie als Auftragnehmer ab. Deshalb sollten Sie sie sorgfältig und gründlich durchführen.

Zum Abschluss dieses Studienhefts haben Sie sich kurz mit der UML beschäftigt und auch schon einige UML-Diagramme kennengelernt.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

1.1 Zur Machbarkeitsstudie gehören:

- das Lastenheft,
- die Kalkulation und
- der Projektplan.

Das Lastenheft beschreibt die Anforderungen an die Software. Die Kalkulation soll die Frage „Was kostet die Entwicklung?“ beantworten. Der Projektplan soll die Frage nach der Entwicklungsdauer beantworten.

Kapitel 2

2.1 Grundsätzlich ist die Gliederung beliebig. Sie können aber zum Beispiel folgende Punkte berücksichtigen:

- Visionen und Ziele,
- Rahmenbedingungen,
- Kontext und Überblick,
- Funktionale Anforderungen und
- Qualitätsanforderungen.

2.2 Grundsätzliche Möglichkeiten sind:

- Fragebögen,
- Stellenbeschreibungen und -profile,
- Skizzen über Drehbücher,
- Interviews mit Mitarbeitern,
- Brainstorming-Workshops,
- teilnehmende Beobachtung oder
- die direkte Mitarbeit als eine Art Praktikant.

2.3 Wenn Sie die Nummerierung in 1er-Schritten durchführen, müssen Sie beim nachträglichen Einfügen neuer Punkte alle nachfolgenden Punkte ebenfalls neu nummerieren.

Kapitel 3

- 3.1 Die Entwicklungskosten werden im Wesentlichen durch die Personalkosten bestimmt.
- 3.2 Das Teufelsquadrat stellt die vier Faktoren Qualität, Quantität, Dauer und Kosten sowie ihre Abhängigkeiten dar. Änderungen an einem Faktor führen automatisch zu Änderungen an einem anderen Faktor, da die Kapazität begrenzt ist.
- 3.3
- Analogiemethode,
 - Prozentsatzmethode und
 - LOC-Varianten.

Bei der Analogiemethode werden die Anforderungen aus dem aktuellen Projekt mit bereits durchgeführten Projekten verglichen. Daraus lässt sich nach Gefühl der Aufwand ermitteln.

Bei der Prozentsatzmethode werden die prozentualen Anteile einer Entwicklungsphase aus ähnlichen Projekten zum Vergleich mit dem aktuellen Projekt benutzt. Aus dem Anteil der bereits durchgeführten Phasen kann dann der gesamte Aufwand abgeleitet werden.

Bei den LOC-Varianten wird der geschätzte Umfang des Systems in Codezeilen durch die Produktivität der Entwickler in einer Zeiteinheit geteilt.

- 3.4 Ein Personenmonat entspricht der Arbeitsleistung, die eine Person in einem Monat erbringen kann. Ein Personenmonat hat circa 20 Personentage. Er entspricht also nicht einem Kalendermonat.

Kapitel 4

- 4.1 Der Umfang des zu erstellenden Systems ist die Functional Size.

- 4.2 Die Kategorien sind:

- Eingabe,
- Ausgabe,
- Abfrage,
- interner Datenbestand und
- externer Datenbestand.

- 4.3 Die durchschnittliche Teamgröße lässt sich mit der Formel

$$\text{Teamgröße} = \text{Aufwand in PM} / \text{optimale Entwicklungsdauer}$$

berechnen.

Kapitel 5

- 5.1 Zwischen den einzelnen Vorgängen bestehen in der Regel Abhängigkeiten, die den Starttermin beeinflussen.
- 5.2 Ein Netzplan stellt im Wesentlichen die Abhängigkeiten zwischen den Vorgängen dar.
- 5.3 Der zeitliche Ablauf von Vorgängen wird mit einem Balkendiagramm – auch Gantt-Chart genannt – dargestellt.
- 5.4 Ein Meilenstein ist ein wichtiger Zwischenschritt in einer Phase oder im gesamten Projekt. Das Erreichen eines Meilensteins muss messbar sein.
- 5.5 Unterschieden werden die Normalfolge, die Anfangsfolge und die Endfolge.

Bei der Normalfolge kann ein Vorgang erst dann beginnen, wenn sein Vorgänger beendet ist.

Bei der Anfangsfolge müssen Vorgänge gleichzeitig beginnen.

Bei der Endfolge müssen Vorgänge gleichzeitig beendet sein.

Kapitel 7

- 7.1 Nein, bei der UML handelt sich nicht um eine Methode, sondern um eine Notation.
- 7.2 Die UML unterscheidet grob zwischen dem statischen und dem dynamischen Modell. Das statische Modell bildet vor allem die Struktur eines Systems ab. Im dynamischen Modell wird vor allem das Verhalten eines Systems beschrieben.
- 7.3 Zum statischen Modell gehören unter anderem:
 - das Klassendiagramm,
 - das Komponentendiagramm,
 - das Paketdiagramm und
 - das Verteilungsdiagramm.

Zum dynamischen Modell gehören zum Beispiel:

- das Anwendungsfalldiagramm – auch *Use-Case-Diagramm* genannt,
- das Aktivitätsdiagramm,
- das Zustandsdiagramm und
- das Sequenzdiagramm.

Für die richtige Antwort reicht es aus, wenn Sie jeweils ein Diagramm für die beiden Modelle angegeben haben.

B. Glossar

Actor	<i>Actor</i> ist der englische Begriff für Akteur.
Aggregation	Die Aggregation ist eine besondere Form der Assoziation zwischen Klassen. Sie drückt aus, dass eine Klasse in einer anderen enthalten ist.
Akteur	Der Begriff Akteur beschreibt allgemein, wer oder was aktiv und direkt auf die Arbeit des Systems Einfluss nimmt. Ein Akteur kann eine Person beziehungsweise eine Rolle sein, aber auch ein anderes System.
Aktion	Eine Aktion ist Teil eines Aktivitätsdiagramms.
Aktivität	Eine Aktivität ist eine Tätigkeit, die zum Erreichen eines Meilensteins in einem Projekt erforderlich ist.
Aktivitätsdiagramm	Das Aktivitätsdiagramm wird für die Darstellung von Aktivitäten in einem Prozess benutzt. Eine Aktivität wird in Aktionen unterteilt. Die einzelnen Aktionen – dargestellt durch Rechtecke mit gerundeten Ecken – werden dabei durch Pfeile verbunden. So lässt sich auch die Reihenfolge der Aktionen ablesen.
Aktor	Aktor ist eine andere Bezeichnung für Akteur.
Analogiemethode	Die Analogiemethode ist eine Schätzmethode zur Aufwandsermittlung. Hier werden die Anforderungen aus dem aktuellen Projekt mit bereits durchgeführten Projekten verglichen und nach Gefühl der Aufwand für das aktuelle Projekt geschätzt.
Anfangsfolge	Die Anfangsfolge beschreibt die Abhängigkeiten zwischen Vorgängen in einem Projekt. Die Vorgänge müssen gleichzeitig beginnen.
Angebot	Das Angebot sind – allgemein ausgedrückt – die Waren und Dienstleistungen, die ein Unternehmen auf dem Markt anbietet. Ein Angebot ist aber auch ein Handelspapier, das den konkreten Preis für eine Ware oder eine Dienstleistung nennt. Diese Angebote werden in der Regel individuell für einen potenziellen Kunden oder Auftraggeber erstellt.
Anwendungsfalldiagramm	Das Anwendungsfalldiagramm stellt Beziehungen zwischen Akteuren und Prozessen – den Anwendungsfällen – dar. Die Akteure werden dabei als „Strichmännchen“ abgebildet und die Anwendungsfälle als Ellipsen.

Arbeitspaket	Ein Arbeitspaket fasst Aktivitäten in einem Projekt zusammen. Die Zusammenfassung kann dabei nach Personen beziehungsweise Rollen oder auch nach inhaltlichen Gesichtspunkten erfolgen.
Attribut	Attribute beschreiben den Zustand eines Objekts. Bei C# werden die Attribute Felder genannt.
Aufwand	Aufwand bezeichnet in der betriebswirtschaftlichen Kosten- und Leistungsrechnung den gesamten Werteverzehr eines Unternehmens in einer Rechnungsperiode – also zum Beispiel Wareneinkäufe, Kosten für Dienstleistungen, Abgaben an den Staat und so weiter.
Ausschreibung	Eine Ausschreibung ist – allgemein ausgedrückt – ein Verfahren, um Preise für die Erbringung einer Leistung zu ermitteln. Dazu werden zum Beispiel verschiedene Anbieter aufgefordert, ein Angebot abzugeben, oder das Angebot wird öffentlich zugänglich gemacht. Für öffentliche Auftraggeber wie Länder oder Gemeinden gibt es festgelegte Ausschreibungsverfahren.
Balkendiagramm	Ein Balkendiagramm wird im Projektplan zur Darstellung der zeitlichen Reihenfolge benutzt.
Brainstorming	<i>Brainstorming</i> ist eine Technik, bei der spontane Einfälle zu einem Thema gesammelt werden. Die Strukturierung erfolgt in der Regel erst nach dem Sammeln.
Durchführbarkeitsstudie	Durchführbarkeitsstudie ist eine andere Bezeichnung für die Machbarkeitsstudie.
Dynamisches Modell (UML)	Im dynamischen Modell der UML wird vor allem das Verhalten eines Systems beschrieben. Dazu werden unter anderem folgende Diagramme verwendet: <ul style="list-style-type: none"> • das Anwendungsfalldiagramm – auch <i>Use-Case-Diagramm</i> genannt, • das Aktivitätsdiagramm, • das Zustandsdiagramm und • das Sequenzdiagramm.
Elementarprozess	Elementarprozesse werden bei der Schätzung über die Function-Point-Methode eingesetzt. Es handelt sich um kleine, in sich abgeschlossene Prozesse, die aus Sicht des Anwenders oder eines anderen Programms für die Arbeit mit der Anwendung sinnvoll sind.
Endfolge	Die Endfolge beschreibt die Abhängigkeiten zwischen Vorgängen in einem Projekt. Die Vorgänge müssen gleichzeitig enden.

Function-Point-Methode	Die <i>Function-Point</i> -Methode ist eine Gewichtungsmethode zur Aufwandsschätzung. Hier wird der Umfang des zu erstellenden Systems – die <i>Functional Size</i> – abgeschätzt und in Function Points (fp) festgehalten. Den Ausgangspunkt der Schätzung bilden die Elementarprozesse. Dabei handelt es sich um kleine, in sich abgeschlossene Prozesse, die aus Sicht des Anwenders oder eines anderen Programms für die Arbeit mit der Anwendung sinnvoll sind.
Gantt-Chart	<i>Gantt-Chart</i> ist eine andere Bezeichnung für ein Balkendiagramm im Projektplan.
Geschlossene Frage	Eine geschlossene Frage ist eine Frage, die auf eine kurze Antwort – meist Ja oder Nein – abzielt.
Gewichtungsmethoden	Gewichtungsmethoden sind Verfahren zur Aufwandschätzung. Sie gehen nicht nur vom Umfang des zu erstellenden Systems aus, sondern berücksichtigen vor allem auch die Komplexität. Dazu werden verschiedene Faktoren über mathematische Formeln verknüpft.
Kalkulation	Eine Kalkulation ist – allgemein ausgedrückt – die Ermittlung von Kosten.
Klasse	In einer Klasse werden Objekte mit ähnlichen Eigenschaften und ähnlichem Verhalten zusammengefasst. Klassen stehen in einem hierarchischen Verhältnis zueinander. Eine Klasse bildet den Bauplan für ein Objekt.
Klassendiagramm	Im Klassendiagramm werden Klassen und ihre Beziehungen untereinander beschrieben. Die Klassen werden dabei durch ein Rechteck dargestellt. Die Beziehungen werden durch verschiedene Linien dargestellt.
Klassenhierarchie	Die Klassenhierarchie beschreibt die Abhängigkeiten der Klassen. Aus den übergeordneten Klassen werden die Eigenschaften der untergeordneten Klassen abgeleitet.
Kritischer Pfad	Der kritische Pfad in einem Projekt fasst alle kritischen Vorgänge zusammen.
Kritischer Vorgang	Ein kritischer Vorgang in einem Projekt ist ein Vorgang, von dem andere Vorgänge abhängen.
Lastenheft	Das Lastenheft beschreibt die Anforderungen an eine Software aus Sicht des Kunden.
Lines of Code	<i>Lines of Code</i> ist der englische Begriff für Quellcodezeilen.

LOC	LOC ist die Abkürzung für <i>Lines of Code</i> .
LOC-Methoden	Die LOC-Methoden sind Schätzmethoden zur Aufwandsermittlung. Hier wird die geschätzte Anzahl der Quellcodezeilen für das Programm durch die Anzahl Quellcodezeilen eines Programmierers in einer bestimmten Zeiteinheit – zum Beispiel einem Personentag – geteilt.
Machbarkeitsstudie	Die Machbarkeitsstudie fasst die Ergebnisse der Planungsphase zusammen. Sie besteht aus dem Lastenheft, einer groben Kalkulation und einem ersten Projektplan.
Mannjahr	Mannjahr ist eine andere Bezeichnung für Personenjahr.
Mannmonat	Mannmonat ist eine andere Bezeichnung für Personenmonat.
Manntag	Manntag ist eine andere Bezeichnung für Personentag.
Marktanalysen	Bei einer Marktanalyse werden Informationen über die Absatzmöglichkeiten eines Produkts gesammelt. Zur Marktanalyse gehören zum Beispiel Untersuchungen, <ul style="list-style-type: none"> • wie sich die Absatzmöglichkeiten entwickeln könnten, • welche Absatzmöglichkeiten vorhanden sind oder • welche Konkurrenten zu erwarten sind.
Meilenstein	Ein Meilenstein stellt einen wichtigen und überprüfbaren Zwischenschritt in einem Projekt oder einer Projektphase dar.
Netzplan	Der Netzplan stellt vor allem die Abhängigkeiten der einzelnen Vorgänge in einem Projektplan dar.
Normalfolge	Die Normalfolge beschreibt die Abhängigkeiten zwischen Vorgängen in einem Projekt. Ein Vorgang kann erst dann begonnen werden, wenn seine Vorgänger abgeschlossen sind.
Objekt	Ein Objekt in der objektorientierten Programmierung ist jede in sich geschlossene Einheit. Ein Objekt besteht aus dem Zustand und dem Verhalten.
Offene Frage	Eine offene Frage ist eine Frage, die auf eine ausführliche Antwort abzielt.
Optimale Entwicklungsdauer	Die optimale Entwicklungsdauer berücksichtigt, dass sich Entwicklungszeiten nicht beliebig durch den Einsatz weiterer Mitarbeiter verkürzen lassen. Für die Ermittlung der optimalen Entwicklungsdauer gibt es spezielle Formeln.

Personenjahr	Ein Personenjahr entspricht der Arbeitsleistung, die eine Person pro Jahr erbringen kann. Ein Personenjahr umfasst dabei 10 Personenmonate.
Personenmonat	Ein Personenmonat entspricht der Arbeitsleistung, die eine Person pro Monat erbringen kann. Ein Personenmonat umfasst ungefähr 22 Personentage.
Personentag	Ein Personentag entspricht der Arbeitsleistung, die eine Person pro Tag erbringen kann.
Projektmanagement	Das Projektmanagement umfasst Methoden zur Planung, Organisation und Steuerung von Projekten.
Projektplan	Ein Projektplan beschreibt die Phasen und Tätigkeiten in einem Projekt. Er stellt sowohl den zeitlichen Ablauf als auch die Abhängigkeiten dar.
Projektplanung	Die Projektplanung ist ein Teil des Projektmanagements. Sie kümmert sich vor allem um die zeitliche Planung des Projekts.
Prozentsatzmethode	Die Prozentsatzmethode ist eine Schätzmethode zur Aufwandsermittlung. Hier werden die prozentualen Anteile einer Entwicklungsphase aus ähnlichen Projekten herangezogen. Aus den prozentualen Anteilen kann der gesamte Aufwand hochgerechnet werden.
Rapid-Verfahren	Das Rapid-Verfahren ist ein vereinfachtes Verfahren der Function-Points-Methode. Es ordnet einer Eingabe 4 Function Points zu, einer Ausgabe 5 Function Points, einer Abfrage 4 Function Points, einem internen Datenbestand 7 Function Points und einem externen Datenbestand 5 Function Points.
Rolle	In der Software-Entwicklung definiert eine Rolle die Aufgaben, Verantwortlichkeiten und Kompetenzen einer Person.
Rollierende Korridorplanung	Die rollierende Korridorplanung ist eine besondere Form der Projektplanung. Zunächst werden die Phasen und die Ziele der Phasen festgelegt. Die Detailplanung für die einzelnen Phasen erfolgt erst dann, wenn die Phase begonnen wird, beziehungsweise kurz vor dem Beginn der Phase.
Rückwärtsplanung	Bei der Rückwärtsplanung wird ein Projekt ausgehend vom spätesten Endtermin geplant. Die Planung ergibt dann den spätesten Starttermin.
Stakeholder	<i>Stakeholder</i> sind im weitesten Sinne alle Beteiligten an der Entwicklung und dem Einsatz eines Produkts.

Statisches Modell (UML)	Im statischen Modell der UML wird vor allem die Struktur eines Systems dargestellt. Dazu werden unter anderem folgende Diagramme verwendet:
	<ul style="list-style-type: none"> • Klassendiagramm, • Komponentendiagramm, • Paketdiagramm und • Verteilungsdiagramm.
Stellenbeschreibung	Eine Stellenbeschreibung fasst die Aufgaben, Aktivitäten und Verantwortlichkeiten für einen Arbeitsplatz in einem Unternehmen zusammen.
Storyboard	<p><i>Storyboard</i> ist der englische Begriff für Drehbuch. Über ein <i>Storyboard</i> können zum Beispiel Prozesse in einfacher grafischer Form skizziert werden.</p>
Teilnehmende Beobachtung	Bei der teilnehmenden Beobachtung nimmt ein Dritter als Zuschauer an einem Arbeitsschritt teil. Er wirkt nicht aktiv mit, sondern versucht, den Arbeitsschritt möglichst exakt festzuhalten – zum Beispiel durch Notizen.
Teufelsquadrat	Das Teufelsquadrat stellt die vier wesentlichen Faktoren der Software-Entwicklung Qualität, Quantität, Dauer und Kosten sowie ihre Abhängigkeiten in grafischer Form dar.
UML	<p>UML steht für <i>Unified Modeling Language</i>. UML ist eine objektorientierte Standardmodellierungssprache zur Beschreibung der Struktur und des Verhaltens von Objekten in Anwendungsbereichen und Datenverarbeitungssystemen.</p>
Unified Modeling Language	Siehe UML.
Vererbung	Über die Vererbung können Sie Attribute und Methoden aus übergeordneten Klassen an eine untergeordnete Klasse weitergeben. Die ererbten Methoden und Attribute können verändert und erweitert werden.
Verteilungsdiagramm	Das Verteilungsdiagramm wird für die Darstellung der Verteilung eines Systems auf unterschiedlichen Rechnern eingesetzt.
Vorgang	Vorgang ist eine andere Bezeichnung für eine Aktivität oder ein Arbeitspaket in einem Projekt.
Vorwärtsplanung	Bei der Vorwärtsplanung wird ein Projekt ausgehend vom frühesten Starttermin geplant. Die Planung ergibt dann den frühesten Endtermin.

C. Literaturverzeichnis

Verwendete Literatur

- Balzert, H. (2004). *Lehrbuch Grundlagen der Informatik und Konzepte, Notationen in UML 2, Java 5, C# und C++, Algorithmik und Software-Technik. Anwendungen.* 2. Aufl., Heidelberg: Spektrum Akademischer Verlag.
- Balzert, H. (2008). *Lehrbuch der Softwaretechnik. Softwaremanagement.* 2. Aufl., Heidelberg: Spektrum Akademischer Verlag.
- Balzert, H. (2009). *Lehrbuch der Softwaretechnik. Basiskonzepte und Requirements Engineering.* 3. Aufl., Heidelberg: Spektrum Akademischer Verlag.
- Meier, R. (2009). *Projektmanagement. Grundlagen, Methoden, Techniken.* Offenbach: Gabal.
- Sneed, H. M. (1999). *Software-Management.* Köln: Rudolf Müller.
- Sommerville, I. (2012). *Software Engineering.* 9. Aufl., Hallbergmoos: Pearson Studium.

Empfohlene Literatur

- DeMarco, T. (2007). *Der Termin. Ein Roman über Projektmanagement.* München: Hanser.
- Krypczyk, V. & Bochkor, O. (2018). *Handbuch für Softwareentwickler: Das Standardwerk zu professionellem Software Engineering.* Bonn: Rheinwerk Computing.
- Litke, H.-D., Kunow, I. & Schulz-Wimmer, H. (2018). *Projektmanagement. Best of.* 4. Aufl., Freiburg: Haufe-Lexware.
- Oestereich, B. & Scheithauer, A. (2013). *Analyse und Design mit der UML 2.5. Objekt-orientierte Softwareentwicklung.* 11. Aufl., Berlin: Oldenbourg.

Das Buch von DeMarco beschreibt in Romanform die Entwicklung einer neuen Software und eignet sich auch als Abendlektüre. Zwar sind nicht mehr alle vorgestellten Techniken brandaktuell, das Buch ist aber sehr unterhaltsam geschrieben.

D. Abbildungsverzeichnis

Abb. 1.1	Von der Idee zur Machbarkeitsstudie	4
Abb. 2.1	Musterfragebogen	9
Abb. 2.2	Das Storyboard für den Prozess „Vermietung“	11
Abb. 3.1	Abhängigkeiten bei der Software-Entwicklung	18
Abb. 3.2	Veränderung nach dem Erhöhen der Quantität (das grüne Quadrat außen)	18
Abb. 3.3	Das „Teufelsquadrat“ von Sneed	19
Abb. 3.4	Auswirkung der verringerten Kosten (hier hat sich die Qualität verringert)	19
Abb. 3.5	Auswirkungen der erhöhten Qualität (hier hat sich die Quantität verringert)	20
Abb. 3.6	Steigende Kosten bei verkürzter Entwicklungsdauer	20
Abb. 4.1	Zuordnung der Function Points zu den Personenmonaten durch eine Kurve	28
Abb. 4.2	Die Kurve zur IBM-Tabelle	29
Abb. 5.1	Meilensteine und Aktivitäten	34
Abb. 5.2	Der Vorgänger und der Nachfolger für das Auswerten der Informationen	36
Abb. 5.3	Normalfolge (links), Anfangsfolge (Mitte) und Endfolge (rechts)	37
Abb. 5.4	Schematisiertes Balkendiagramm	38
Abb. 5.5	Schematisierter Netzplan	38
Abb. 5.6	Die Abhängigkeiten der Planungsphase	40
Abb. 5.7	Darstellung in einer Projektmanagement-Software (hier: Microsoft Project)	41
Abb. 5.8	Der Netzplan für Autovermietung 2030	42
Abb. 7.1	Das Klassendiagramm des Geldautomaten	47
Abb. 7.2	Das Anwendungsfalldiagramm des Geldautomaten	48
Abb. 7.3	Das Aktivitätsdiagramm des Geldautomaten	49
Abb. 7.4	Erstellen von Klassendiagrammen mit Microsoft Visio	50
Abb. G.1	Stark vereinfachtes Klassendiagramm	68

E. Tabellenverzeichnis

Tab. 2.1	Tabelle zum Festlegen der Qualitätsanforderungen.....	8
Tab. 4.1	Punktwerte	25
Tab. 4.2	Die Function Points für die Eingabe	26
Tab. 4.3	Die Function Points für die Autovermietung	26
Tab. 4.4	Zuordnung der Function Points zur Dauer	28
Tab. 4.5	Zuordnung der Function Points zur Dauer nach IBM	29
Tab. 5.1	Die Vorgänge.....	39
Tab. 5.2	Die Abhängigkeiten	39
Tab. 5.3	Die Termine	40

F. Sachwortverzeichnis

A

Aggregation	48
Akteur	48
Aktivität	34
Aktivitätsdiagramm	49
Analogiemethode	20
Anfangsfolge	37
Anforderung	
funktionale	7
Anwendungsfalldiagramm	48
Arbeitspaket	35
Aufwand	17
Aufwands- und Kostenschätzung	17

B

Balkendiagramm	38
Beobachtung	
teilnehmende	12
Brainstorming	12

D

drei Amigos	45
Durchführbarkeitsstudie	3

E

Endfolge	37
Entwicklungsduer	
Ermittlung der	27
optimale	31

F

Frage	
geschlossene	10
offene	10
Fragebogen	9

G

Gantt-Chart	38
Gewichtungsmethode	22

I

IBM-Tabelle	29
Istanalyse	9

K

Kalkulation	3
Kapazität	37
Klassendiagramm	47
Kontext und Überblick	7
Korridorplanung	
rollierende	33

L

Lastenheft	3
Aufbau eines	7
Informationssammlung	8
LOC-Variante	21

M

Machbarkeitsstudie	3
Marktanalyse	9
Meilenstein	34
Modell	
dynamisches	46
statisches	46

N

Nachfolger	36
Netzplan	38
Normalfolge	37

P

Personenjahr	21
Personenmonat	21
Personentag	21
Pfad	
kritischer	37
Projektmanagement	33
Projektmanagement-Software	39
Projektplan	3, 33
Projektplanung	33
Prozentsatzmethode	21
Pufferzeit	37

Q

Qualität	17
Qualitätsanforderung	7
Quantität	17

R

Rahmenbedingung	7
Rapid-Verfahren	25
Rückwärtsplanung	37

S

Schätzmethode	20
Stakeholder	12
Stellenbeschreibung	10
Stellenprofil	10

T

Termin- und Kapazitätsplanung	34
Terminplanung	35
Teufelsquadrat	18

U

UML	45
Diagramme der	46
Geschichte der	45
UML-Werkzeug	49
Unified Modeling Language	45

V

Visionen und Ziele	7
Vorgang	35
kritischer	37
Vorgänger	36
Vorwärtsplanung	37

G. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP19D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

- Ein Sportverein möchte seine Mitglieder über ein Computerprogramm verwalten. Entwickeln Sie ein entsprechendes Lastenheft.

Das System soll mindestens folgende Funktionen abdecken:

- Verwaltung der Adressdaten der Mitglieder,
- Verwaltung der Sportarten,
- Verwaltung der Übungsleiter und Trainer,
- Eintritt eines Mitglieds,
- Austritt eines Mitglieds und
- Erstellen der Beitragsrechnung.

Ein Mitglied kann eine oder mehrere Sportarten betreiben.

Fehlende Informationen können Sie selbst ergänzen.

Denken Sie bitte daran, dass das Lastenheft die Software aus Sicht des Auftraggebers – also des Vereins – darstellen soll.

50 Pkt.

- Einem Auftraggeber sind die Entwicklungskosten einer Software zu hoch. Er will die Kosten reduzieren, ohne die Qualität oder die Quantität zu verringern. Welche Auswirkungen hat dieser Wunsch?

Skizzieren Sie die Auswirkung zusätzlich am Teufelsquadrat.

10 Pkt.

- Sie unterhalten sich mit einem anderen Software-Entwickler über die Probleme bei Aufwandsschätzungen. Ihr Kollege sagt Ihnen, dass er sich um die Entwicklungszeit eigentlich keine großen Gedanken mache. Man könne ja jederzeit weitere Entwickler hinzuziehen und so die Zeit nahezu beliebig verkürzen.

Stimmt diese Aussage? Begründen Sie bitte Ihre Antwort.

10 Pkt.

4. Die Entwicklungszeit einer Software wird auf 1 300 Personentage geschätzt. Rechnen Sie diese Angabe in Personenjahre um. Geben Sie dabei bitte auch den Rechenweg an.

10 Pkt.

5. Betrachten Sie bitte das folgende Klassendiagramm:

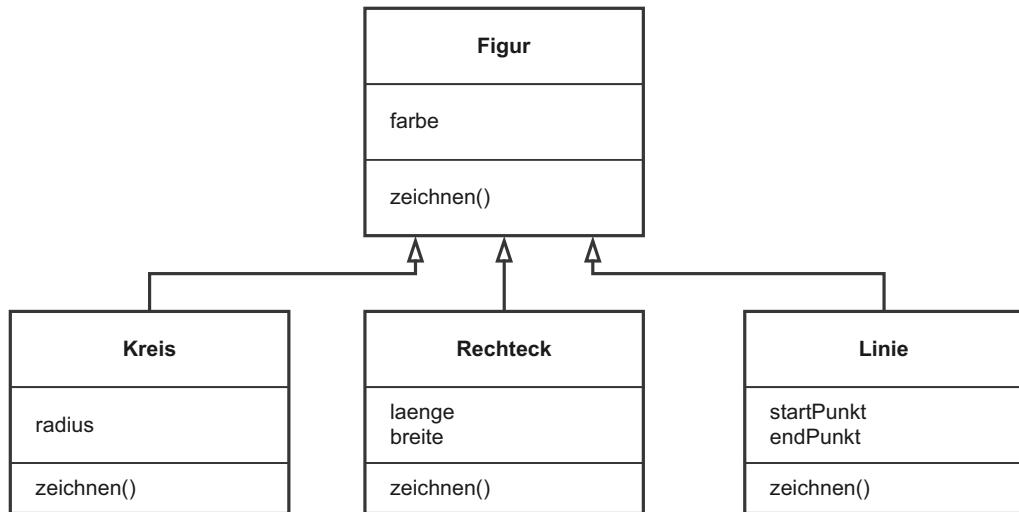


Abb. G.1: Stark vereinfachtes Klassendiagramm

Geben Sie bitte **sämtliche** Attribute und Methoden der Klasse **Linie** an.

Welche Beziehung besteht zwischen den Klassen **Linie** und **Figur**?

20 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt:
Objektorientierte Analyse und Entwurf

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1019N01

CSHP20D

Objektorientierte Software-Entwicklung mit C#

**Beispielprojekt:
Objektorientierte Analyse und Entwurf**

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Objektorientierte Analyse und Entwurf

Inhaltsverzeichnis

Einleitung	1
1 Aufgaben und Ergebnisse der Analyse und des Entwurfs	3
Zusammenfassung	6
2 Das erste, grobe dynamische Modell	7
2.1 Ermitteln der wesentlichen Produktfunktionen	7
2.2 Ermittlung der Systemgrenzen	8
2.3 Das erste Anwendungsfalldiagramm	10
2.4 Verfeinerung der Anwendungsfalldiagramme	13
2.5 Beschreibung der Anwendungsfälle	15
2.6 Kategorisierung der Anwendungsfälle	16
Zusammenfassung	17
3 Das erste, grobe statische Modell	18
3.1 Objekt- und Klassenkandidaten ermitteln	20
3.2 Beziehungen ermitteln	21
3.3 Attribute ermitteln	22
3.4 Klassendiagramm erstellen	23
Zusammenfassung	26
4 Die Verfeinerung des dynamischen Modells	28
4.1 Die Elemente in einem Aktivitätsdiagramm	28
4.2 Beschreibung der Aktivitäten	31
Zusammenfassung	32
5 Die Verfeinerung des statischen Modells	33
5.1 Attributtypen festlegen	33
5.2 Ermitteln der Methoden	37
5.3 Die Methoden im Klassendiagramm	39
Zusammenfassung	41

6 Verfeinerung des Klassendiagramms	42
6.1 Präzisierung der Methoden	42
6.2 Ergänzen von Methoden	44
6.3 Festlegen der Sichtbarkeiten	45
6.4 Präzisierung der Klassenbeziehungen	48
6.5 Exkurs: Entwurfsmuster	51
Zusammenfassung	52
7 Die Datenhaltung	54
7.1 Was ist eine Datenbank?	54
7.2 Das relationale Datenmodell	55
7.3 Die Tabellen für die Autovermietung	57
Zusammenfassung	59
8 Agile Softwareentwicklung	60
Zusammenfassung	62
Schlussbetrachtung	63
Anhang	
A. Lösungen der Aufgaben zur Selbstüberprüfung	64
B. Glossar	67
C. Literaturverzeichnis	71
D. Abbildungsverzeichnis	72
E. Tabellenverzeichnis	74
F. Sachwortverzeichnis	75
G. Einsendeaufgabe	77

Einleitung

In diesem Studienheft erstellen wir das Analyse- und Entwurfsmodell für die Software unserer Autovermietung.

Diese Modelle beschreiben, was die Software abbilden soll und wie sie technisch umgesetzt werden soll. Anders als das recht formlose Lastenheft muss das Analysemodell dabei den Problembereich und die Anforderungen der Anwender abstrahieren und auch formalisieren. So werden zum Beispiel die Produktfunktionen über Anwendungsfalldiagramme beschrieben und die Produktdaten über Klassendiagramme. Beim Entwurfsmodell dagegen steht die Sicht des Entwicklers im Vordergrund.

Häufig ist statt von „Entwurf“ auch von „Design“ der Software die Rede. Gemeint ist aber dasselbe.



Im Einzelnen lernen Sie in diesem Studienheft:

- welche Aufgaben die Analyse übernimmt,
- welche Ergebnisse die Analyse liefert,
- wie Sie ein erstes, grobes dynamisches Modell erstellen,
- wie Produktfunktionen mit Anwendungsfalldiagrammen modelliert werden,
- wie Anwendungsfälle beschrieben werden können,
- wie Sie ein erstes, grobes statisches Modell erstellen,
- wie Sie Klassen und Attribute ermitteln,
- wie Produktdaten mit Klassendiagrammen modelliert werden,
- wie das dynamische Modell mit Aktivitätsdiagrammen verfeinert wird,
- wie das statische Modell durch eine Beschreibung der Attribute verfeinert wird,
- wie Sie die Methoden für die Klassen ermitteln,
- welche Aufgaben der Entwurf übernimmt,
- welche Ergebnisse der Entwurf liefert,
- wie das Klassendiagramm aus der Analyse im Entwurf verfeinert wird,
- wie die Klassenbeziehungen präzisiert werden,
- wie Sie Entwurfsmuster einsetzen können,
- was eine Datenbank ist,
- wie die Datenspeicherung in unserem Beispielprojekt Autovermietung 2030 über ein relationales Modell abgebildet wird,
- welche Vorteile agile Entwicklungsmethoden haben und
- welche Besonderheiten agile Methoden wie Extreme Programming und Scrum aufweisen.



Bitte beachten Sie:

Die UML-Diagramme in diesem Studienheft wurden mit dem Programm Visio von Microsoft erstellt. Beim Einsatz eines anderen Programms können die UML-Diagramme etwas anders aussehen.

Christoph Siebeck

1 Aufgaben und Ergebnisse der Analyse und des Entwurfs

Bevor wir mit dem Erstellen der verschiedenen Modell beginnen, wollen wir uns zunächst einmal kurz die Aufgaben und Ergebnisse der objektorientierten Analyse und des objektorientierten Entwurfs ansehen.

Wie Sie ja bereits wissen, geht es bei der Modellierung eines Softwaresystems grundsätzlich um die Fragen:

- „Was soll die Software aus der Realität abbilden?“ und
- „Wie soll sie es abbilden?“

Zum „Was“ gehören dabei nicht nur Prozesse, sondern zum Beispiel auch das Umfeld – also Personen, Gegenstände wie Bücher, CDs und so weiter.

Die Antwort auf die Frage „Was?“ liefert das **Analysemodell**. Es ist Voraussetzung für die Antwort auf die Frage „Wie?“ – also für das Erstellen des **Entwurfsmodells**.

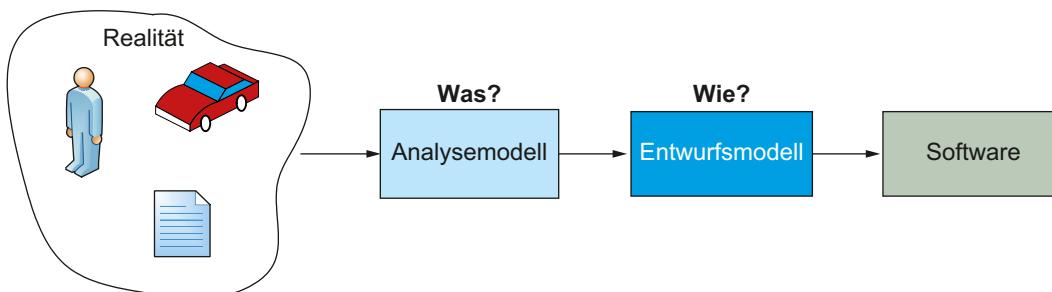


Abb. 1.1: Analyse- und Entwurfsmodell

Anders als das Lastenheft, das die Anforderungen an die Software vor allem aus Kundensicht beschreibt, geht es bei der Analyse darum, eine fachliche Lösung für die Anforderungen zu modellieren – also zu beschreiben, was bei der Abbildung der Realität für die Software zu beachten ist. Dabei müssen ebenfalls der Kunde beziehungsweise die späteren Anwender mit einbezogen werden. Sie müssen prüfen, ob die fachliche Lösung auch tatsächlich ihren Anforderungen entspricht.

Da zur Antwort auf die Frage „Was?“ neben den Prozessen auch das Umfeld des Systems gehört, muss die Analyse sowohl ein dynamisches Modell als auch ein statisches Modell enthalten.

Zur Auffrischung:

Ein statisches Modell beschreibt vor allem die Struktur eines Systems. Ein dynamisches Modell beschreibt vor allem das Verhalten eines Systems.



Welches Modell dabei den Schwerpunkt bildet, hängt von der Anwendung ab, die erstellt werden soll. Bei Datenbankanwendungen ist zum Beispiel das statische Modell besonders wichtig, bei sehr stark interaktiven Anwendungen – wie zum Beispiel einer Textverarbeitung oder einem Grafikprogramm – steht das dynamische Modell im Mittelpunkt.



Eine Interaktion ist – allgemein ausgedrückt – eine Wechselbeziehung. Bei einem Computerprogramm sind damit vor allem die Aktionen und die Einflussnahme des Anwenders gemeint.

Beide Modelle müssen mehr oder weniger formal beschrieben werden. Zum einen wird so die Umsetzung im Entwurfsmodell vereinfacht, zum anderen lassen sich Missverständnisse bei der Kommunikation mit dem Kunden beziehungsweise mit dem Anwender vermeiden. In unserem Fall benutzen wir für die formale Beschreibung verschiedene UML-Diagramme und erweitern sie um einige umgangssprachliche Darstellungen sowie zusätzliche Grafiken.

Hinweis:

UML-Diagramme allein reichen in der Regel für die Kommunikation mit dem Kunden beziehungsweise den Anwendern nicht aus. Sie können bei einem Laien keine UML-Kenntnisse voraussetzen.

In welcher Reihenfolge Sie die beiden Modelle erstellen, hängt wieder von der Art der Anwendung ab. In der Praxis haben sich aber folgende Schritte bewährt, die sowohl dynamische als auch statische Aspekte gleichermaßen berücksichtigen:

1. Es wird ein erstes, grobes dynamisches Modell erstellt. Dazu werden typische Anwendungsfälle identifiziert und mit Anwendungsfalldiagrammen skizziert.
2. Es wird ein erstes, grobes statisches Modell als Objekt- und Klassendiagramm erstellt. Dieses Modell ist gleichzeitig das grobe **konzeptionelle Datenmodell**. Was sich hinter diesem Begriff genau verbirgt, erfahren Sie im weiteren Verlauf des Studienhefts.
3. Die Anwendungsfälle aus dem ersten Schritt werden mit Aktivitätsdiagrammen verfeinert.
4. Das grobe konzeptionelle Datenmodell aus dem zweiten Schritt wird überarbeitet und verfeinert.

Als Ergebnis liefert die Analyse ein **Pflichtenheft**.

Dieses Pflichtenheft findet sich genau wie das Lastenheft nicht nur bei der Software-Entwicklung, sondern auch bei vielen anderen Projekten. Allgemein ausgedrückt beschreibt es die vom Auftragnehmer erarbeiteten Realisierungsvorgaben für die Umsetzung des Lastenhefts.¹



Der Unterschied zwischen Lastenheft und Pflichtenheft

Das Lastenheft beschreibt die Forderungen des Auftraggebers. Das Pflichtenheft beschreibt, wie der Auftragnehmer diese Forderungen umsetzen will.

1. Quelle: DIN 69905.

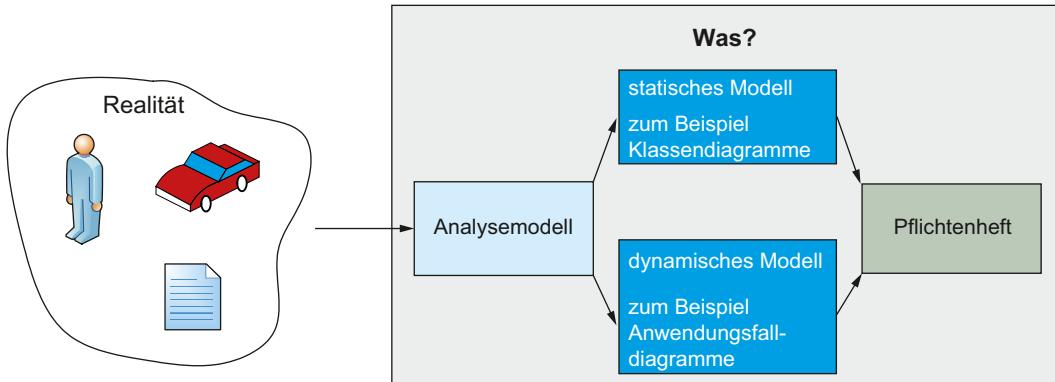


Abb. 1.2: Von der Realität zum Pflichtenheft

So viel zum Analysemodell.

Das Entwurfsmodell liefert die Antwort auf die Frage „Wie?“. Es bildet ein System aus Sicht des Entwicklers ab – also unter technischen Gesichtspunkten.

Zu diesen technischen Gesichtspunkten gehören zum Beispiel folgende Fragen:

- Wie sollen die Klassen mit ihren Attributen und Methoden konkret umgesetzt werden?
- Welche zusätzlichen Methoden werden für die Umsetzung benötigt?
- Kann das System in kleinere Teilsysteme zerlegt werden? Wenn ja: in welche?
- Soll das System auf mehreren Rechnern verteilt arbeiten? Wenn ja: Wie sieht diese Verteilung aus?
- Wie werden die Daten der Anwendung gespeichert?

Zusätzlich müssen auch folgende Aspekte berücksichtigt werden:

- Müssen die Daten des Systems vor unbefugten Zugriffen geschützt werden? Muss sich zum Beispiel ein Anwender erst anmelden, bevor er mit dem System arbeiten kann?
- Sind spezielle Maßnahmen zum Schutz gegen Datenverlust erforderlich? Müssen zum Beispiel regelmäßig Sicherungskopien automatisch vom System erzeugt werden?
- Sind besondere Anforderungen an die Ausführungsgeschwindigkeit zu beachten? Müssen bestimmte Antwortzeiten zwingend eingehalten werden?
- Muss das System besonders ausfallsicher sein? Müssen Teile des Systems redundant aufgebaut werden?

Die **Antwortzeit** beschreibt die Zeitspanne, die das System braucht, um eine Rückmeldung zu einer Aktion zu geben.



Redundant bedeutet eigentlich so viel wie überflüssig oder überreichlich. In der Informationstechnik bezeichnet ein **redundantes System** ein System, in dem alle wichtigen Komponenten mehrfach in identischer Form vorhanden sind. Fällt eine Komponente aus, springt eine andere identische Komponente ein.

Im Ergebnis liefert das Entwurfsmodell einen detaillierten Bauplan für die Anwendung – die **Software-Architektur**.

Ausgangspunkt für das Entwurfsmodell sind dabei die Ergebnisse des Analysemodells – also das Pflichtenheft. Aus diesem Pflichtenheft werden zum Beispiel die Klassendiagramme übernommen und um technische Details erweitert.

So viel zu den Aufgaben und Ergebnissen der Analyse und des Entwurfs. In den nächsten Kapiteln werden wir die einzelnen Schritte für die Software unserer Autovermietung durchführen.

Zusammenfassung

Die Analyse modelliert eine fachliche Lösung für die Anforderungen an die Software. Sie umfasst sowohl ein statisches als auch ein dynamisches Modell.

Die Ergebnisse der Analyse werden im Pflichtenheft dargestellt.

Das Entwurfsmodell bildet ein System aus Sicht des Entwicklers ab.

Es liefert einen detaillierten Bauplan für die Software – die Software-Architektur.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Warum muss in der Analyse in der Regel sowohl ein dynamisches als auch ein statisches Modell erstellt werden?

- 1.2 Was unterscheidet das Lastenheft und das Pflichtenheft?

- 1.3 Was bildet das Entwurfsmodell ab?

2 Das erste, grobe dynamische Modell

In diesem Kapitel erstellen wir das erste, grobe dynamische Modell für unser Softwareprodukt Autovermietung 2030. Dieses grobe Modell umfasst die wesentlichen Produktfunktionen – also das, was die Software leisten soll.

Für das Erstellen des Modells sind folgende einzelne Schritte erforderlich:

1. Aus dem Lastenheft werden die wesentlichen Produktfunktionen ermittelt – also die Funktionen, die tatsächlich umgesetzt werden können beziehungsweise umgesetzt werden sollen.
2. Die Systemgrenzen werden festgelegt. Sie bilden die Stellen, an denen die Software mit der Außenwelt kommunizieren muss.
3. Die wesentlichen Produktfunktionen und die Systemgrenze werden als Anwendungsfalldiagramme skizziert.
4. Die Anwendungsfälle werden detailliert beschrieben.
5. Die Anwendungsfälle werden anhand ihrer Wichtigkeit kategorisiert. Damit wird auch eine Reihenfolge für die technische Umsetzung vorgegeben.

Schauen wir uns diese Schritte jetzt im Detail an. Beginnen wir mit dem Ermitteln der wesentlichen Produktfunktionen.

2.1 Ermitteln der wesentlichen Produktfunktionen

Das Ermitteln der wesentlichen Produktfunktionen ist in unserem Beispiel relativ einfach. Wir nehmen die Beschreibungen der verschiedenen funktionalen Anforderungen aus dem Lastenheft und überprüfen, welche dieser Funktionen tatsächlich umgesetzt werden können beziehungsweise umgesetzt werden sollen.

Schauen wir uns die einzelnen funktionalen Anforderungen aus dem Lastenheft noch einmal an. Die Beschreibungen lassen wir zur besseren Übersicht weg.

- LF10 Pflege der Kundendaten
- LF20 Pflege der Fahrzeugdaten
- LF30 Vermietung
- LF40 Überführung eines gemieteten Fahrzeugs
- LF50 Rückgabe
- LF60 Rückführung der zurückgegebenen Fahrzeuge
- LF70 Reservierung
- LF80 Listen

Hinweis:

Wenn Sie nicht mehr genau wissen, was sich im Detail hinter einer bestimmten funktionalen Anforderung verbirgt, sehen Sie bitte noch einmal im Lastenheft unseres Produkts nach.

Aus der Liste können wir jetzt im ersten Schritt die funktionalen Anforderungen LF40 und LF60 streichen. Hier werden ja die Fahrzeuge zur Mietstation und wieder zurück transportiert. Diese Vorgänge sind für unsere Software nicht von Bedeutung.

Wesentlich ist in unserem Beispiel nur, dass festgehalten werden kann, dass die Fahrzeuge an den Kunden übergeben wurden beziehungsweise vom Kunden zurückgegeben wurden. Diese Informationen erhalten wir in der funktionalen Anforderungen LF30 und LF50.

Die funktionalen Anforderungen LF10, LF20, LF30 und LF50 müssen in jedem Fall umgesetzt werden. Es handelt sich hier ja um Basisfunktionen, ohne die eine Autovermietung nicht arbeiten könnte.

Auf die funktionalen Anforderungen LF70 und LF80 könnten wir grundsätzlich verzichten, da es sich ja mehr um Komfortfunktionen handelt. Allerdings ist mit dem Auftraggeber vereinbart worden, dass diese Funktionen ebenfalls in der ersten Version der Software umgesetzt werden sollen. Also werden sie nicht gestrichen.

Wie Sie gerade gesehen haben, geht das Ermitteln der wesentlichen Produktfunktionen über das Lastenheft sehr einfach und auch sehr schnell. Allerdings birgt es auch eine große Gefahr: Wenn nämlich im Lastenheft eine wesentliche Produktfunktion fehlt, fällt sie bei dieser Technik zwangsläufig unter den Tisch. Überprüfen Sie daher in diesem Schritt noch einmal sorgfältig, ob Sie wirklich alle wesentlichen Produktfunktionen festgehalten haben.

2.2 Ermittlung der Systemgrenzen

Kommen wir nun zum Ermitteln der Systemgrenzen. Wie Sie ja bereits wissen, sind die Systemgrenzen die Stellen, an denen das Software-System mit der Umwelt kommunizieren muss.

In unserem Beispiel können wir bei der Ermittlung der Systemgrenzen ebenfalls auf das Lastenheft zurückgreifen – und zwar auf den Kontext und den Überblick und die Rahmenbedingungen.

Wir müssen hier vor allem die Kommunikation mit den verschiedenen Anwendern abbilden. Dabei lassen sich grob die Mitarbeiter der Autovermietung und die eigentlichen Kunden unterscheiden. Die Systemgrenzen können dann grafisch so dargestellt werden:

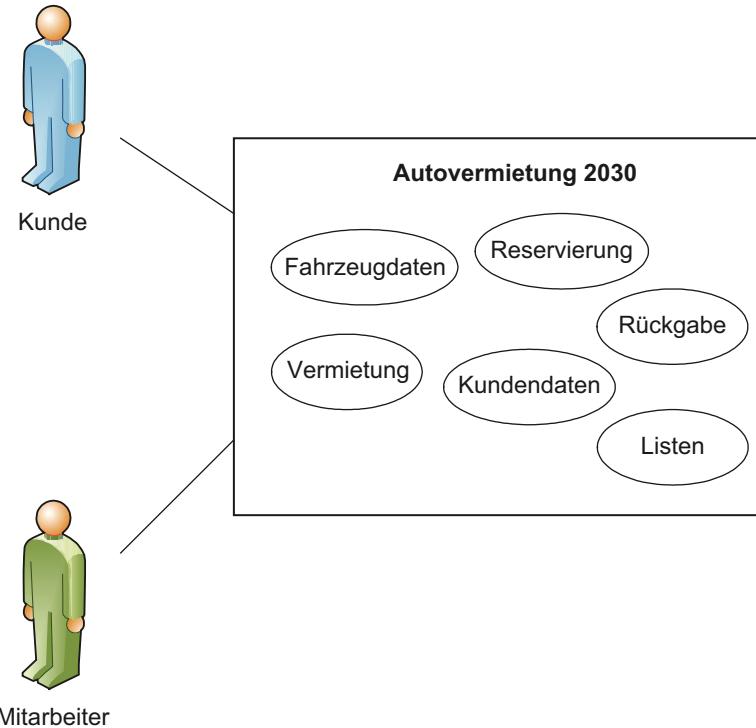


Abb. 2.1: Die Systemgrenzen

Auf der linken Seite werden die beiden wesentlichen Anwendergruppen dargestellt. Rechts befindet sich dann das System mit den wesentlichen Produktfunktionen. Die Produktfunktionen werden dabei durch die Ellipsen abgebildet.

Anhand der Systemgrenzen lässt sich bereits jetzt ein erster grober Entwurf der Benutzeroberfläche für das Programm Autovermietung 2030 erstellen. So sollen ja zum Beispiel die Kunden nicht direkt auf Produktfunktionen wie das Anlegen von Fahrzeugdaten oder die Reservierung zugreifen können, sondern im Wesentlichen nur Listen erzeugen können. Wir benötigen also in jedem Fall zwei unterschiedliche Oberflächen: eine für die Mitarbeiter der Autovermietung und eine für die Kunden.

Ein erster sehr grober Entwurf der Oberflächen könnte zum Beispiel so aussehen:

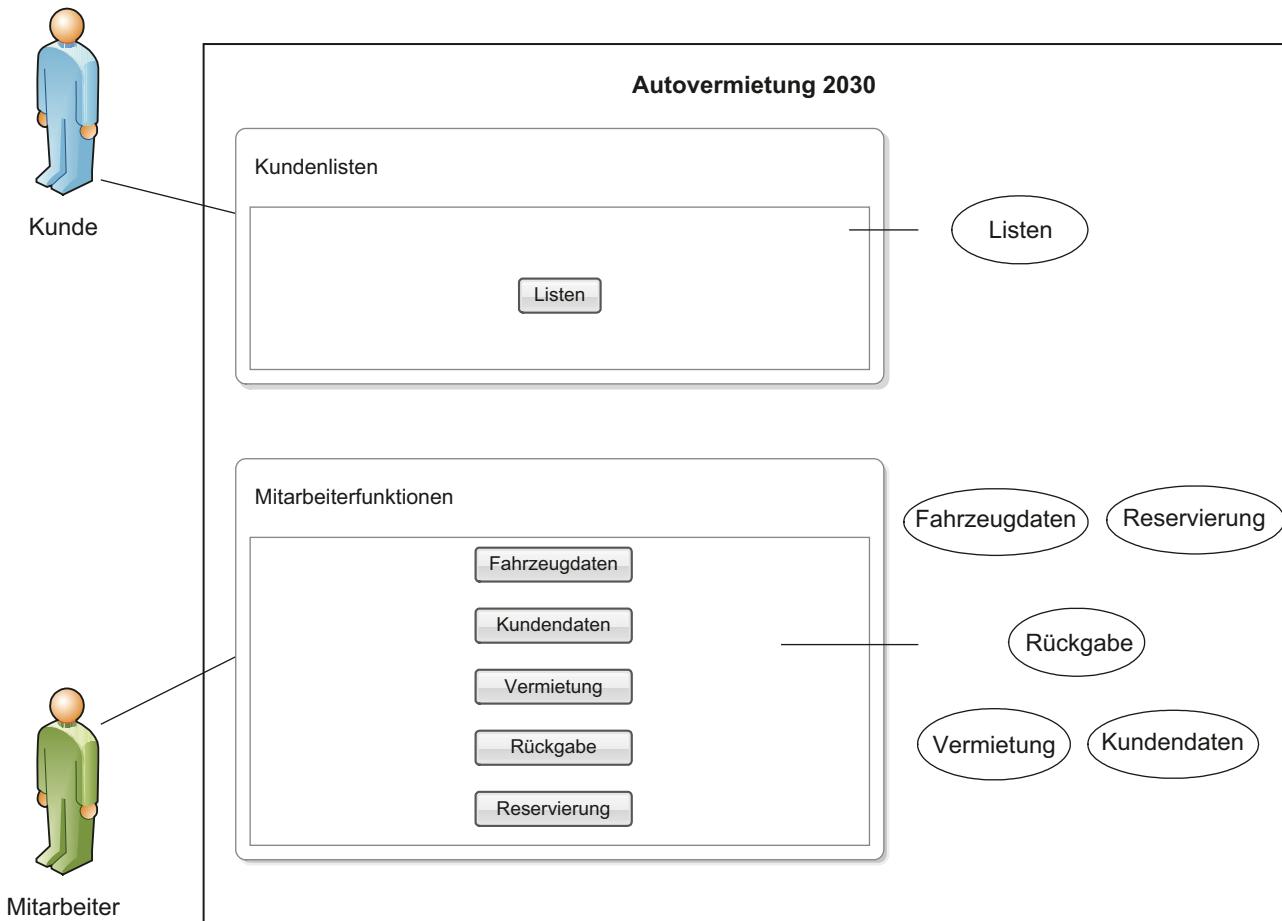


Abb. 2.2: Ein erster Entwurf der Oberflächen



Bitte beachten Sie:

In dieser Phase handelt es sich bei der Oberfläche lediglich um einen sehr groben unvollständigen Prototyp, der nur einen sehr kleinen Ausschnitt darstellt. Die Oberfläche soll nur skizziert werden, aber noch keinerlei Funktion haben.

2.3 Das erste Anwendungsfalldiagramm

Jetzt können wir endlich das erste echte UML-Diagramm für unsere Software erstellen: ein **Anwendungsfalldiagramm** – auch **Use-Case-Diagramm** genannt.

Diese Diagrammform haben Sie ja bereits kurz kennengelernt.

Wiederholen wir noch einmal:

Das Anwendungsfalldiagramm stellt Beziehungen zwischen Akteuren und Prozessen – den Anwendungsfällen – dar. Als Akteure gelten dabei sowohl die Personen, die aktiv mit dem System arbeiten, als auch andere Systeme, die mit dem System zusammenarbeiten.

Die Akteure werden als „Strichmännchen“ abgebildet und durch einen kurzen Text unter dem „Strichmännchen“ beschrieben.

Die Anwendungsfälle werden als Ellipsen dargestellt. Die Beschreibung erfolgt durch einen kurzen Text in der Ellipse – zum Beispiel in der Form Substantiv + Verb.

Die Verbindung zwischen den Akteuren und den Anwendungsfällen wird durch Linien hergestellt.

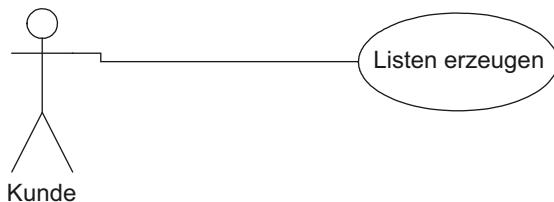


Abb. 2.3: Ein Akteur (links) und ein Anwendungsfall (rechts)

Bei der Verbindung zwischen den Akteuren und den Anwendungsfällen werden nur die Akteure zugeordnet, die tatsächlich am System mit dem Anwendungsfall arbeiten – also die Akteure, die später im Programm auch die entsprechende Funktion ausführen.

Neben den Akteuren und den Anwendungsfällen werden im Anwendungsfalldiagramm auch die Systemgrenzen dargestellt. Dazu wird ein Rechteck benutzt.

**Bitte beachten Sie:**

Die Akteure befinden sich immer außerhalb der Systemgrenzen und die Anwendungsfälle immer innerhalb der Systemgrenzen.

Das erste Anwendungsfalldiagramm für unser Programm Autovermietung 2030 könnte dann so aussehen:

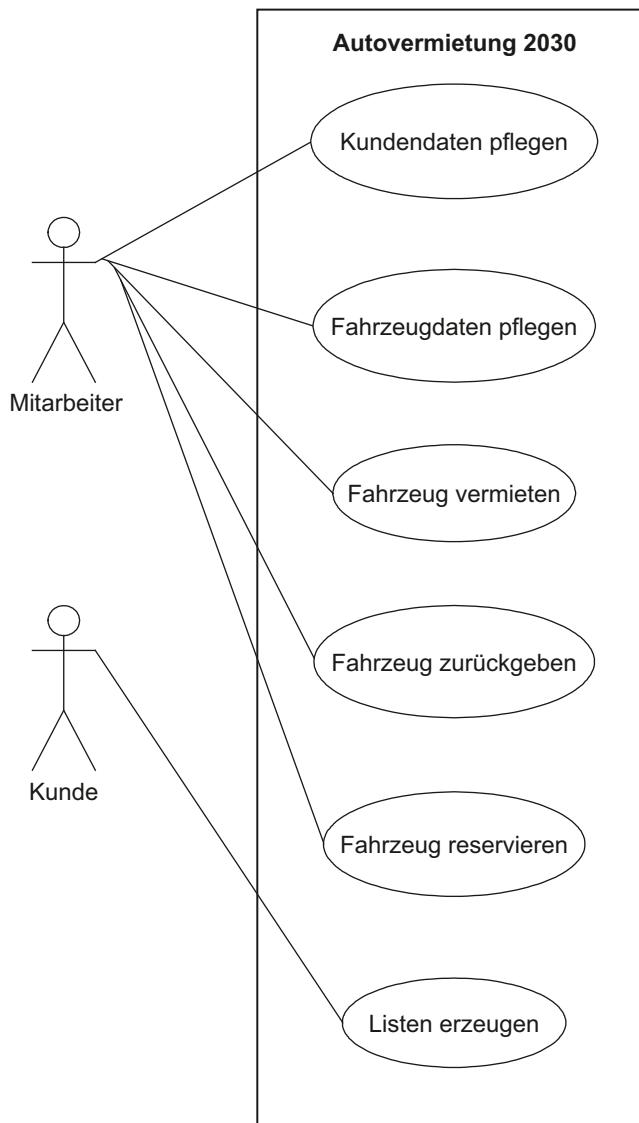


Abb. 2.4: Das erste Anwendungsfalldiagramm für Autovermietung 2030

Als Akteure haben wir lediglich Mitarbeiter und Kunden angegeben. Auf eine detailliertere Darstellung der verschiedenen Mitarbeiter haben wir bewusst verzichtet. In unserem Programm sollen sämtliche Verwaltungsfunktionen zunächst einmal auch sämtlichen Mitarbeitern zur Verfügung stehen. Aus Sicht des Systems spielt es daher keine Rolle, ob es sich nun um einen Mitarbeiter aus der Verwaltung oder um einen Mitarbeiter aus der Vermietung handelt.

Auch die Aushilfen haben wir nicht als eigenen Akteur abgebildet. Hier ist ja ebenfalls egal, ob ein Anwendungsfall nun von einem Mitarbeiter oder von einer Aushilfe ausgeführt wird – die Funktion ist immer identisch.

Hinweis:

Wenn unterschiedliche Rollen der Anwender berücksichtigt werden müssen, benötigen Sie auch für jede Rolle einen eigenen Akteur. Wenn unser Programm zum Beispiel die Anwendungsfälle „Kundendaten pflegen“ und „Fahrzeugdaten pflegen“ nur für Mitarbeiter in der Verwaltung anbieten soll, müssen Sie auch bei den Akteuren zwischen Mitarbeitern in der Verwaltung und sonstigen Mitarbeitern unterscheiden.

2.4 Verfeinerung der Anwendungsfalldiagramme

Das erste grobe Anwendungsfalldiagramm kann jetzt in weiteren Schritten verfeinert werden. Dazu können Sie zum Beispiel für die einzelnen Anwendungsfälle weitere detaillierte Anwendungsfalldiagramme erstellen.

Bitte beachten Sie:

Ein Diagramm sollte möglichst übersichtlich bleiben. Im Idealfall passt es auf eine Seite. Erstellen Sie lieber mehrere Einzeldiagramme, als zu viele Informationen in ein einzelnes Diagramm zu „quetschen“.



Für den Anwendungsfall „Fahrzeug vermieten“ könnte das verfeinerte Anwendungsfalldiagramm dann zum Beispiel so aussehen:

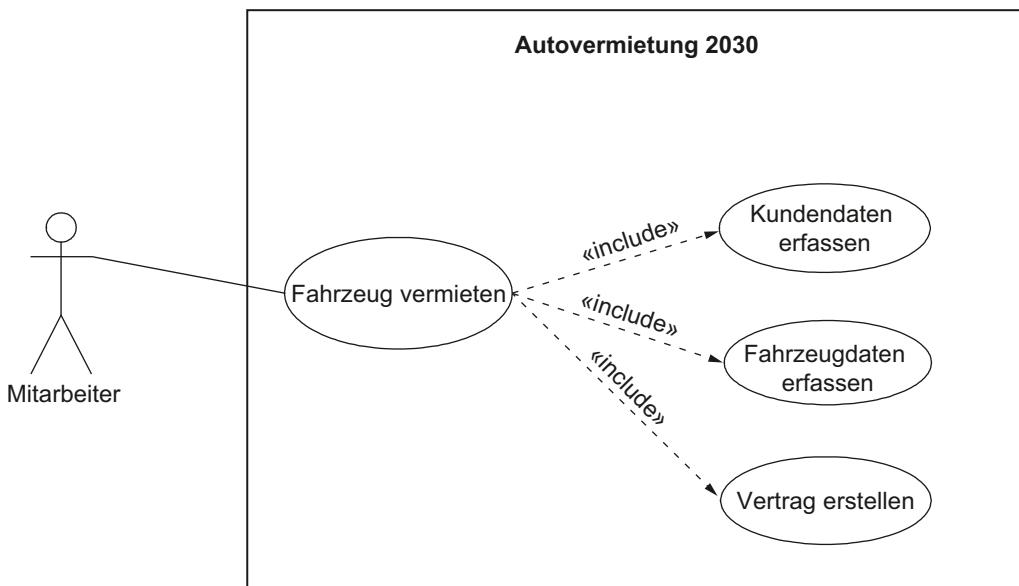


Abb. 2.5: Das verfeinerte Anwendungsfalldiagramm für „Fahrzeug vermieten“

In diesem Diagramm finden Sie jetzt auch die **include**-Beziehung wieder, die Sie ja ebenfalls bereits kennen.

Die **include**-Beziehung drückt aus, dass ein Anwendungsfall auf einen anderen zugreift beziehungsweise dass ein Anwendungsfall Teil eines anderen Anwendungsfalls ist. *include* bedeutet übersetzt so viel wie „einschließen“.



Dargestellt wird die **include**-Beziehung durch einen gestrichelten Pfeil und die Bezeichnung <<include>>. Der Pfeil zeigt dabei auf den Anwendungsfall, der in dem anderen enthalten ist. In unserem Beispiel sind also die drei Anwendungsfälle rechts im Diagramm im Anwendungsfall links im Diagramm enthalten.

Neben der **include**-Beziehung gibt es auch noch weitere Beziehungen – zum Beispiel **extend**.



Die **extend**-Beziehung drückt aus, dass ein Anwendungsfall ein Sonderfall eines anderen Anwendungsfalls ist. Er wird nur dann ausgeführt, wenn bestimmte Bedingungen zutreffen. *extend* bedeutet übersetzt so viel wie „erweitern“.

Schauen wir uns auch die **extend**-Beziehung an einem Beispiel an. Dazu erstellen wir ein verfeinertes Anwendungsfalldiagramm für den Anwendungsfall „Fahrzeug zurückgeben“.

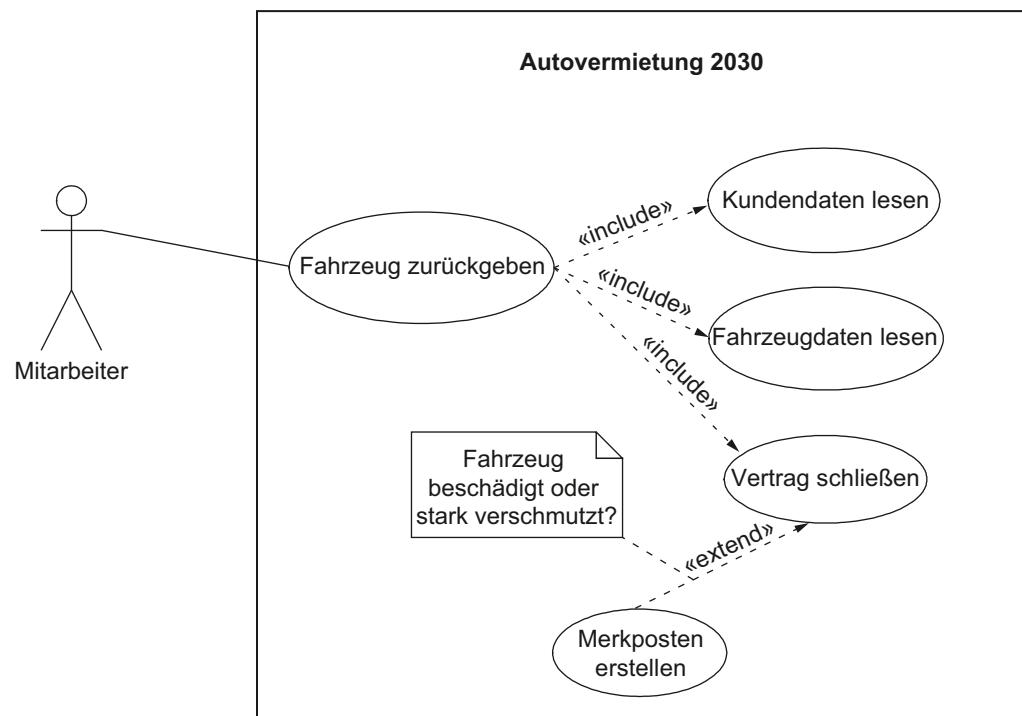


Abb. 2.6: Das verfeinerte Anwendungsfalldiagramm für „Fahrzeug zurückgeben“

Der Anwendungsfall „Merkposten erstellen“ unten im Diagramm ist ein Sonderfall des Anwendungsfalls „Vertrag schließen“. Er wird nur dann ausgeführt, wenn der Kunde das Fahrzeug beschädigt oder stark verschmutzt zurückgegeben hat, und erzeugt einen Merkposten für Folgeaktionen.

Die Darstellung der **extend**-Beziehung erfolgt ebenfalls mit einem gestrichelten Pfeil. Die Pfeilspitze zeigt dabei auf den Anwendungsfall, der erweitert wird. Die Bedingung für das Ausführen des Anwendungsfalls wird häufig über eine Notiz angegeben.

2.5 Beschreibung der Anwendungsfälle

Neben der Darstellung der Anwendungsfälle als UML-Diagramm sollten Sie die Anwendungsfälle zusätzlich auch in Textform beschreiben. Denn der Kunde beziehungsweise die Anwender sollen ja eine Überprüfung durchführen – und das ist nur möglich, wenn sie verstehen, was dargestellt wird.

Hinweis: Eine kleine Anekdote:

Bei der Entwicklung eines komplexen Systems wurden in der Analyse umfangreiche und detaillierte Anwendungsfalldiagramme ohne weitere Beschreibung erstellt.

Diese Diagramme wurden dem Kunden mit der Bitte um Prüfung vorgelegt. Bereits nach kurzer Zeit kamen die Diagramme mit folgender Notiz zurück: „Was haben Männchen, die Pfeile auf Kreise werfen, mit unserer Software zu tun?“

In welcher Form und wie umfangreich die Beschreibung erfolgt, ist nicht festgelegt. Be währt hat sich aber ein formales Vorgehen in Tabellenform. In dieser Tabelle werden dabei zum Beispiel auch Eingangsbedingungen, Ausnahmen und die Auswirkungen des Anwendungsfalls beschrieben. Die Eingangsbedingungen stellen dabei dar, wann beziehungsweise unter welchen Umständen der Anwendungsfall ausgeführt werden kann.

Bitte beachten Sie:

Die Beschreibung muss für die Anwender verständlich sein. Vermeiden Sie daher auch hier „Fachchinesisch“. Wenn sich Fachausdrücke nicht vermeiden lassen, verweisen Sie auf das Glossar aus dem Lastenheft.



Für den Anwendungsfall „Fahrzeug zurückgeben“ könnte die Beschreibung zum Beispiel so aussehen:

Tab. 2.1: Beschreibung für den Anwendungsfall „Fahrzeug zurückgeben“

Anwendungsfall	Fahrzeug zurückgeben
Akteur	Mitarbeiter
Zweck	Das ausgeliehene Fahrzeug, das ein Kunde zurückbringt, wird wieder als vorhanden im System erfasst. Der Mietvertrag mit dem Kunden wird geschlossen.
Eingangsbedingung	keine Der Anwendungsfall kann jederzeit gestartet werden.
Beschreibung	Im ersten Schritt werden die Daten des Kunden eingelesen. Danach werden die Daten des zurückgegebenen Fahrzeugs eingelesen. Anhand dieser Daten sucht das System nach dem entsprechenden Mietvertrag und markiert das Fahrzeug wieder als vorhanden. Danach wird der Mietvertrag geschlossen.
Ausnahmen	Der Kunde gibt das Fahrzeug beschädigt oder stark verschmutzt zurück. In diesem Fall wird ein „Merkposten“ über Folgeaktionen erzeugt. Danach wird der Mietvertrag geschlossen.

Auswirkungen	Das zurückgegebene Fahrzeug gilt wieder als verfügbar. Der Mietvertrag ist geschlossen.
andere benutzte Anwendungsfälle	Kundendaten lesen Fahrzeugdaten lesen Vertrag schließen Merkposten erzeugen (nur bei den Ausnahmen)
Besonderheiten	keine
Verweis auf das Lastenheft	siehe LF50

Zum Teil finden Sie in der Literatur bei der Beschreibung der Anwendungsfälle auch detaillierte Ablaufbeschreibungen – also Darstellungen, wie der Anwendungsfall durchlaufen wird. Wir erstellen diese Beschreibungen aber in einem späteren Schritt und benutzen dazu unter anderem auch Aktivitätsdiagramme.

2.6 Kategorisierung der Anwendungsfälle

Im letzten Schritt werden jetzt die Anwendungsfälle nach Prioritäten kategorisiert. Dadurch wird festgelegt, wann der Anwendungsfall technisch umgesetzt werden soll – also wann mit der Programmierung des Anwendungsfalls begonnen wird. Je höher die Priorität, desto eher muss mit der Umsetzung begonnen werden.

Wie Sie die Prioritäten im Detail setzen, müssen Sie im Einzelfall selbst entscheiden. Grundsätzlich gilt, dass Anwendungsfälle mit grundlegenden Funktionen die höchste Priorität erhalten müssen. Sie sind ja für das System besonders wichtig.

In unserem Beispiel wären das die Anwendungsfälle:

- Kundendaten pflegen,
- Fahrzeugdaten pflegen,
- Fahrzeug vermieten und
- Fahrzeug zurückgeben.

Ohne diese Anwendungsfälle wäre das gesamte System nicht zu gebrauchen.

Die Anwendungsfälle „Fahrzeug reservieren“ und „Listen erstellen“ können dagegen durchaus auch niedrigere Prioritäten erhalten. Sie sind ja für eine korrekte Funktion des Systems nicht zwingend erforderlich.

So viel zum groben dynamischen Modell. Im nächsten Kapitel werden wir mit dem Erstellen des groben statischen Modells beginnen. Dazu benutzen wir vor allem Klassendiagramme.

Zusammenfassung

Das erste grobe, dynamische Modell wird in mehreren Schritten erstellt.

Die wesentlichen Produktfunktionen können Sie zum Beispiel aus dem Lastenheft ermitteln.

Im Anwendungsfalldiagramm werden die Akteure und die Anwendungsfälle dargestellt.

Zusätzlich sollten Sie die Anwendungsfälle noch in Textform beschreiben.

Aufgaben zur Selbstüberprüfung

- 2.1 Was verstehen Sie unter den Systemgrenzen? Wie werden die Systemgrenzen im Anwendungsfalldiagramm dargestellt?

- 2.2 Wird jeder Akteur aus dem Lastenheft auch im Anwendungsfalldiagramm dargestellt? Begründen Sie bitte kurz Ihre Antwort.

- 2.3 Was unterscheidet eine **extend**- und eine **include**-Beziehung?

- 2.4 Wie wird eine **extend**-Beziehung im Anwendungsfalldiagramm dargestellt?

3 Das erste, grobe statische Modell

In diesem Kapitel erstellen wir das erste grobe, statische Modell für unser Software-Projekt Autovermietung 2030.

Dazu werden die Produktdaten – also die Daten, die unser Programm verarbeiten soll – abstrahiert und über Objekt- beziehungsweise Klassendiagramme modelliert. Das Ergebnis ist ein erstes, grobes **konzeptionelles Datenmodell** – auch **konzeptuelles Datenmodell** oder **Domänenmodell** genannt.

Hinweis:

Eine Domäne ist – allgemein ausgedrückt – ein eindeutig abgegrenzter Bereich. Konzeptionell bedeutet „die Konzeption betreffend“ oder „in Bezug auf die Konzeption“. Konzeptuell bedeutet „ein Konzept aufweisend“.



Bitte beachten Sie:

Das konzeptionelle Datenmodell beschreibt die Daten, die das System verarbeiten soll, und ihre Beziehung untereinander. Wie diese Daten tatsächlich gespeichert werden, spielt im konzeptionellen Datenmodell keine Rolle. Das konzeptionelle Datenmodell bildet also zum Beispiel **nicht** die Speicherung von Informationen in einem Datenbanksystem ab.

Auch für das Erstellen des ersten, groben statischen Modells hat sich ein schrittweises Vorgehen in der Praxis bewährt:

1. Es werden Kandidaten für Objekte beziehungsweise Klassen bestimmt.
2. Die Beziehungen der Objekte beziehungsweise Klassen untereinander werden ermittelt.
3. Es werden Attribute für die Objekte beziehungsweise Klassen ermittelt.
4. Die Objekte beziehungsweise Klassen werden über entsprechende UML-Diagramme dargestellt.

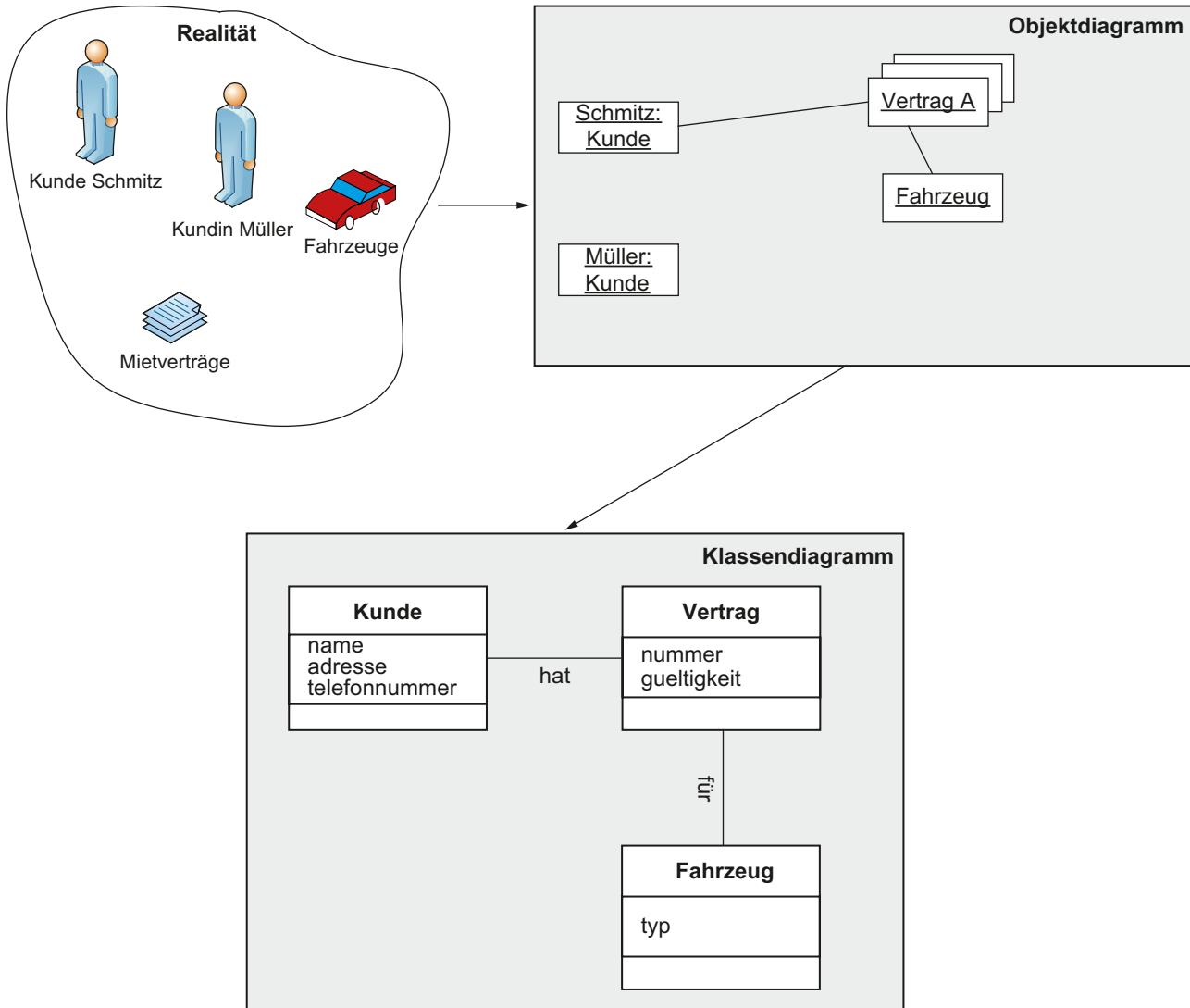


Abb. 3.1: Von der Realität zum Klassendiagramm

Die Objektdiagramme bilden dabei lediglich einen Zwischenschritt. Sie sind nicht in jedem Fall erforderlich.

Bitte beachten Sie:

Beim ersten, groben statischen Modell werden in der Regel nur die Attribute der Objekte beziehungsweise Klassen ermittelt. Die Methoden werden später hinzugefügt.



Schauen wir uns nun die einzelnen Schritte am Beispiel für unsere Autovermietung an. Beginnen wir mit dem Ermitteln der Objekt- beziehungsweise Klassenkandidaten.

3.1 Objekt- und Klassenkandidaten ermitteln

Im ersten Schritt stellt sich die Frage:



Was sind überhaupt Kandidaten für Objekte und Klassen?

Hier geht es also darum, festzustellen, was aus der realen Welt im System abgebildet werden kann beziehungsweise abgebildet werden muss.

Grundsätzlich infrage kommen:

- konkrete Gegenstände (zum Beispiel ein Fahrzeug),
- abstrakte Begriffe (zum Beispiel Adresse),
- Personen und Personenrollen (zum Beispiel Mitarbeiter oder Kunde),
- schriftliche Dokumente (zum Beispiel Verträge oder Kataloge),
- Ereignisse (zum Beispiel ein Unfall),
- Transaktionen (zum Beispiel Bezahlung)
- und so weiter.



Eine Transaktion fasst oft mehrere Einzelschritte zu einem logischen Schritt zusammen.

Wir benötigen für unser Software-Produkt in jedem Fall folgende Objekte beziehungsweise Klassen:

- Kunde,
- Fahrzeug,
- Vertrag und
- Reservierung.

Bei der Festlegung der Kandidaten sollten Sie sich nach Möglichkeit noch zusätzlich absichern – zum Beispiel über eine **Dokumentanalyse**.

Dazu nehmen Sie vorhandene Formulare oder andere Dokumente aus dem Einsatzgebiet des Systems genau unter die Lupe und prüfen, welche Daten dort festgehalten werden. Diese Daten wiederum können dann als Klassenkandidaten verwendet werden. Schauen wir uns das einmal am Beispiel eines alten Mietvertrags aus der Autovermietung an:

Kundendaten			
Kunde	Nr.: _____	Ausleihdatum: _____	
Name, Vorname: _____		Rückgabedatum: _____	
Anschrift: _____		Ort: _____	
Telefon: _____		Vertragsnummer: _____	
Fahrzeugdaten		Vertragsdaten	
Hersteller: _____		pro Tag: _____	
Typ: _____		Sonderkosten: _____	
Modell: _____		Gesamtkosten: _____	
Kennzeichen: _____			

Fahrzeugdaten

Abb. 3.2: Ein analysierter Mietvertrag

In dem Mietvertrag finden sich oben einige Kundendaten, unten Daten zum Fahrzeug und rechts Vertragsdaten.

Die Dokumentanalyse liefert Ihnen nicht nur Hinweise für Klassenkandidaten, sondern auch bereits wichtige Informationen für einen weiteren Schritt – die Ermittlung der Attribute für die Klassen. Im Mietvertrag der Autovermietung finden Sie ja zum Beispiel auch schon Daten, die für einen Kunden verarbeitet werden müssen, oder einige Daten für den Vertrag.

Für unsere Autovermietung hat die Dokumentanalyse keine weiteren wichtigen Klassenkandidaten ergeben. Es bleibt also bei den vier Klassen „Fahrzeug“, „Kunde“, „Vertrag“ und „Reservierung“.

Trotzdem war die Dokumentanalyse nicht umsonst. Wir haben hier ja wichtige Informationen für das Ermitteln der Attribute gewonnen.

3.2 Beziehungen ermitteln

Im nächsten Schritt ermitteln Sie die Beziehungen zwischen den Objekten beziehungsweise Klassen. Diese Beziehungen werden auch **Assoziation** genannt.

Eine Assoziation ist – allgemein ausgedrückt – ein Zusammenschluss oder eine Vereinigung.



Dazu können Sie zum Beispiel folgende Fragen verwenden:

- Ist eine Klasse Bestandteil einer anderen Klasse?
- Benutzt eine Klasse eine andere Klasse?
- Verweist eine Klasse auf eine andere Klasse?
- Besitzt eine Klasse eine andere?
- Kommuniziert eine Klasse mit einer anderen?

Wie die Assoziation genau aussieht, ist in diesem Schritt zunächst nicht weiter wichtig. Es geht lediglich darum, herauszufinden, ob überhaupt eine Beziehung zwischen einzelnen Klassen besteht.

In unserem Beispiel ist das Ermitteln der Assoziationen nicht weiter schwierig:

- Ein Kunde hat einen Mietvertrag.
- Dieser Mietvertrag verweist auf ein Fahrzeug.
- Außerdem kann ein Kunde eine Reservierung vornehmen.
- Die Reservierung verweist ebenfalls auf ein Fahrzeug.

3.3 Attribute ermitteln

Nachdem Sie die Assoziationen ermittelt haben, können Sie sich nun um die Attribute für die einzelnen Klassen kümmern.



Zur Erinnerung:

Attribute speichern den Zustand eines Objekts. In C# werden die Attribute Felder genannt.

Einen Teil der Attribute haben wir ja bereits durch die Dokumentanalyse erhalten. Sie finden diese Attribute noch einmal zusammengefasst in der Tab. 3.1.

Tab. 3.1: Die Attribute für die Klassen aus der Dokumentanalyse

Klasse	Attribute
Fahrzeug	Nummer, Bezeichnung, Kennzeichen, Preis pro Tag
Kunde	Nummer, Name, Vorname, Anschrift, Telefon
Vertrag	Nummer, Ausleihdatum, Rückgabe bis, Gesamtpreis

Weitere Attribute haben wir durch die Liste der Klassenkandidaten erhalten.

- Um nicht für jeden Fahrzeugtyp eine eigene Klasse erstellen zu müssen, soll der Fahrzeugtyp als Attribut in der Klasse „Fahrzeug“ gespeichert werden.
- Zusätzlich soll für jedes Fahrzeug festgehalten werden, ob es ausgeliehen ist. Dazu ergänzen wir ein weiteres Attribut für die Klasse „Fahrzeug“.

Wenn wir diese weiteren Attribute ergänzen, sieht die Tabelle so aus:

Tab. 3.2: Die ergänzten Attribute für die Klassen

Klasse	Attribute
Fahrzeug	Nummer, Bezeichnung, Kennzeichen, Preis pro Tag, Typ, ist ausgeliehen
Kunde	Nummer, Name, Vorname, Anschrift, Telefon
Vertrag	Nummer, Ausleihdatum, Rückgabe bis, Gesamtpreis

Jetzt fehlen nur noch die Attribute für die Reservierung. Hier benötigen wir nur ein eindeutiges Kennzeichen für die Identifizierung und ein Datum, wie lange die Reservierung bestehen bleiben soll.

Die vollständige Tabelle mit den Attributen sieht dann so aus:

Tab. 3.3: Die Attribute für die Klassen

Klasse	Attribute
Fahrzeug	Nummer, Bezeichnung, Kennzeichen, Preis pro Tag, Typ, ist ausgeliehen
Kunde	Nummer, Name, Vorname, Anschrift, Telefon
Vertrag	Nummer, Ausleihdatum, Rückgabe bis, Gesamtpreis
Reservierung	Nummer, gültig bis

Damit haben wir alle wichtigen Attribute erfasst und können nun im nächsten Schritt das erste Klassendiagramm für unsere Autovermietung erstellen.

3.4 Klassendiagramm erstellen

Klassendiagramme haben Sie ja ebenfalls bereits kennengelernt.

Wiederholen wir noch einmal:

Im Klassendiagramm werden Klassen und ihre Beziehungen untereinander beschrieben. Die Klassen werden dabei durch ein Rechteck dargestellt. Oben im Rechteck steht der Name der Klasse, darunter folgen die Attribute und Methoden. Die Beziehungen werden durch verschiedene Linien dargestellt.



Bei der Namensvergabe an Klassen, Attribute und Methoden gibt es einige Konventionen, die die Lesbarkeit des Klassendiagramms verbessern sollen. Einige dieser Konventionen kennen Sie auch bereits von der Namensvergabe an C#-Objekte.

- Benutzen Sie möglichst aussagekräftige und kurze Namen, die bereits erste Hinweise auf die Verwendung der Klasse beziehungsweise des Attributs geben.

Einige Beispiele:

Ein Klassename F ist zwar sehr kurz, sagt aber überhaupt nichts über die Verwendung aus.

Ein Klassename „Fahrzeug der Autovermietung“ ist zwar aussagekräftig, aber durch die Länge recht unhandlich.

Gut geeignet wäre hier „Fahrzeug“.

- Klassen werden in der Regel durch ein Substantiv im Singular bezeichnet. Der erste Buchstabe wird großgeschrieben.
- Bei Attributen wird ein Substantiv oder ein Adjektiv beziehungsweise eine Kombination aus beidem benutzt. Der erste Buchstabe wird in der Regel kleingeschrieben. Leerzeichen werden im Namen eines Attributs nicht verwendet. Stattdessen wird das neue Wort mit einem Großbuchstaben angefangen.
- Der Name der Klasse wird im Klassendiagramm fett und zentriert dargestellt. Die Attributnamen werden nicht besonders formatiert.

Damit Klassen und Attribute unterschieden werden können, muss der Name eindeutig sein. Sie können also nicht zwei Klassen mit dem Namen **Person** erstellen und in einer Klasse nicht zweimal ein Attribut **name** verwenden.

Hinweis:

In vielen Unternehmen gibt es eigene Regeln für die Namensvergabe. Fragen Sie im Zweifelsfall nach.

Häufig werden auch englische Bezeichnungen benutzt – zum Beispiel *Client* statt Kunde und *Contract* statt Vertrag. Wir bleiben hier aber bei deutschen Bezeichnungen, um das Ganze nicht unnötig kompliziert zu machen.

Die Vergabe der Bezeichner im Klassendiagramm ist grundsätzlich erst einmal unabhängig von einer Programmiersprache. Sie können aber natürlich auch schon im Klassendiagramm die Bezeichner verwenden, die Sie später im echten Programm benutzen wollen.

Schauen wir uns jetzt das erste, grobe Klassendiagramm für unsere Autovermietung an. Die Klassen und Attribute übernehmen wir dabei aus der vorigen Tabelle. Die Namen der Attribute passen wir noch ein wenig an, um zum Beispiel die verschiedenen Nummern sofort unterscheiden zu können.

Hinweis:

Grundsätzlich könnten Sie in verschiedenen Klassen jeweils ein Attribut mit dem Namen **nummer** verwenden. Dann ist später aber nicht sofort eindeutig zu erkennen, zu welcher Klasse das Attribut denn nun gehört. Daher sollten Sie bei den Attributnamen immer auf eindeutige Bezeichnungen achten, auch wenn es nicht zwingend erforderlich ist.

Die Assoziationen zwischen den Klassen haben wir ja ebenfalls bereits festgelegt – und zwar vor dem Bestimmen der Attribute. Das Klassendiagramm könnte zum Beispiel so aussehen:

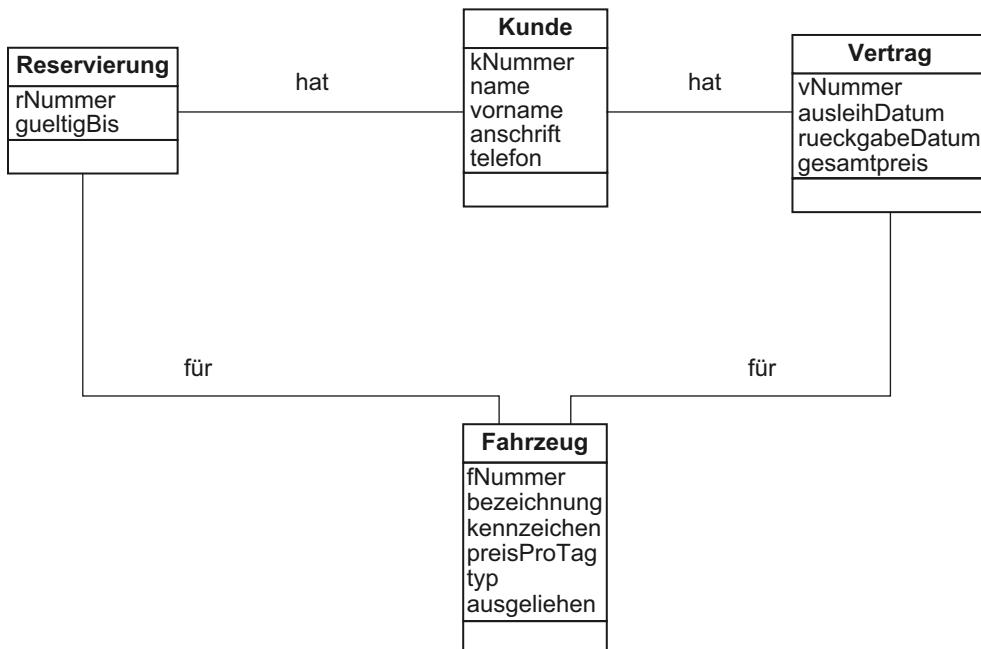


Abb. 3.3: Das erste Klassendiagramm für Autovermietung 2030

Bitte beachten Sie:

Die Methoden ergänzen wir erst bei der Verfeinerung. Daher ist der untere Bereich der Klassen in der vorherigen Abbildung noch leer.



Bei den Assoziationen haben wir zusätzlich noch eine kurze Beschreibung hinzugefügt. Das ist zwar nicht vorgeschrieben, verbessert aber die Lesbarkeit und damit die Verständlichkeit des Diagramms erheblich.

So viel zu den Klassendiagrammen. Schauen wir uns zum Ende dieses Kapitels noch kurz ein **Objektdiagramm** an.

In einem Objektdiagramm werden keine abstrahierten Klassen, sondern konkrete Objekte mit ihren Attributen dargestellt.



Die Darstellung der Objekte erfolgt wieder mit Rechtecken. Oben im Rechteck wird der Name des Objekts angegeben. Dahinter steht – getrennt durch einen Doppelpunkt – gegebenenfalls der Name der Klasse. Der gesamte Name wird unterstrichen. Darunter folgen dann die Attribute mit den konkreten Attributwerten für das Objekt.

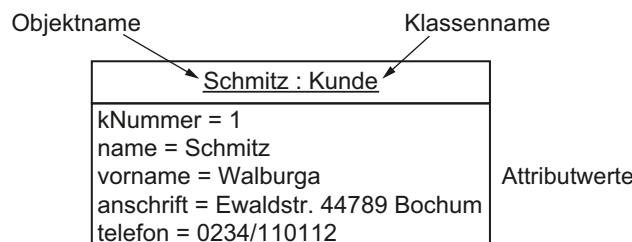


Abb. 3.4: Darstellung eines Objekts

Das folgende Objektdiagramm stellt zum Beispiel dar, dass die Kundin Frau Walburga Schmitz eine Reservierung für ein Fahrzeug vorgenommen hat.

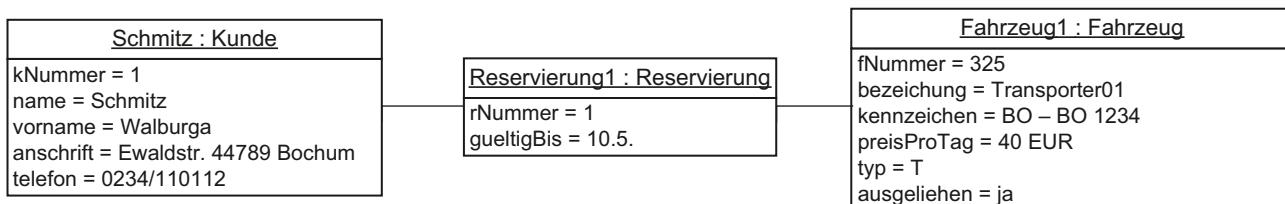


Abb. 3.5: Ein Objektdiagramm

Zusammenfassung

Das Erstellen des ersten, groben statischen Modells erfolgt in mehreren Schritten. Zuerst ermitteln Sie Kandidaten für Klassen und anschließend die Beziehungen zwischen den Klassen. Danach ermitteln Sie die Attribute und erstellen ein Klassendiagramm.

Im Klassendiagramm werden Klassen durch ein Rechteck dargestellt. Oben im Rechteck steht fett und zentriert der Name der Klasse. Darunter folgen die Attribute.

Für die Namensvergabe an Klassen und Attribute gibt es einige Konventionen. So sollte der Klassename zum Beispiel immer ein Substantiv im Singular sein und großgeschrieben werden.

Aufgaben zur Selbstüberprüfung

- 3.1 Was ist das konzeptionelle Datenmodell?

- 3.2 Müssen Sie in jedem Fall ein Objektdiagramm erstellen?

- 3.3 Was wird im ersten, groben Klassendiagramm dargestellt: Die Attribute einer Klasse oder die Methoden beziehungsweise Operationen einer Klasse?

- 3.4 Welche Techniken für die Ermittlung von Klassenkandidaten kennen Sie?

- 3.5 Was ist eine Assoziation zwischen Klassen?

- 3.6 In einem UML-Diagramm werden verschiedene Rechtecke dargestellt. Oben in jedem Rechteck steht ein unterstrichener Name. Handelt es sich um ein Objekt- oder um ein Klassendiagramm?

4 Die Verfeinerung des dynamischen Modells

In diesem Kapitel beschäftigen wir uns mit der Verfeinerung des dynamischen Modells. Wir werden die bisher grob skizzierten Funktionen detaillierter darstellen.

Für diese detaillierte Darstellung können Sie zum Beispiel **Aktivitätsdiagramme** verwenden. Diese Diagrammform haben Sie ja ebenfalls bereits kurz kennengelernt.



Zur Wiederholung:

Das Aktivitätsdiagramm wird für die Darstellung von Aktivitäten in einem Prozess benutzt. Die Aktivitäten werden in Aktionen unterteilt. Diese Aktionen werden durch Rechtecke mit gerundeten Ecken dargestellt und durch Pfeile verbunden. So lässt sich auch die Reihenfolge der Aktionen ablesen.

Hinweis:

Sie müssen nicht für jeden Anwendungsfall auch ein eigenes Aktivitätsdiagramm erstellen. Die Aktivitätsdiagramme werden vor allem dann eingesetzt, wenn komplexere Abläufe aufgegliedert werden sollen.

Schauen wir uns jetzt die einzelnen Elemente in einem Aktivitätsdiagramm einmal genauer an.

4.1 Die Elemente in einem Aktivitätsdiagramm

Eine **Aktion** wird durch ein Rechteck mit gerundeten Ecken dargestellt. In dem Rechteck steht eine Beschreibung der Aktion.

Kundendaten lesen

Abb. 4.1: Darstellung einer Aktion

Die Verbindung zwischen den Aktionen erfolgt durch Linien. Sie legen die Reihenfolge der durchzuführenden Aktionen fest. Dabei gilt im Normalfall, dass eine Aktion erst vollständig durchgeführt sein muss, bevor die folgende Aktion begonnen wird.



Abb. 4.2: Verbindung von Aktionen durch Linien

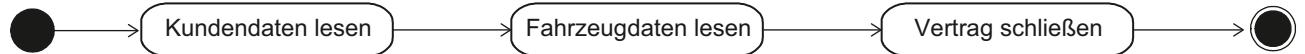


Durch ein Aktivitätsdiagramm lässt sich – anders als beim Anwendungsfalldiagramm – auch die zeitliche Reihenfolge von Aktionen beschreiben.

Der Anfang und das Ende werden durch spezielle Symbole dargestellt. Für den Start wird ein ausgefüllter Kreis verwendet und für das Ende ein ausgefüllter Kreis mit einem weiteren Kreis außen.

**Abb. 4.3:** Anfang und Ende in einem Aktivitätsdiagramm

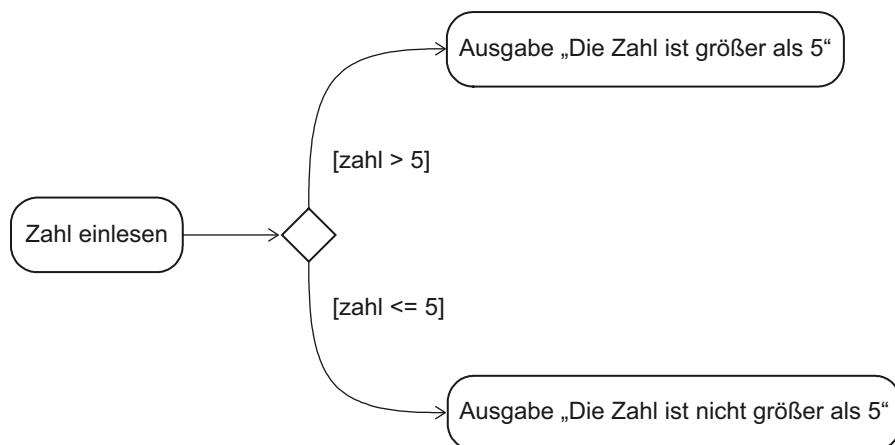
Für den Anwendungsfall „Fahrzeug zurückgeben“ könnte ein einfaches Aktivitätsdiagramm zum Beispiel so aussehen:

**Abb. 4.4:** Einfaches Aktivitätsdiagramm für den Anwendungsfall „Fahrzeug zurückgeben“

Hier wird deutlich, dass erst die Kundendaten gelesen werden müssen und danach die Fahrzeugdaten. Anschließend kann dann der Vertrag gelöscht werden.

Auch Verzweigungen, die in Abhängigkeit von einer Bedingung ausgeführt werden sollen, lassen sich in einem Aktivitätsdiagramm darstellen. Dazu wird eine Raute \diamond als Symbol verwendet. An den Verbindungslinien wird dann in eckigen Klammern die jeweilige Bedingung angegeben.

In dem folgenden Ausschnitt aus einem Aktivitätsdiagramm wird zum Beispiel eine Zahl eingelesen und anschließend überprüft, ob die Zahl größer ist als 5. Wenn das der Fall ist, wird der Text „Die Zahl ist größer als 5“ ausgegeben. Ist die Zahl kleiner oder gleich 5, wird der Text „Die Zahl ist nicht größer als 5“ ausgegeben.

**Abb. 4.5:** Eine Entscheidung in einem Aktivitätsdiagramm

In unserem Anwendungsfall „Fahrzeug zurückgeben“ können wir mit dieser Technik jetzt auch abbilden, dass für beschädigte oder stark verschmutzte Fahrzeug ein Merkposten erzeugt werden soll. Dazu fügen wir nach dem Einlesen des Fahrzeugs eine Verzweigung ein, die entweder direkt zum Schließen des Vertrags geht oder aber erst einen Merkposten erzeugt. Als Bedingung verwenden wir „Fahrzeug ok“ beziehungsweise „Fahrzeug beschädigt oder stark verschmutzt“.

Das Aktivitätsdiagramm könnte dann so aussehen:

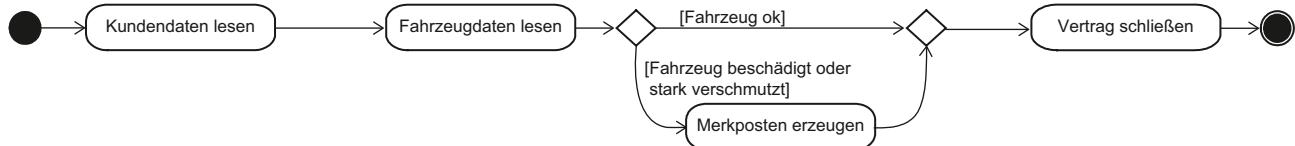


Abb. 4.6: Das Aktivitätsdiagramm für den Anwendungsfall „Fahrzeug zurückgeben“ mit einer Verzweigung

Hinweis:

Die Stelle, an der die verschiedenen Zweige wieder zusammentreffen, wird ebenfalls durch das Symbol \diamond gekennzeichnet.

Neben Verzweigungen, die an eine Bedingung geknüpft sind, können Sie in einem Aktivitätsdiagramm auch eine Teilung und eine Vereinigung von Aktivitäten ohne Bedingungen darstellen.

Eine **Teilung** ohne Bedingung liegt zum Beispiel dann vor, wenn mehrere Aktionen parallel laufen. Bei einer **Vereinigung** laufen mehrere Aktionen ohne Bedingung wieder zusammen. Die Stellen, an denen die Aktionen auseinander- beziehungsweise zusammenlaufen, werden durch eine kurze dicke Linie im Diagramm abgebildet.



Die Teilung wird im Fachjargon auch **splitting** oder **fork** genannt.^{a)} Die Vereinigung wird auch **Synchronisation** oder **join** genannt.^{b)}

- a) *Splitting* stammt vom englischen *to split* („zerspalten, zerreißen“). *Fork* bedeutet übersetzt „Gabel“ oder „Vergabelung“.
- b) Eine Synchronisation ist – allgemein ausgedrückt – eine zeitliche Abstimmung. *Join* bedeutet übersetzt „etwas verbinden“.

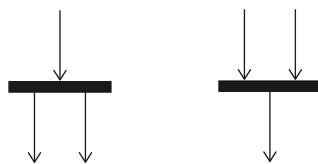


Abb. 4.7: Teilung (links) und Vereinigung (rechts) in einem Aktivitätsdiagramm

Hinweis:

Aktivitätsdiagramme können sehr schnell enorme Ausmaße annehmen und werden dann allein durch ihre Größe sehr unübersichtlich. Untergliedern Sie daher komplexe Vorgänge weiter und erstellen Sie mehrere getrennte Aktivitätsdiagramme.

So viel zum UML-Aktivitätsdiagramm. Sehen wir uns zum Abschluss dieses Kapitels noch eine Möglichkeit zur sprachlichen Beschreibung von Aktivitäten an.

4.2 Beschreibung der Aktivitäten

Bei komplexen Aktivitäten sollten Sie zusätzlich noch eine sprachliche Ablaufbeschreibung erstellen. Dazu hat sich eine Tabelle bewährt, in der links die Aktionen des Anwenders aufgeführt werden und rechts die Systemantwort. Die einzelnen Aktivitäten werden dabei fortlaufend nummeriert.

Für die Rückgabe eines Fahrzeugs könnte diese Tabelle zum Beispiel so aussehen:

Tab. 4.1: Ablaufbeschreibung für „Fahrzeug zurückgeben“

Aktion	Systemantwort
1. Start	2. Funktion wird gestartet.
3. Kundendaten werden eingelesen.	4. Die Kundendaten werden zwischen- gespeichert.
5. Fahrzeugdaten werden eingelesen.	6. Die Fahrzeugdaten werden zwischen- gespeichert.
7. Einlesen beenden	8. Der Vertrag wird gesucht und geschlossen.

Damit die Ablaufbeschreibungen möglichst übersichtlich bleiben, werden in der Regel keine Ausnahmen dargestellt. Falls das erforderlich ist, sollten Sie eine zusätzliche getrennte Ablaufbeschreibung anlegen. Für den Fall, dass das Fahrzeug beschädigt oder stark verschmutzt ist, könnte diese Ablaufbeschreibung so aussehen.

Tab. 4.2: Ablaufbeschreibung für „Fahrzeug zurückgeben“ mit der Ausnahme „Fahrzeug ist beschädigt oder stark verschmutzt“

Aktion	Systemantwort
1. Start	2. Funktion wird gestartet.
3. Kundendaten werden eingelesen.	4. Die Kundendaten werden zwischen- gespeichert.
5. Fahrzeugdaten werden eingelesen.	6. Die Fahrzeugdaten werden zwischen- gespeichert.
7. Einlesen beenden	8. Der Vertrag wird gesucht. Es erscheint eine Abfrage, ob das Fahrzeug beschädigt oder stark verschmutzt ist.
9. Abfrage wird mit „Ja“ beantwortet	10. Vertrag wird geschlossen und ein Merkposten erzeugt.

So viel zur Verfeinerung des dynamischen Modells. Im nächsten Kapitel werden wir das statische Modell noch ein wenig erweitern.

Zusammenfassung

Das Aktivitätsdiagramm wird für die Darstellung von Aktivitäten in einem Prozess benutzt. Die einzelnen Aktivitäten – dargestellt durch Rechtecke mit gerundeten Ecken – werden dabei durch Pfeile verbunden.

In einem Aktivitätsdiagramm können auch Teilungen und Vereinigungen von mehreren Aktivitäten dargestellt werden. Außerdem kann die Ausführung einer Aktivität von einer Bedingung abhängig gemacht werden.

Aufgaben zur Selbstüberprüfung

- 4.1 Was ist ein wesentlicher Unterschied zwischen einem Anwendungsfall-diagramm und einem Aktivitätsdiagramm?

- 4.2 Wie wird eine Verzweigung, die von einer Bedingung abhängt, im Aktivitäts-diagramm dargestellt?

- 4.3 Wie lauten die Fachbegriffe für eine Teilung und eine Vereinigung ohne Bedingung im Aktivitätsdiagramm?

5 Die Verfeinerung des statischen Modells

In diesem Kapitel werden wir uns um den letzten wesentlichen Schritt bei der Analyse kümmern. Das erste, grobe statische Modell wird verfeinert.

Dazu werden die Typen für die Attribute bestimmt und die Methoden beziehungsweise die Operationen der Klassen ermittelt. Schauen wir uns die verschiedenen Schritte für die Verfeinerung des statischen Modells der Reihe nach an. Beginnen wir mit dem Festlegen der Attributtypen.

5.1 Attributtypen festlegen

In diesem Schritt geht es darum, detailliert zu beschreiben, welche Art von Daten ein Attribut speichern soll.

So können wir für die verschiedenen Nummern unserer Klassen zum Beispiel eine ganze Zahl verwenden – also einen `int`-Typ. Für den Fahrzeugtyp dagegen benutzen wir ein einziges alphanumerisches Zeichen – also einen `char`-Typ. Wir müssen hier ja lediglich die verschiedenen Fahrzeuge eindeutig voneinander unterscheiden können. Dazu verwenden wir verschiedene Buchstaben wie „P“ für „Pkw“ oder „T“ für „Transporter“.

Für das Attribut „ausgeliehen“ der Klasse „Fahrzeug“ reicht es aus, wenn wir wissen, ob das Fahrzeug ausgeliehen ist oder nicht. Wir müssen also nur zwei unterschiedliche Werte speichern können – entweder „wahr“ oder „falsch“. Das heißt, für dieses Attribut können wir den Datentyp `bool` benutzen.

Für den Kundennamen und -vornamen sowie für die Telefonnummer verwenden wir den Datentyp `string`.

Hinweis: Überlegen Sie einmal selbst

Warum sollten Sie eine Telefonnummer nicht über einen numerischen Datentyp abbilden?

Die Antwort ist recht einfach: Wenn Sie einen numerischen Typ verwenden, können Sie keine Trennzeichen für die Vorwahl und die eigentliche Rufnummer eingeben. Außerdem werden bei numerischen Typen keine führenden Nullen gespeichert. Die Vorwahl 0234 würde also bei der Speicherung als numerischer Typ auf 234 verkürzt.

Für das Ausleihdatum und das Rückgabedatum benutzen wir den Typ `DateTime`². Er kann sowohl Datums- als auch Zeitangaben speichern.

Die Preise speichern wir im Datentyp `float`.

Zu guter Letzt bilden wir noch die Anschrift des Kunden über eine Struktur ab, die wir `adresse` nennen. Diese Struktur soll die Straße, die Hausnummer sowie Postleitzahl und Ort abbilden können.

Eine zusammenfassende Darstellung der verschiedenen Attributtypen finden Sie noch einmal in der folgenden Tabelle. Die Attribute haben wir dabei aus dem Klassendia gramm aus der Abb. 3.3 übernommen.

2. `DateTime` steht für „Datum und Zeit“.

Tab. 5.1: Die Attributtypen für die Klassen unserer Autovermietung

Klasse	Attribute	Attributtyp
Fahrzeug	fNummer bezeichnung kennzeichen preisProTag typ ausgeliehen	int string string float char bool
Kunde	kNummer name vorname anschrift telefon	int string string adresse string
Vertrag	vNummer ausleihDatum rueckgabeDatum gesamtPreis	int DateTime DateTime float
Reservierung	rNummer gueltigBis	int DateTime

Hinweis:

Sie können für die Datentypen auch allgemeine Beschreibungen wie Ganze Zahl, Zeichen oder Datum benutzen. Diese allgemeinen Beschreibungen müssen Sie dann aber spätestens bei der konkreten Umsetzung des Systems mit einer Programmiersprache durch die exakten Bezeichnungen der Datentypen ersetzen, die auch die Programmiersprache verwendet.

Benutzen Sie daher bereits in diesem Schritt – wenn möglich – für die Attributtypen die Bezeichnungen, die auch in der Programmiersprache verwendet werden – in unserem Fall also die C#-Typen. Im Zweifelsfall können Sie die Bedeutung dieser Typen einem Laien relativ schnell erklären.

Tipp:

Einige UML-Tools stellen Ihnen für die Attributtypen die Standarddatentypen unterschiedlichster Programmiersprachen zur Verfügung.

Der Attributtyp wird im Klassendiagramm hinter dem Namen des Attributs angegeben und durch einen Doppelpunkt getrennt.

Beispiel 5.1:

```
kNummer : int
```

legt fest, dass das Attribut kNummer vom Typ int ist.

Wenn wir die Attributtypen aus der vorherigen Tabelle nun in unser Klassendiagramm für die Anwendung Autovermietung 2030 einfügen, sieht das Diagramm so aus:

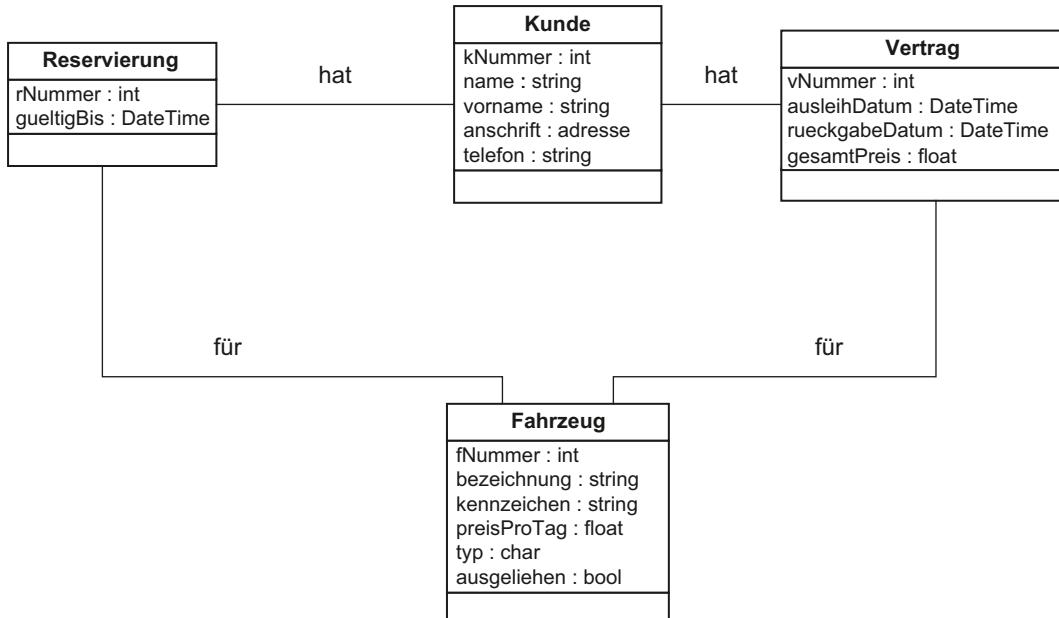


Abb. 5.1: Das Klassendiagramm für Autovermietung 2030 mit den Attributtypen

Neben dem Typ können Sie außerdem für jedes Attribut noch einen **Vorgabewert** und die **Multiplizität** festlegen.

Über den **Vorgabewert** – im Fachjargon **default value**³ genannt – können Sie einen Wert festlegen, der beim Erstellen einer Instanz der Klasse für das Attribut eingesetzt wird. Der Vorgabewert wird hinter dem Attributtyp angegeben und durch ein Gleichheitszeichen = eingeleitet.

Beispiel 5.2:

```
ausgeliehen : bool = false
```

Das Attribut `ausgeliehen` vom Typ `bool` wird mit dem Wert `false` für „falsch“ vorbelegt. Das heißt, bei jeder neuen Instanz der Klasse „Fahrzeug“ wird automatisch das Attribut `ausgeliehen` so gesetzt, dass das Fahrzeug als nicht ausgeliehen markiert ist.

Hinweis:

Durch welchen Wert der Ausleihstatus markiert wird, ist eigentlich beliebig. Sie könnten auch festlegen, dass bei einem Fahrzeug, das ausgeliehen ist, das Attribut `ausgeliehen` den Wert `true` haben soll. Das wäre allerdings unlogisch und macht das Verständnis erheblich schwieriger.

3. Default value ist die englische Übersetzung für „Vorgabewert“.

So viel zum Vorgabewert.

In der Regel kommt ein Attribut einer Klasse genau einmal vor. Es kann aber auch sein, dass ein Attribut überhaupt nicht oder aber mehrfach vorkommt. Diese Varianten können Sie über die **Multiplizität** beschreiben.



Die Multiplizität beschreibt, wie oft ein Attribut vorkommen kann beziehungsweise vorkommen muss.

Die Multiplizität wird in der Form `[Anzahl] beziehungsweise [untere Grenze..obere Grenze]` angegeben und hinter dem Typ des Attributs aufgeführt. Das Sternchen `*` steht dabei für beliebig viele Werte. Einige Beispiele für die Multiplizität finden Sie in der folgenden Tabelle:

Tab. 5.2: Beispiele für die Multiplizität

Angabe	Bedeutung
<code>[2]</code>	Hier wird nur ein konkreter Wert angeben. Dieser Wert ist gleichzeitig die untere und auch die obere Grenze. Das heißt, das Attribut muss exakt so oft vorkommen wie angegeben – in unserem Beispiel also zweimal.
<code>[0..1]</code>	Die untere Grenze ist <code>0</code> ; das heißt, das Attribut ist optional. Es muss also nicht vorkommen. Die obere Grenze ist <code>1</code> . Das bedeutet, wenn das Attribut vorkommt, kommt es maximal einmal vor.
<code>[0..*]</code>	Hier ist die obere Grenze beliebig. Das wird durch das <code>*</code> ausgedrückt. Das heißt, es können beliebig viele Attribute vorkommen. Durch die <code>0</code> als untere Grenze ist auch eingeschlossen, dass überhaupt kein Attribut vorkommt.
<code>[m..n]</code>	Hier werden <code>m</code> und <code>n</code> durch konkrete Werte ersetzt. <code>m</code> als Untergrenze muss dabei kleiner sein als <code>n</code> als Obergrenze. Die Angabe <code>[2..4]</code> legt beispielsweise fest, dass das Attribut mindestens zweimal und maximal viermal auftritt.

Der Ausdruck

```
vorname : string [1..*]
```

legt also fest, dass das Attribut `vorname` vom Typ `string` mindestens einmal vorkommen **muss** und beliebig oft vorkommen **kann**. Eine Person kann ja durchaus auch mehrere Vornamen haben.

Hinweis:

Wenn ein Attribut nur einmal vorkommt, müssen Sie keine Angaben zur Multiplizität machen. Ohne weitere Angaben wird davon ausgegangen, dass ein Attribut exakt einmal erscheint.

Schauen wir uns nun die Vorgabewerte und die Multiplizitäten für unsere Attribute in der Anwendung Autovermietung 2030 an.

Vorgabewerte verwenden wir lediglich für das Attribut `ausgeliehen` der Klasse „Fahrzeug“. Das Attribut soll immer mit dem Wert `false` belegt werden. Das heißt, ein Fahrzeug ist nicht ausgeliehen.

Multiplizitäten verwenden wir für den Vornamen eines Kunden und für die Telefonnummer. Der Vorname muss mindestens einmal vorkommen und kann maximal dreimal vorkommen. Die Telefonnummer dagegen kann auch fehlen und darf maximal zweimal erscheinen. Durch das maximal zweimalige Erscheinen können wir dann neben der Festnetznummer zum Beispiel auch noch eine Mobilfunknummer erfassen.

Das Klassendiagramm mit den Vorgabewerten und Multiplizitäten sieht dann so aus:

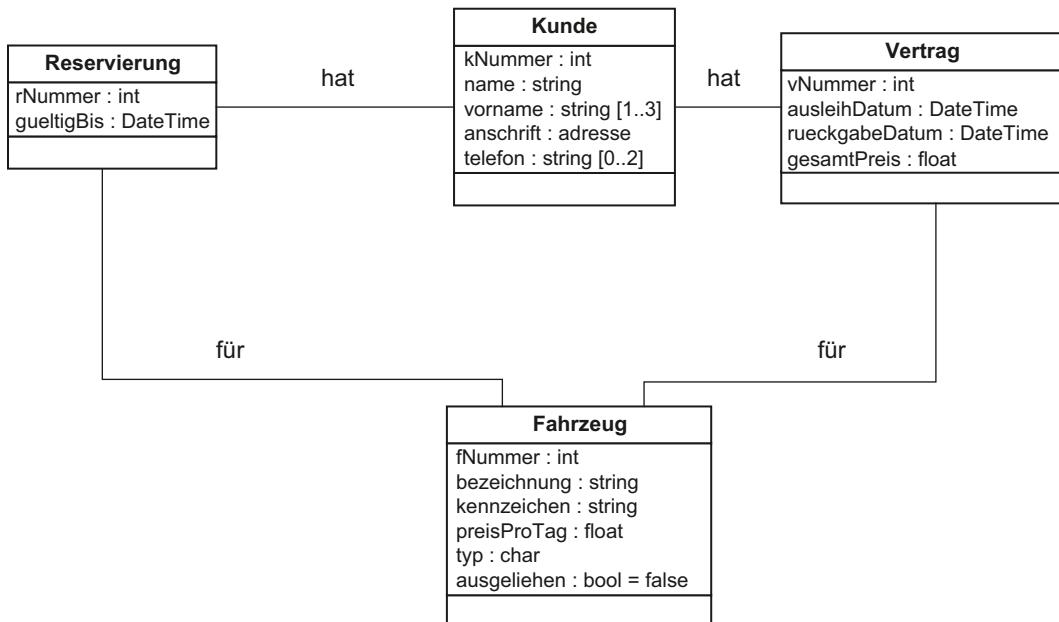


Abb. 5.2: Das Klassendiagramm für Autovermietung 2030 mit Vorgabewerten und Multiplizitäten

Damit wären die Attribute zunächst einmal vollständig. Jetzt können wir uns um die Methoden der Klassen kümmern.

5.2 Ermitteln der Methoden

Die Methoden der Klassen können Sie sehr gut aus den Anwendungsfall- und Aktivitätsdiagrammen ableiten. Hier ist ja beschrieben, was das System tun soll. Im Wesentlichen geht es jetzt darum, die Methoden den einzelnen Klassen zuzuordnen.

In unserem Fall könnte diese Zuordnung so aussehen:

Tab. 5.3: Eine erste Zuordnung der Methoden für die Klassen

Klasse	Methode
Fahrzeug	neues Fahrzeug anlegen, vorhandenes Fahrzeug löschen, vorhandenes Fahrzeug ändern, Listen erzeugen
Kunde	neuen Kunden anlegen, vorhandenen Kunden löschen, vorhandenen Kunden ändern
Vertrag	neuen Vertrag anlegen, vorhandenen Vertrag schließen, Merkposten erzeugen
Reservierung	neue Reservierung anlegen

Damit hätten wir die wesentlichen Informationen aus den Anwendungsfall- und Aktivitätsdiagrammen in entsprechenden Methoden abgebildet. Die beiden großen Anwendungsfälle „Fahrzeug vermieten“ und „Fahrzeug zurückgeben“ bilden wir über „neuen Vertrag anlegen“ und „vorhandenen Vertrag schließen“ bei der Klasse „Vertrag“ ab.

Zusätzlich benötigen wir noch zwei Methoden, die eine vorhandene Reservierung wieder löschen und einen Merkposten wieder auflösen – also ebenfalls löschen. Andernfalls würden diese Daten ja nach dem Anlegen ständig als eine Art „Karteileiche“ im System bleiben.

Die Tabelle mit den Methoden sieht jetzt so aus:

Tab. 5.4: Die zusätzlichen Methoden für die Klassen

Klasse	Methode
Fahrzeug	neues Fahrzeug anlegen, vorhandenes Fahrzeug löschen, vorhandenes Fahrzeug ändern, Listen erzeugen
Kunde	neuen Kunden anlegen, vorhandenen Kunden löschen, vorhandenen Kunden ändern
Vertrag	neuen Vertrag anlegen, vorhandenen Vertrag schließen, Merkposten erzeugen, vorhandenen Merkposten löschen
Reservierung	neue Reservierung anlegen, vorhandene Reservierung löschen

Ein wichtiger Punkt fehlt aber noch: Die Zusammenarbeit der verschiedenen Klassen. Das ist ja vor allem für die Ausleihe und die Rückgabe wichtig. Hier sollen ja im ersten Schritt zunächst einmal die Kunden- und Fahrzeugdaten gelesen werden und dann dazu die passenden Daten ermittelt werden. Das heißt, wir müssen aus der Klasse „Vertrag“, die ja neue Verträge anlegt und auch vorhandene Verträge schließt, auf Informationen aus den Klassen „Fahrzeug“ und „Kunde“ zurückgreifen.

Eine Möglichkeit wäre es, direkt aus der Klasse „Vertrag“ auf die Attribute der Klassen „Fahrzeug“ und „Kunde“ zuzugreifen. Solch ein direkter Zugriff würde aber – wie Sie ja wissen – das Prinzip der Datenkapselung verletzen.

Zur Auffrischung:

Das Prinzip der Datenkapselung beschreibt, dass der Zugriff auf die Attribute einer Klasse nur durch die Klasse selbst erfolgen sollte und nicht von außen.



Wir erstellen daher Methoden für die Klasse „Vertrag“, die mit den Klassen „Fahrzeug“ und „Kunde“ kommuniziert und uns so die nötigen Daten beschafft.

Eine Methode „Kundendaten lesen“ für die Klasse „Vertrag“ könnte zum Beispiel zunächst einmal die Daten für den Kunden einlesen und dann die Methode „Kunde suchen“ für die Klasse „Kunde“ aufrufen. Die Methode „Kunde suchen“ sucht nach dem gewünschten Kunden und gibt die Ergebnisse der Suche über eine weitere Methode „Kundeninfo senden“ an die Klasse „Vertrag“ zurück.

Mit einer ähnlichen Technik könnten wir dann auch die Fahrzeugdaten von der Klasse „Fahrzeug“ abrufen. Zunächst werden die Fahrzeugdaten in der Klasse „Vertrag“ gelesen. Anhand der eingelesenen Daten sucht dann eine Methode der Klasse „Fahrzeug“ nach den Informationen und liefert sie an die Klasse „Vertrag“ zurück.

Bei der Reservierung gehen wir ähnlich vor. Wir lesen die Daten ein und lassen dann die anderen Klassen nach den Informationen suchen.

Mit diesen zusätzlichen Methoden sieht die Tabelle dann so aus:

Tab. 5.5: Weitere Methoden für die Klassen

Klasse	Methode
Fahrzeug	neues Fahrzeug anlegen, vorhandenes Fahrzeug löschen, vorhandenes Fahrzeug ändern, Listen erzeugen, Fahrzeug suchen, Fahrzeuginfo senden
Kunde	neuen Kunden anlegen, vorhandenen Kunden löschen, vorhandenen Kunden ändern, Kunden suchen, Kundeninfo senden
Vertrag	neuen Vertrag anlegen, vorhandenen Vertrag schließen, Merkposten erzeugen, vorhandenen Merkposten löschen, Kundendaten lesen, Fahrzeugdaten lesen
Reservierung	neue Reservierung anlegen, vorhandene Reservierung löschen, Fahrzeugdaten lesen, Kundendaten lesen

5.3 Die Methoden im Klassendiagramm

Nachdem wir jetzt die nötigen Methoden ermittelt haben, wollen wir sie im Klassendiagramm darstellen.

Für die Namensvergabe an Methoden gelten dabei folgende Konventionen, die Sie in großen Teilen ebenfalls schon von C#-Namenskonventionen für Methoden kennen:

- Der Name beginnt mit einem Verb und wird – wenn erforderlich – durch ein Substantiv erweitert.

- Leerzeichen werden nicht verwendet. Stattdessen wird das neue Wort mit einem großen Buchstaben angefangen.
- Hinter dem Namen werden runde Klammern () angegeben. Damit wird sofort klar, dass es sich um eine Methode und nicht um ein Attribut handelt.

Anders als bei C# beginnen die Namen von Methoden in einem Klassendiagramm üblicherweise mit einem kleinen Buchstaben.

Das Klassendiagramm mit den Methoden sieht dann so aus:

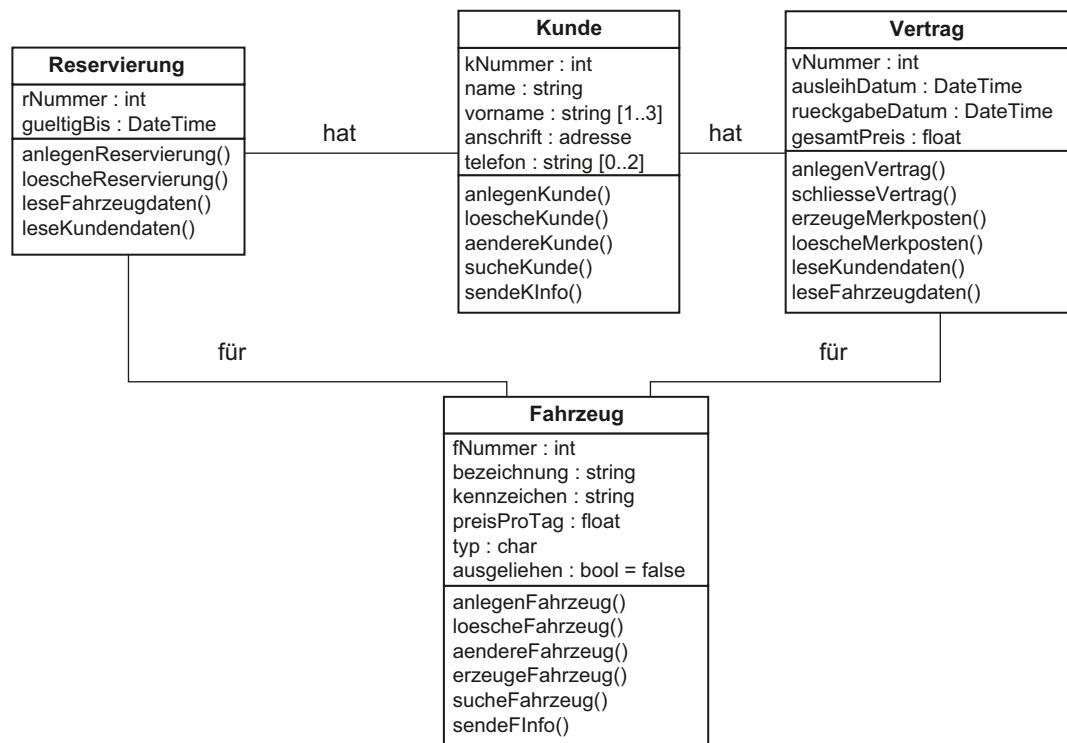


Abb. 5.3: Das Klassendiagramm mit den Methoden

Hinweis:

Einige UML-Werkzeuge erzeugen automatisch auch `get-` und `set-`Methoden, um die Werte der Attribute einer Klasse lesen und schreiben zu können. Für ein Attribut `name` könnten diese Methoden zum Beispiel so aussehen:

```

getName()
setName()
  
```

So viel an dieser Stelle zunächst einmal zum Klassendiagramm für unsere Software Autovermietung 2030.

Zusammenfassung

Bei der Verfeinerung des statischen Modells werden die Typen für die Attribute bestimmt und die Methoden beziehungsweise die Operationen der Klassen ermittelt.

Der Attributtyp wird hinter dem Namen des Attributs angegeben und durch einen Doppelpunkt vom Namen getrennt.

Neben dem Typ können Sie für jedes Attribut noch einen Vorgabewert und die Multiplizität festlegen.

Aufgaben zur Selbstüberprüfung

- 5.1 Welche Wirkung hat ein Vorgabewert für ein Attribut?

- 5.2 Was drückt die Multiplizität für ein Attribut aus?

- 5.3 Woran können Sie bereits am Namen in einem Klassendiagramm erkennen, ob es sich um ein Attribut oder eine Methode einer Klasse handelt?

6 Verfeinerung des Klassendiagramms

In diesem Kapitel erstellen wir das Entwurfsmodell für unsere Software Autovermietung 2030. Im ersten Schritt werden wir das Klassendiagramm aus der Analyse noch einmal verfeinern.

Wir präzisieren dabei die Methoden beziehungsweise Operationen der Klassen, legen die Sichtbarkeiten der Attribute und Methoden fest und untersuchen die Klassenbeziehungen noch einmal genauer.

6.1 Präzisierung der Methoden

Bisher haben wir die Methoden unserer Klassen lediglich grob beschrieben und den Namen im Klassendiagramm aufgeführt.

Bei vielen Methoden sind aber noch weitere Informationen erforderlich – zum Beispiel, welche Daten die Methode verarbeiten soll und welche Daten die Methode nach der Bearbeitung wieder zurückliefert. Sie müssen also Argumente beziehungsweise Parameter angeben und den Rückgabetyp festlegen.



Zur Auffrischung:

Über Argumente beziehungsweise Parameter legen Sie fest, welche Werte an eine Methode zur weiteren Verarbeitung übergeben werden.

Der Rückgabetyp legt fest, ob und – wenn ja – welchen Wert eine Methode zurückliefert.

Für unsere Methode `sucheKunde()` könnten wir also festlegen, dass zum Beispiel die Kundennummer als ganze Zahl verarbeitet werden soll. Die allgemeine Beschreibung der Methode würde dann so aussehen:

```
sucheKunde (kNr: int)
```



Bitte beachten Sie:

Im UML-Klassendiagramm werden Argumente in der Form `name: typ` dargestellt. Bei C# erfolgt die Angabe in der Form `typ name`.

Wenn Sie mehrere Argumente an eine Methode übergeben wollen, geben Sie sie wie von C# gewohnt hintereinander an und trennen sie durch ein Komma.



Beispiel 6.1:

```
rechnen(zahl1: int, zahl2: int)
```

Die Methode `rechnen()` erhält hier zwei Argumente – eins mit dem Namen `zahl1` und ein weiteres mit dem Namen `zahl2`. Der Typ beider Argumente ist `int` – also eine ganze Zahl.

Neben dem Namen und dem Typ können Sie für ein Argument auch noch einen Vorgabewert festlegen. Der Vorgabewert wird ganz am Ende aufgeführt und mit dem Gleichheitszeichen = eingeleitet. Unsere Methode `sucheKunde()` mit dem Vorgabewert 0 für die Kundennummer würde dann so aussehen:

```
sucheKunde(kNr: int = 0)
```

Der Vorgabewert wird allerdings nur dann benutzt, wenn beim Aufruf der Methode kein Wert für das Argument angegeben wird.

So viel zu den Argumenten.

Kommen wir nun zum Rückgabetyp. Er wird hinter den runden Klammern () im Namen der Methode angegeben und durch einen Doppelpunkt abgetrennt.

Nehmen wir einmal an, unsere Methode `sucheKunde()` soll den Wert „wahr“ liefern, wenn der Kunde gefunden wurde, und den Wert „falsch“, wenn der Kunde nicht gefunden wurde. Die Beschreibung der Methode könnte dann so aussehen:

```
sucheKunde(kNr: int = 0): bool
```

Bitte beachten Sie:

Beim Rückgabetyp müssen Sie ein wenig umdenken. Bei einer C#-Methode steht er weit vorn im Kopf der Methode, im UML-Klassendiagramm ganz hinten.



Sowohl die Argumente als auch der Typ der Rückgabe sind optional. Wenn Sie keine Argumente verwenden und keine Werte zurückgeben, lassen Sie die entsprechenden Angaben einfach weg.

Nach der Präzisierung der Methoden könnte unser Klassendiagramm für Autovermietung 2030 so aussehen:

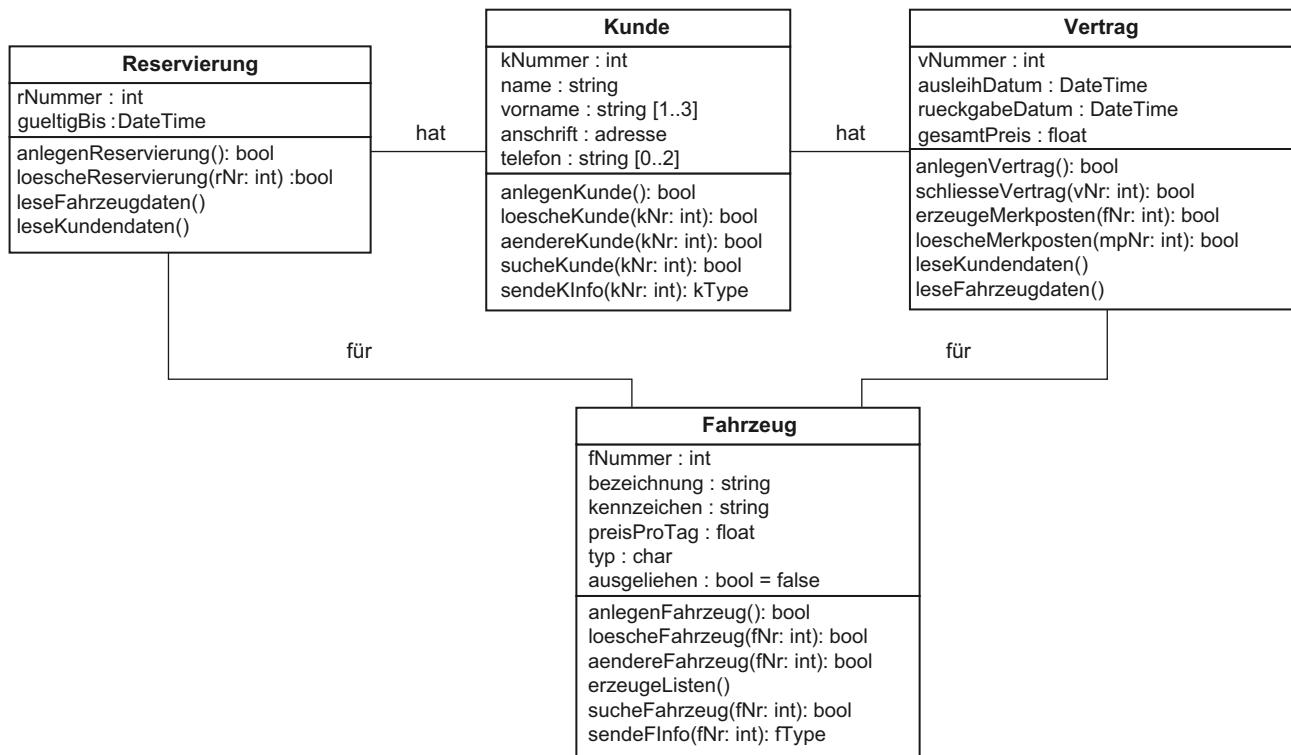


Abb. 6.1: Das Klassendiagramm mit den präzisierten Methoden

Besonderheiten gibt es eigentlich nur bei den Rückgabetypen für die Methoden `sendeKInfo()` und `sendeFInfo()`. Hier verwenden wir jeweils einen selbst definierten Typ, der nur die Informationen aus den Kunden- beziehungsweise Fahrzeugdaten enthält, die wir für das Anlegen des Vertrags benötigen.

Bei den anderen Methoden werden vor allem Nummern als Argument übergeben, die beschreiben, was die Methode genau bearbeiten soll. Die Methoden zum Erzeugen und Löschen der Merkposten verarbeiten einmal die Fahrzeugnummer (beim Erzeugen) und einmal die Nummer des Merkpostens (beim Löschen).

Der Rückgabetyp der meisten Methoden ist `bool`. Dadurch können wir prüfen, ob die Ausführung erfolgreich war oder nicht.

6.2 Ergänzen von Methoden

Jetzt fehlen noch einige Methoden, die wir aus technischer Sicht benötigen.

Für das Anlegen von Kunden, Fahrzeugen, Verträgen und Reservierungen benötigen wir eine Methode, die uns beim Anlegen eine laufende Nummer liefert. Die Methode soll ermitteln, welche Nummer jeweils zuletzt vergeben wurde, und dann den Wert um 1 erhöhen. Wenn also der letzte angelegte Kunde unter der Nummer 10 gespeichert worden wäre, erhielte der nächste Kunde die Nummer 11.

Außerdem muss beim Erzeugen eines Mietvertrags das Attribut `ausgeliehen` des jeweiligen Fahrzeugs auf `true` gesetzt werden. Damit markieren wir, dass das Fahrzeug nun nicht mehr von einem anderen Kunden ausgeliehen werden kann. Beim Schließen des Vertrags – also bei der Rückgabe – muss das Attribut `ausgeliehen` dann wieder den Wert `false` bekommen. Dieses Umschalten des Ausleihstatus setzen wir in einer Methode um, die den Wert, der gesetzt werden soll, als Argument erhält.

Eine weitere Methode soll das Ausleihdatum für den Vertrag automatisch auf das aktuelle Datum setzen.

Wenn wir diese Methoden jeweils am Ende der Klassen ergänzen, sieht das Diagramm so aus:

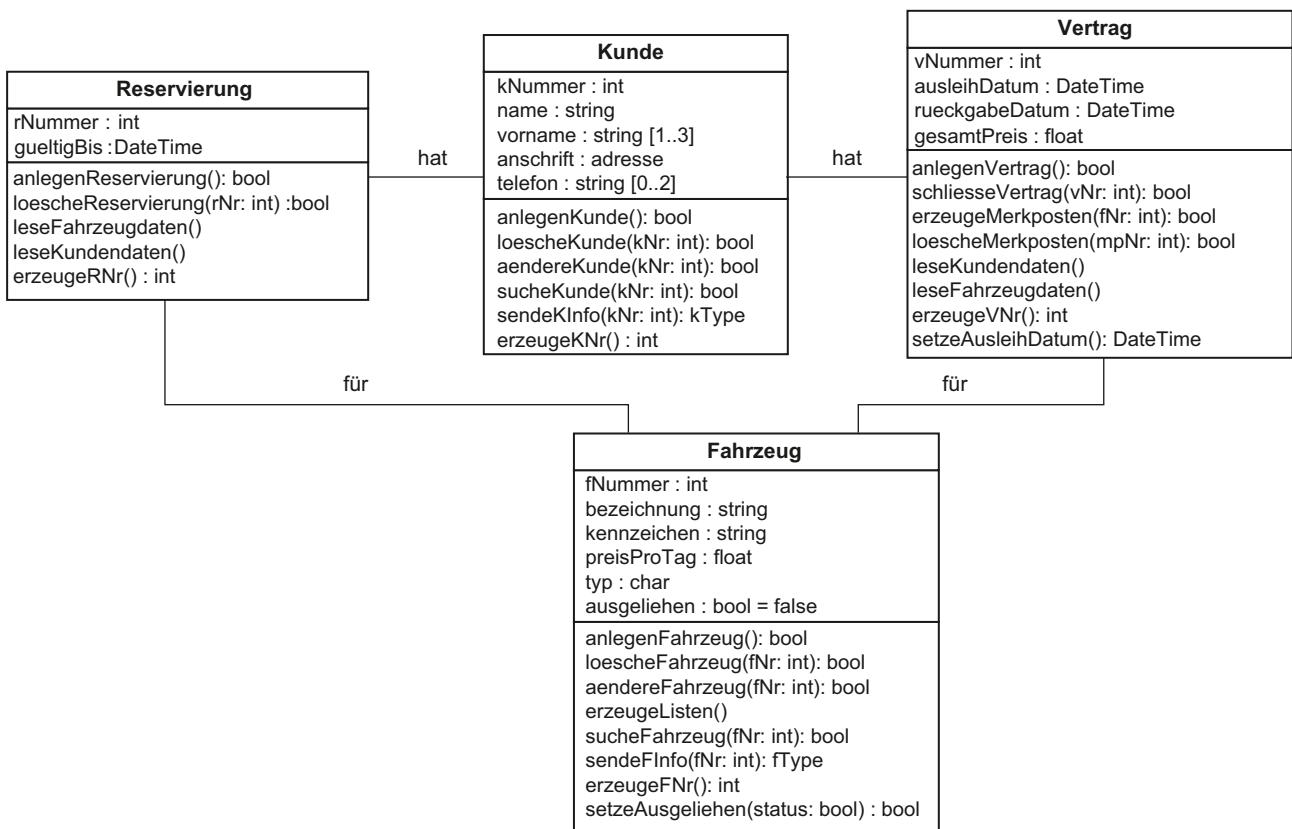


Abb. 6.2: Das Klassendiagramm mit den neuen Methoden

6.3 Festlegen der Sichtbarkeiten

Im nächsten Schritt können wir uns nun um die Sichtbarkeiten der Methoden und auch der Attribute kümmern. Diese Sichtbarkeiten kennen Sie ja bereits.

Wiederholen wir noch einmal:

Die Sichtbarkeit legt fest, ob nur eine Klasse selbst auf ihre Methoden und Attribute beziehungsweise Felder zugreifen kann oder ob der Zugriff auch anderen Klassen gestattet wird.



Anders als die Programmiersprache C#, die insgesamt fünf verschiedene Sichtbarkeiten kennt, unterscheidet die UML vier wesentliche Sichtbarkeiten.

Tab. 6.1: Sichtbarkeiten der UML

Sichtbarkeit	Auswirkung
private	Die Attribute und Methoden stehen nur der Klasse selbst zur Verfügung.
protected	Die Attribute und Methoden stehen der Klasse selbst und allen abgeleiteten Klassen zur Verfügung.
public	Die Attribute und Methoden stehen allen Klassen zur Verfügung.
package^{a)}	Die Attribute und Methoden stehen nur den Klassen im selben Paket zur Verfügung. Diese Sichtbarkeit entspricht ungefähr den Sichtbarkeiten internal beziehungsweise protected internal von C#.

a) Übersetzt bedeutet *package* „Paket“.



Bitte beachten Sie:

Die Sichtbarkeit **package** wird nicht von allen Programmiersprachen unterstützt. C# verwendet ähnliche Entsprechungen, C++ dagegen kennt zum Beispiel nur die Sichtbarkeiten **private**, **protected** und **public**.

In der UML wird die Sichtbarkeit durch Symbole ausgedrückt, die vor dem Namen des Attributs oder der Methode stehen. Das Minuszeichen – steht zum Beispiel für die Sichtbarkeit **private** und das Pluszeichen + für die Sichtbarkeit **public**.

Der Ausdruck

`-kNummer : int`

legt zum Beispiel fest, dass die Sichtbarkeit des Attributs `kNummer` privat ist.

Tab. 6.2: UML-Symbole für die Sichtbarkeiten

Symbol	Sichtbarkeit
–	private/privat
+	public/öffentlich
#	protected/geschützt
~	package/Paket

Bei der Festlegung der Sichtbarkeiten gilt, wie Sie ja bereits wissen:



So privat wie möglich und so öffentlich wie nötig.

Das heißt: Ein Attribut beziehungsweise Feld hat die Sichtbarkeit **private** oder bei Klassen, die vererben, die Sichtbarkeit **protected**.

Methoden dagegen sollten zunächst einmal die Sichtbarkeit **public** erhalten, damit die Kommunikation zwischen den Klassen möglich ist. Danach können Sie dann überprüfen, ob eine Methode nur von der Klasse selbst benutzt wird – also auch die Sichtbarkeit **private** beziehungsweise **protected** haben kann. In unserem Beispiel wäre das der Fall bei den Methoden, die die laufende Nummer für die Kunden und so weiter liefern. Diese Methoden müssen nicht von anderen Klassen aufgerufen werden können. Hier reicht also auch die Sichtbarkeit **private**.

Wenn wir jetzt auch noch die Sichtbarkeiten in unser Klassendiagramm eintragen, sieht es so aus:

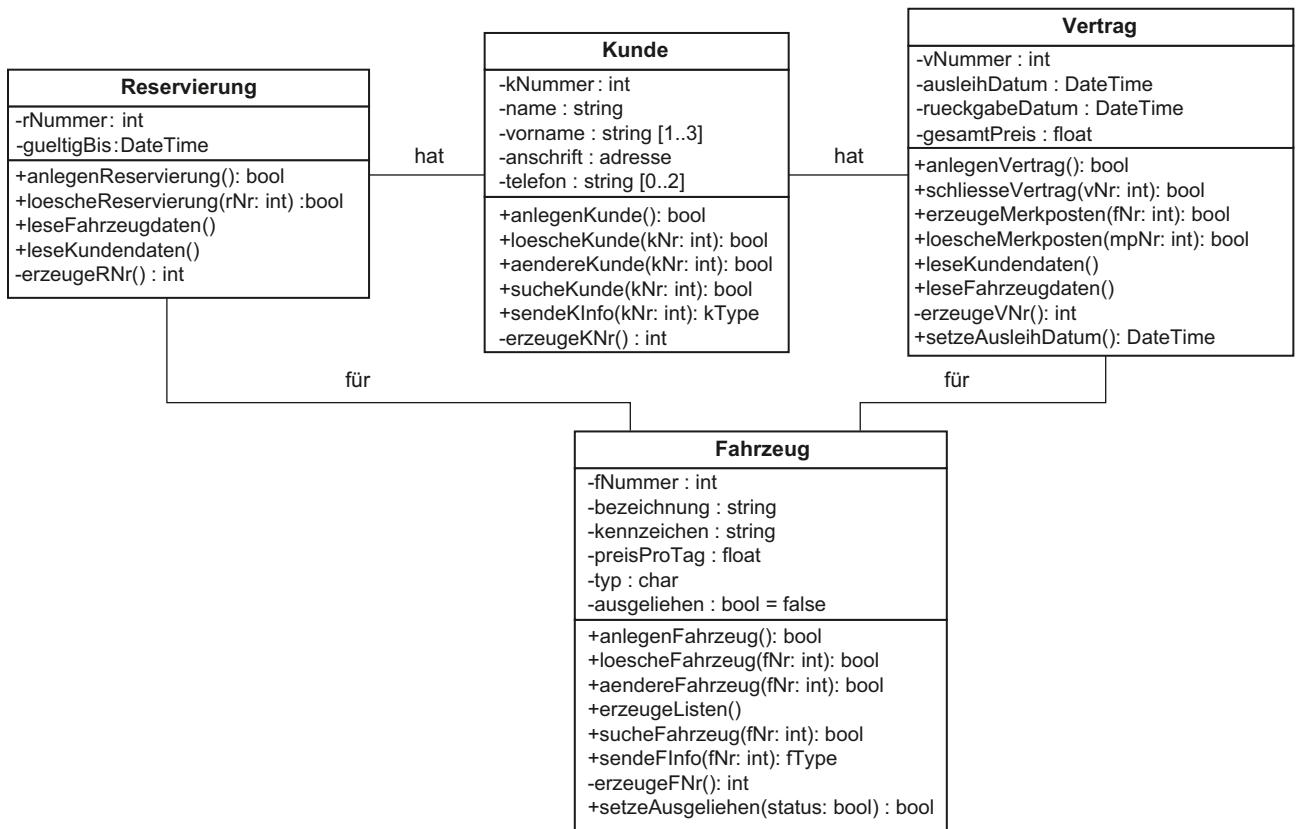


Abb. 6.3: Das Klassendiagramm mit den Sichtbarkeiten

Hinweis:

Viele UML-Werkzeuge setzen die Sichtbarkeiten für Attribute automatisch auf **private** und die Sichtbarkeit von Methoden automatisch auf **public**. Bei einigen Werkzeugen können Sie die Sichtbarkeiten auch nicht ändern.

So viel zu den Sichtbarkeiten. Kommen wir nun zum Präzisieren der Klassenbeziehungen.

6.4 Präzisierung der Klassenbeziehungen

Bisher haben wir in unserem Klassendiagramm nur einfache Assoziationen zwischen den Klassen dargestellt.



Abb. 6.4: Eine einfache Assoziation

Diese Assoziation drückt lediglich aus, dass zwischen den Klassen „Kunde“ und „Vertrag“ irgendeine Art von Beziehung besteht. Nun ist es aber so, dass zum Beispiel zu einem Vertrag genau ein Kunde gehört. Ein und dasselbe Fahrzeug kann ja nicht von mehreren Kunden gleichzeitig ausgeliehen werden. Und ein Vertrag ohne Kunden ergibt keinen Sinn.

Aus Sicht des Kunden dagegen sieht die Beziehung anders aus. Ein Kunde kann durchaus auch mehrere Verträge gleichzeitig haben. Außerdem kann es auch vorkommen, dass ein Kunde überhaupt keinen Vertrag hat – nämlich dann, wenn er keine Fahrzeuge ausgeliehen hat.

Umgangssprachlich lassen sich diese Beziehungen so ausdrücken:

Jeder Vertrag ist genau einem Kunden zugeordnet. Jeder Kunde kann beliebig viele Verträge haben.

Im Klassendiagramm werden solche Beziehungen durch die **Multiplizität** der Assoziation beschrieben. Diese Multiplizität kennen Sie ja bereits von den Attributen.

Bei den Assoziationen wird die Multiplizität an das jeweilige Ende der Linie geschrieben – und zwar unterhalb der Linie. Grafisch dargestellt sieht die Beschreibung der Assoziation zwischen Kunde und Vertrag dann so aus:



Abb. 6.5: Ein Kunde hat beliebig viele Verträge

Damit die Darstellung einfacher zu verstehen ist, können Sie auch noch die **Leserichtung** für die Beziehung angeben. Dazu wird ein kleines Dreieck hinter dem Namen verwendet.



Abb. 6.6: Die Leserichtung für eine Assoziation

Neben den einfachen Assoziationen kann es zwischen Klassen auch besondere Formen der Beziehung geben. Dazu gehören:

- die Generalisierung,
- die Aggregation und
- die Komposition.

Schauen wir uns diese drei Sonderformen der Reihe nach an.

Die **Generalisierung** kennen Sie bereits. Hier werden bei der Vererbung untergeordnete Klassen in einer übergeordneten Klasse verallgemeinert.

Im Klassendiagramm wird die Generalisierung durch einen Pfeil mit einem nicht ausgefüllten Dreieck als Spitze dargestellt. Die Spitze zeigt dabei von der untergeordneten zur übergeordneten Klasse.

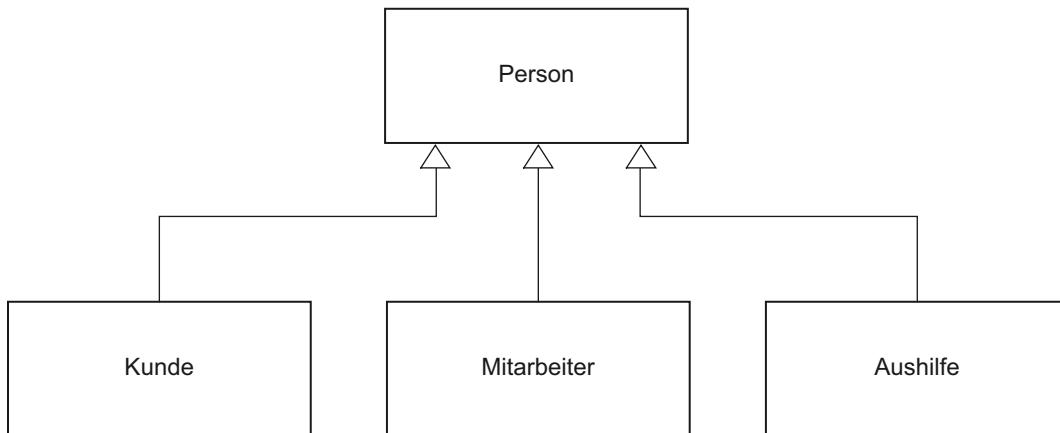


Abb. 6.7: Eine Generalisierung im Klassendiagramm

Hinweis:

Die Richtung des Pfeils kann für etwas Verwirrung sorgen, wenn Sie mit dem Begriff „Vererbung“ arbeiten. Dann zeigt der Pfeil ja eigentlich in die falsche Richtung. Merken Sie sich, dass eine Vererbung als Generalisierung dargestellt wird und die Pfeilspitze immer zur übergeordneten Klasse beziehungsweise zu der Klasse weist, von der geerbt wird.

So viel zur Generalisierung.

Die **Aggregation** haben Sie schon ganz kurz kennengelernt. Sie drückt aus, dass eine Klasse in einer anderen enthalten ist, und wird mit einer nicht ausgefüllten Raute am Ende der Linie dargestellt.

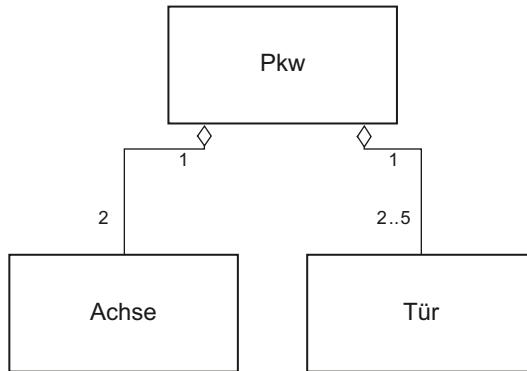


Abb. 6.8: Eine Aggregation im Klassendiagramm

Die Klasse „Pkw“ in der vorigen Abbildung enthält zum Beispiel zwei Achsen und zwei bis fünf Türen.

Die **Komposition** ist eine strengere Form der Aggregation. Sie drückt aus, dass eine Klasse in einer anderen Klasse enthalten ist und auch nur dann existieren kann, wenn diese andere Klasse existiert. So kann ein Zimmer zum Beispiel nur dann vorhanden sein, wenn es auch ein Gebäude gibt.



Merken Sie sich:

Bei einer **Aggregation** kann jede Klasse auch für sich alleine existieren. Bei einer **Komposition** hängt die Existenz der einen von der Existenz der anderen Klasse ab.

Die Komposition wird im Klassendiagramm durch eine ausgefüllte Raute am Ende der Linie dargestellt.

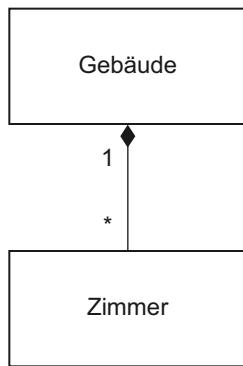


Abb. 6.9: Eine Komposition im Klassendiagramm

So viel zu den verschiedenen Assoziationen in einem Klassendiagramm.

Erweitern wir jetzt das Klassendiagramm unserer Autovermietung ein letztes Mal und geben noch die Multiplizitäten sowie die Leserichtungen an. Über die Multiplizitäten nehmen wir dabei noch einige Einschränkungen vor. So soll ein Kunde zum Beispiel maximal fünf Verträge gleichzeitig haben können und jeder Vertrag darf maximal zehn Fahrzeuge umfassen.

Das Diagramm sieht dann so aus:

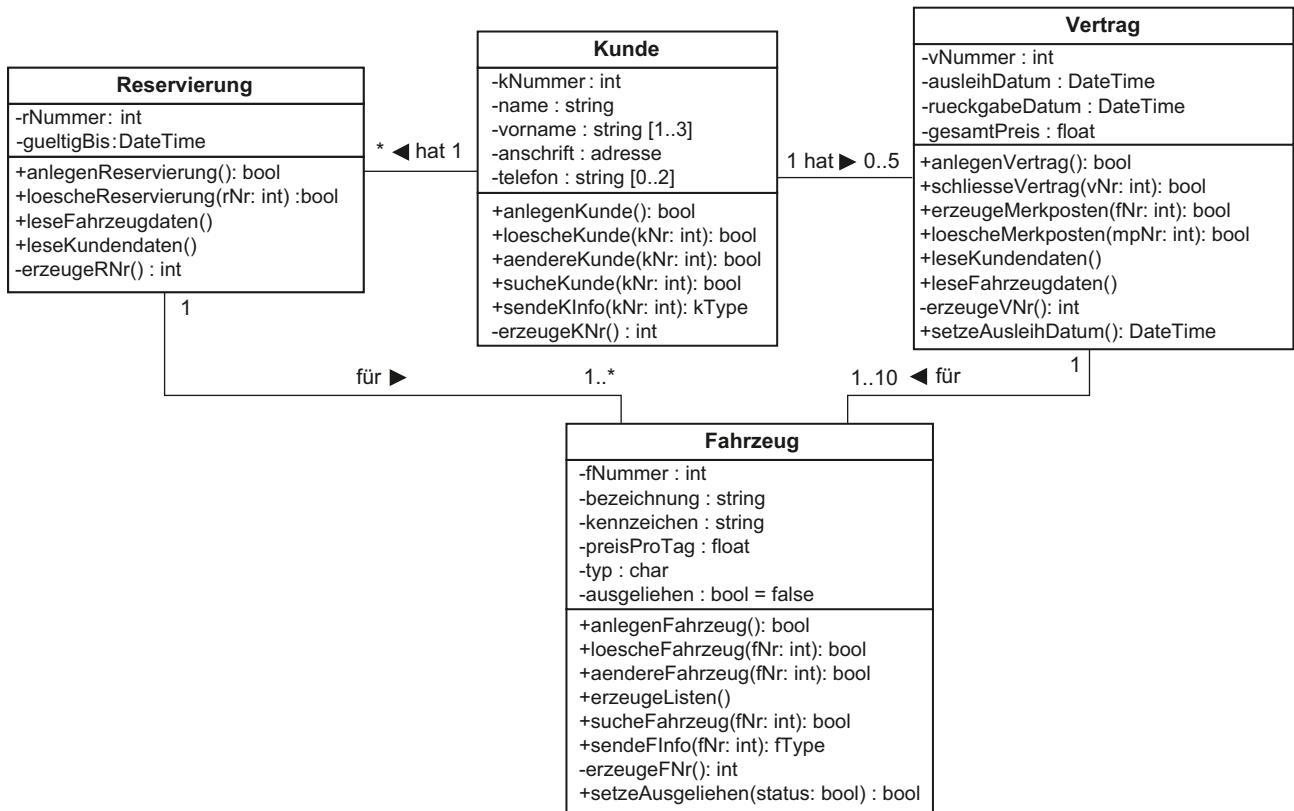


Abb. 6.10: Das Klassendiagramm von Autovermietung 2030 mit Multiplizitäten und Leserichtungen

Schauen wir uns abschließend noch kurz den Einsatz von Entwurfsmustern an.

6.5 Exkurs: Entwurfsmuster

Bei der Software-Entwicklung werden Sie immer wieder auf gleiche oder sehr ähnliche Problemstellungen treffen. So müssen Sie zum Beispiel häufig mehrere Objekte einer Klasse einem Objekt einer anderen Klasse zuordnen oder eine Baugruppe, die aus mehreren Einzelteilen besteht, als Klassen abbilden.

Dabei müssen Sie nun nicht jedes Mal „das Rad neu erfinden“. Sie können einmal umgesetzte Techniken als Entwurfsmuster ablegen und dann später immer wieder einsetzen.

Entwurfsmuster werden im Fachjargon auch **Patterns^{a)}** genannt.



a) *Pattern* bedeutet übersetzt „Muster“.

Schauen wir uns das an einem Beispiel an. Wir wollen ein Entwurfsmuster anlegen, das mehrere Objekte einer Klasse einer anderen Klasse zuordnet – also eine Beziehung, wie sie in einer Leihbücherei zum Beispiel zwischen dem Vertrag und den Medien besteht. In der folgenden Abbildung finden Sie oben ein Objektdiagramm, in der Mitte das Klassendiagramm und unten das abgeleitete Entwurfsmuster.

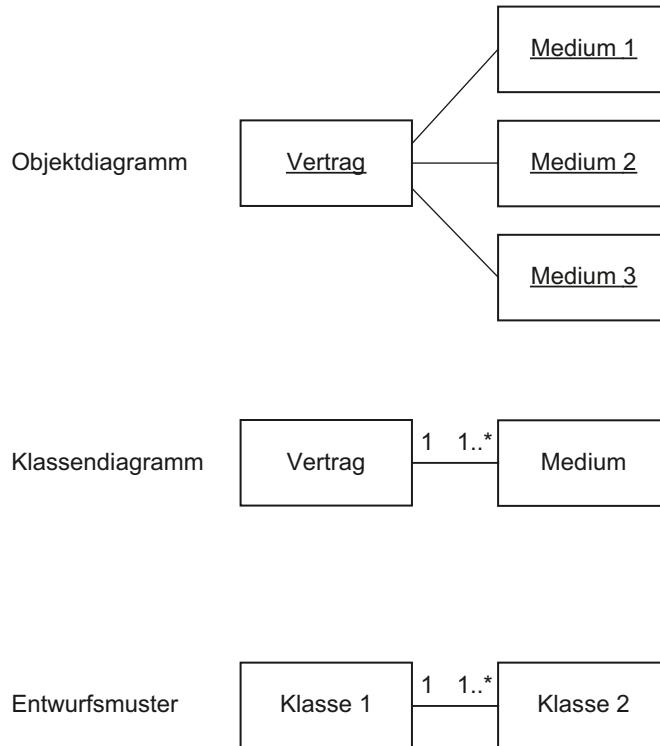


Abb. 6.11: Ein Entwurfsmuster

Solche Entwurfsmuster gibt es auch für zahlreiche andere Standardprobleme bei der Programmierung. Beispiele finden Sie unter anderem bei Gamma, Helm, Johnson und Vlissides.

So viel zur Verfeinerung des Klassendiagramms im Entwurf.

Zusammenfassung

Bei Assoziationen kann die Multiplizität und die Leserichtung angegeben werden.

Neben einfachen Assoziationen gibt es noch die Generalisierung, die Aggregation und die Komposition.

Über Entwurfsmuster können einmal erstellte Lösungsansätze wiederverwendet werden.

Aufgaben zur Selbstüberprüfung

- 6.1 Welche vier Sichtbarkeiten kennt die UML? Beschreiben Sie die Sichtbarkeiten kurz.

- 6.2 Wie wird eine Generalisierung im Klassendiagramm dargestellt?

- 6.3 Was unterscheidet eine Aggregation und eine Komposition?

- 6.4 Wie wird eine Methode `suchen` im Klassendiagramm dargestellt, die ein ganzzahliges Argument `Nr` erhält und öffentlich sichtbar ist?

7 Die Datenhaltung

In diesem Kapitel beschäftigen wir uns mit der Speicherung der Daten in unserer Anwendung Autovermietung 2030.

Grundsätzlich gibt es zwei verschiedene Möglichkeiten, Daten zu speichern:

1. Die Informationen werden unstrukturiert in einer Datei abgelegt.
2. Die Informationen werden strukturiert in einer Datenbank gespeichert.

Die unstrukturierte Speicherung ist sehr flexibel, da beliebige Informationen in nahezu beliebiger Größe gesichert werden können. Allerdings hat sie auch einen gravierenden Nachteil: Der gezielte Zugriff auf einzelne Information in einer Datei ist – wie Sie ja bereits aus eigenen Versuchen wissen – recht umständlich. Wenn wir zum Beispiel nach einem bestimmten Kunden suchen wollen, müssten wir die Datei von Anfang an nach einem passenden Eintrag durchforsten.

Für die Speicherung und Verwaltung von strukturierten Informationen werden daher **Datenbank-Management-Systeme** (DBMS) verwendet. Sie ermöglichen einen schnellen und direkten Zugriff auf Daten.

7.1 Was ist eine Datenbank?

Datenbanken sind das elektronische Gegenstück zu den altbekannten und bewährten Karteien. Genau wie in Karteien werden in Datenbanken Informationen unterschiedlicher Art in strukturierter Form gesammelt und verwaltet – zum Beispiel Adressen, Artikeldaten oder Kundendaten.



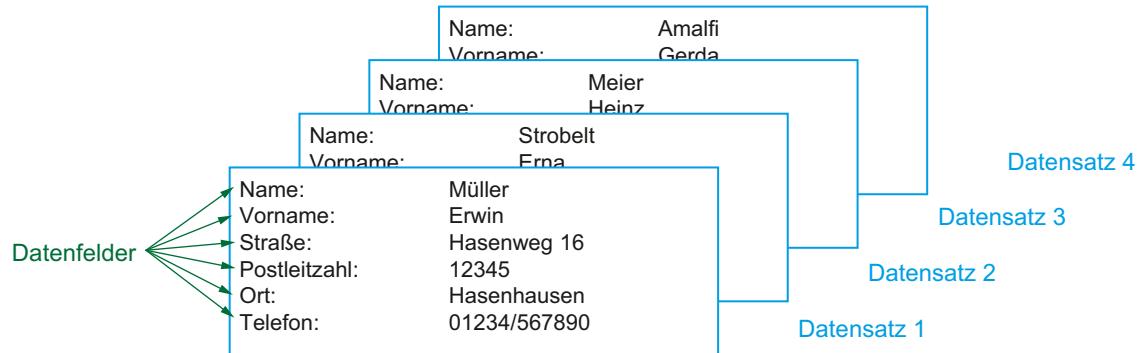
Eine Datenbank ist eine strukturierte Sammlung von Informationen.

Logisch zusammengehörende Informationen, wie eine Adresse oder die Daten zu einem Artikel, werden gemeinsam in einem **Datensatz** gespeichert. Ein Datensatz entspricht damit einer einzelnen Karteikarte.

Ein Datensatz selbst besteht aus **Datenfeldern**. Ein Datenfeld enthält Teilinformationen – zum Beispiel die Postleitzahl oder den Namen bei einer Adresse. Es entspricht damit in etwa einer Zeile auf einer Karteikarte. Die Datenfelder können in der Regel nur Informationen eines bestimmten **Datentyps** speichern – zum Beispiel Texte, ganze Zahlen oder logische Werte. Dieses Konzept kennen Sie ja bereits von den Datentypen des .NET Frameworks. Außerdem hat ein Datenfeld auch besondere **Feldeigenschaften** – zum Beispiel die maximale Länge der enthaltenen Daten.



Eine Datenbank speichert die Daten in Form von Datensätzen. Ein Datensatz wiederum besteht aus Datenfeldern mit bestimmten Eigenschaften.



Das Diagramm zeigt eine Tabelle mit den Spaltennamen Name, Vorname, Straße, PLZ, Ort und Telefon. Es enthält vier Zeilen, die den gleichen Struktur als oben dargestellte Datensätze haben.

	Name	Vorname	Straße	PLZ	Ort	Telefon
Datensatz 1	Müller	Erwin	Hasenweg 37	12345	Hasenstadt	01234/567890
Datensatz 2	Strobelt	Erna	Birnenplatz 1	56789	Bullerbü	0900/110111
Datensatz 3	Meier	Heinz	Gasse 17	98765	Utopia	0001/98765
Datensatz 4	Amalfi	Gerda	Lahnweg 3	20146	Hamburg	040/001001

Abb. 7.1: Datenablage in einem Karteikasten (oben) und in einem Datenbank-Management-System (unten)

Der Aufbau der Datensätze in einer Tabelle ist dabei identisch. Sie können also zum Beispiel nicht für eine bestimmte Adresse zusätzlich ein Datenfeld für die Faxnummer anlegen. Wenn Sie diese Informationen speichern wollen, müssen Sie das Datenfeld für die gesamte Datenbank anlegen und dann bei Einträgen, die keine Faxnummer haben, das Feld leer lassen.

Bitte beachten Sie:

Der Begriff Datenbank bezeichnet eigentlich nur die strukturierte Sammlung der Daten. Diese Sammlung wird vom Datenbank-Managementsystem verwaltet und Anwendungsprogrammen zur Verfügung gestellt. Häufig wird aber für das Datenbank-Management-System der Einfachheit halber der Begriff Datenbank benutzt.



7.2 Das relationale Datenmodell

Für die Abbildung der Datenstrukturen und der Beziehungen der Daten untereinander können verschiedene **Datenmodelle** verwendet werden – zum Beispiel das hierarchische Datenmodell oder das Netzwerkmodell. Ein sehr weitverbreitetes Modell ist das **relationale Datenmodell**, das wir uns jetzt einmal ein wenig genauer ansehen wollen.

Beim relationalen Datenmodell werden die Daten in Tabellenform – den **Relationen** – abgelegt. Die Spalten der Tabelle – im relationalen Datenmodell **Attribut** genannt – beschreiben dabei die Eigenschaften der Daten – also zum Beispiel den Namen, den Vornamen oder die Straße.

Die Zeilen einer Relation – die **Tupel** – enthalten dann alle Attribute, die zu einem Eintrag – zum Beispiel einem bestimmten Namen – gehören. Eine Zeile in einer Tabelle steht damit quasi für einen Datensatz.



Tupel ist ein Begriff aus der Mathematik. Er bezeichnet eine geordnete Zusammenstellung von Objekten.

Attribut

Tupel →

Name	Vorname	Straße	PLZ	Ort	Telefon
Müller	Erwin	Hasenweg 16	12345	Hasenhausen	01234/567890
Strobelt	Erna	Birnenplatz 1	56789	Bullerbü	0900/110111
Meier	Heinz	Gasse 17	98765	Utopia	0001/98765
Amalfi	Gerda	Lahnweg 3	20146	Hamburg	040/001001

Abb. 7.2: Darstellung von Daten in einer Relation

Für die eindeutige Identifikation werden die Zeilen in einer Tabelle häufig mit einem eindeutigen Schlüssel versehen – dem **Primärschlüssel**. Dieser Primärschlüssel kann zum Beispiel eine Kundennummer sein, die für jeden Kunden unterschiedlich ist.



Ein Primärschlüssel identifiziert jeden Datensatz eindeutig.

Primärschlüssel

↓

KNR	Name	Vorname	Straße	PLZ	Ort	Telefon
1001	Müller	Erwin	Hasenweg 16	12345	Hasenhausen	01234/567890
1002	Strobelt	Erna	Birnenplatz 1	56789	Bullerbü	0900/110111
1003	Meier	Heinz	Gasse 17	98765	Utopia	0001/98765
1004	Amalfi	Gerda	Lahnweg 3	20146	Hamburg	040/001001

Abb. 7.3: Die Kundentabelle mit Primärschlüssel

Über den Primärschlüssel einer Tabelle kann diese Tabelle auch mit anderen Tabellen verbunden werden. Das ist zum Beispiel erforderlich, wenn Sie die Verträge eines Kunden aus unserer Autovermietung in einer eigenen Tabelle ablegen.

In der Tabelle mit den Verträgen wird dann neben den eigentlichen Vertragsdaten auch die Kundennummer gespeichert. Damit kann jeder Vertrag eindeutig einem Kunden zugeordnet werden, ohne dass die Daten des Kunden in der Tabelle mit den Verträgen noch einmal vollständig gespeichert werden müssen.

The diagram illustrates the relationship between two tables. The top table, labeled 'Kunden', has columns: KNR, Name, Vorname, Straße, PLZ, Ort, and Telefon. The bottom table, labeled 'Verträge', has columns: VNR, Ausleihdatum, Rueckgabe, and KNR. An arrow points from the 'KNR' column in the top table to the 'KNR' column in the bottom table, indicating that the primary key 'KNR' from the first table serves as a foreign key in the second table.

KNR	Name	Vorname	Straße	PLZ	Ort	Telefon
1001	Müller	Erwin	Hasenweg 16	12345	Hasenhausen	01234/567890
1002	Strobelt	Erna	Birnenplatz 1	56789	Bullerbü	0900/110111
1003	Meier	Heinz	Gasse 17	98765	Utopia	0001/98765
1004	Amalfi	Gerda	Lahnweg 3	20146	Hamburg	040/001001

VNR	Ausleihdatum	Rueckgabe	KNR
0001	10.01.	10.02.	1001
0002	11.02.	11.03.	1004
0003	03.03.	03.04.	1001
0004	10.03.	10.04.	1002

Abb. 7.4: Verbindung von Tabellen über Schlüssel

Bitte beachten Sie, dass der Primärschlüssel aus der Tabelle mit den Kunden in der Tabelle mit den Verträgen nicht mehr als Primärschlüssel dienen kann, da er hier ja auch mehrfach erscheint – zum Beispiel, wenn für einen Kunden mehrere Verträge vorliegen.

Primärschlüssel einer Tabelle, die in einer anderen Tabelle verwendet werden, um eine Beziehung herzustellen, werden **Fremdschlüssel** genannt.



Normalerweise stellt ein Datenbank-Management-System keine Oberfläche für die Eingabe und das Abrufen von Daten zur Verfügung. Es kümmert sich vor allem um die Organisation und die Ablage der Daten. Die eigentliche Arbeit mit den Datenbanktabellen – zum Beispiel das Erfassen neuer Daten oder das Abfragen von gespeicherten Daten – erfolgt über eigene Anwendungsprogramme. Diese Anwendungsprogramme übergeben dem Datenbank-Management-System Befehle, welche Aufgaben es erledigen soll.

Für die Kommunikation zwischen Anwendungsprogrammen und relationalen Datenbank-Management-Systemen hat sich der Sprachstandard **SQL**(Structured Query Language)⁴ etabliert. Über SQL können Sie beispielsweise Daten in einer Datenbank suchen oder auch neue Einträge zu einer Tabelle hinzufügen.

7.3 Die Tabellen für die Autovermietung

Die Tabellen für unsere Autovermietung können wir recht einfach aus den Attributen aus dem Klassendiagramm ableiten. Wir müssen lediglich sicherstellen, dass mehrere Spalten für die Telefonnummern vorhanden sind, und den benutzerdefinierten Datentyp `adresse` in einzelne Spalten auflösen. Die Vornamen eines Kunden dagegen fassen wir in einer Spalte zusammen, da wir sie nicht getrennt verarbeiten werden.

4. *Structured Query Language* bedeutet übersetzt so viel wie „Strukturierte Abfragesprache“.

Als Primärschlüssel benutzen wir die Kundennummer, die Vertragsnummer und so weiter. Über diese Schlüssel stellen wir auch die Verbindung zwischen den verschiedenen Tabellen her.

Grafisch lässt sich die Tabellenstruktur so darstellen:

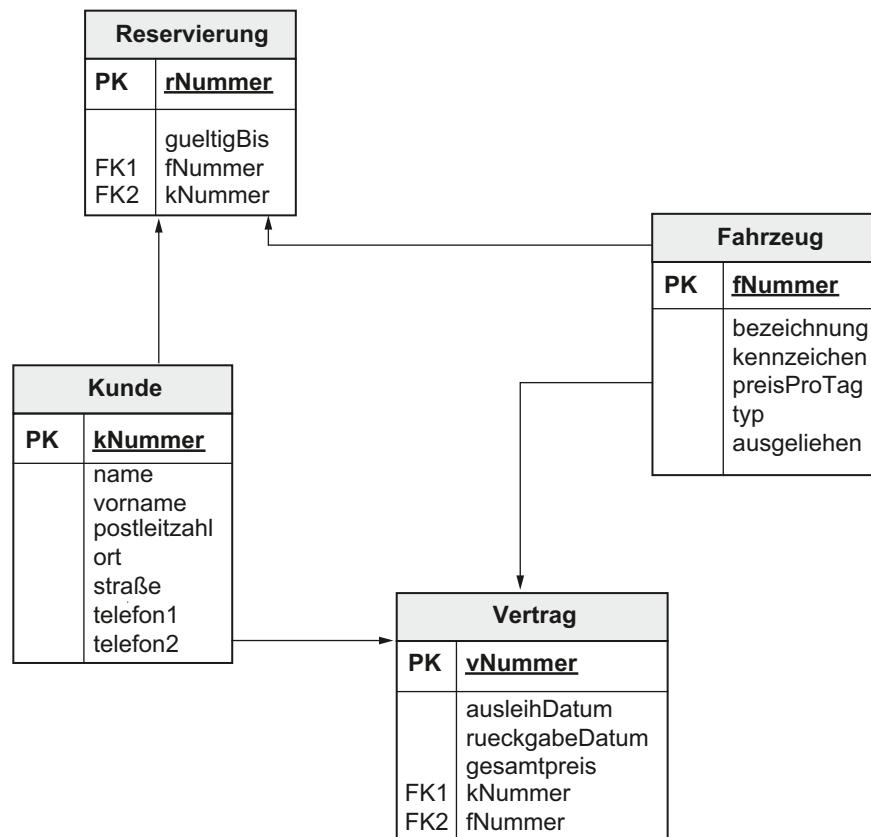


Abb. 7.5: Die Tabellen für die Autovermietung

Die Primärschlüssel der einzelnen Tabellen sind in der Abbildung jeweils fett und unterstrichen dargestellt. Zusätzlich finden Sie links vor dem Namen noch die Buchstaben **PK** als Abkürzung für *Primary Key*⁵.

Die Fremdschlüsselelemente zur Verbindung der verschiedenen Tabellen stehen jeweils unten in den Tabellen. Sie sind durch die Buchstaben **FK** für *Foreign Key*⁶ und eine laufende Nummer gekennzeichnet.



Bitte beachten Sie:

Die Darstellung in der vorigen Abbildung ist **kein** UML-Diagramm. Es handelt sich um ein allgemeines Datenbankmodelldiagramm.

So viel zur Datenhaltung für die Autovermietung. Wir werden uns später bei der praktischen Umsetzung noch einmal ein wenig intensiver mit dem Thema beschäftigen.

Damit haben wir den Entwurf abgeschlossen.

5. *Primary key* ist die englische Bezeichnung für „Primärschlüssel“.

6. *Foreign key* ist die englische Bezeichnung für „Fremdschlüssel“.

Zusammenfassung

Die Speicherung von Daten kann grundsätzlich in Dateiform oder als Datenbank erfolgen.

Ein weitverbreitetes Datenmodell ist das relationale Modell. Es speichert Daten in Tabellenform.

Aufgaben zur Selbstüberprüfung

- 7.1 Wodurch kann ein Datensatz im relationalen Modell eindeutig identifiziert werden?

- 7.2 Wie können mehrere Tabellen im relationalen Modell verbunden werden?

8 Agile Softwareentwicklung

In diesem Kapitel beschäftigen wir uns kurz mit der agilen Software-Entwicklung.

Das Vorgehensmodell, das wir bei der Entwicklung der Software für die Autovermietung eingesetzt haben, ist vergleichsweise aufwendig und erfordert zum Teil auch detaillierte Kenntnisse der eingesetzten Methoden. Abhängig von der Komplexität des zu entwickelnden Systems vergeht sehr viel Zeit, bis überhaupt erste Ergebnisse sichtbar sind. Außerdem müssen umfangreiche Dokumentationen produziert werden, die nicht selten ungelesen in einer Schublade verstauben.

Schauen wir uns dazu ein Beispiel an:

Die Autovermietung hat beschlossen, den Zugriff auf die vorhandenen Fahrzeuge auch über eine App zu ermöglichen. So können Benutzer auch von unterwegs nachsehen, ob ein Fahrzeug aktuell verfügbar ist, und es gegebenenfalls reservieren. Nach dem Vorgehensmodell, das wir für die Entwicklung verwendet haben, müssten jetzt erst wieder ein Lasten- und ein Pflichtenheft erstellt werden. Ausgehend vom Pflichtenheft erfolgen dann der Entwurf und die Analyse und abschließend die Umsetzung. Das kostet nicht nur viel Zeit und Geld, sondern ist auch recht umständlich. Denn einige Funktionen sind ja bereits vorhanden und könnten auch für die App verwendet werden.

Um schneller zum Ziel – zur fertigen Software – zu kommen, haben sich immer mehr **agile Vorgehensmodelle**⁷ etabliert. Dabei werden Praktiken eingesetzt, die sich bei der Software-Entwicklung als bestes Vorgehen – als *Best Practice* – bewährt haben. Auf Formalien wird nach Möglichkeit verzichtet.



Die ersten agilen Vorgehensmodelle wurden auch **leichtgewichtige Prozessmodelle** (engl.: *light-weight processes*) genannt.

Eines dieser Modelle ist das eXtreme Programming (XP)⁸. Es unterscheidet sich vor allem durch folgende Punkte von dem klassischen Vorgehen:

- Die Entwicklung erfolgt in sehr kleinen und damit sehr schnellen Schritten.
Ausgehend von den grundlegenden Basisanforderungen wird das System immer weiter entwickelt. Im Idealfall kann dem Kunden wöchentlich eine neue Version zur Abnahme und Korrektur vorgelegt werden. Die Entwicklung orientiert sich damit konsequent an den Bedürfnissen des Kunden.
- Permanente Veränderungen werden zum Teil der Entwicklung.
Beim klassischen Vorgehen werden Veränderungen während der Entwicklung nach Möglichkeit vermieden. Denn dann müssten ja unter Umständen auch das Lasten- und das Pflichtenheft überarbeitet und Schritte im Prozess wiederholt werden. Beim Extreme Programming werden Veränderungen dagegen als Teil des Prozesses akzeptiert.

7. Agil bedeutet so viel wie flink oder beweglich.

8. Wörtlich übersetzt bedeutet *Extreme Programming* so viel wie „extremes Programmieren“.

tier. Der Kunde hat jederzeit die Möglichkeit, seine Anforderungen an das System neu zu definieren. Damit wird er von dem Zwang befreit, noch vor der eigentlichen Entwicklung festlegen zu müssen, was die Software können muss.

- Die Entwicklung berücksichtigt nur die aktuellen Anforderungen.

Umgesetzt werden nur die Anforderungen, die sich aktuell stellen. Erweiterungen werden erst dann berücksichtigt, wenn sie tatsächlich erforderlich sind. Dadurch soll das System so einfach wie eben möglich bleiben.

- Zentrales Element der Entwicklung ist der Programmcode.

Der Code stellt das Ergebnis der Analyse und des Entwurfs dar. Auf eigene Dokumentationen für diese Bereiche wird weitgehend verzichtet. Der Kunde muss das System nicht an schwer verständlichen abstrakten Modellen prüfen, sondern kann die Software tatsächlich ausführen.

Um auch bei diesem scheinbar unstrukturierten Vorgehen die Qualität der Software sicherzustellen, müssen folgende wichtige Prinzipien des Extreme Programmings berücksichtigt werden:

- Es erfolgt eine ständige Kommunikation.

Nicht nur die Entwickler untereinander stehen im ständigen Austausch, sondern auch die Entwickler und der Kunde. Das heißt, der Kunde beziehungsweise ein Ansprechpartner beim Kunden muss permanent verfügbar sein.

- Die Programmierung erfolgt immer zu zweit.

Ein Programmierer erstellt den Code, der andere sieht ihm dabei über die Schulter und weist seinen Partner auf mögliche Fehler oder Probleme hin. Die beiden Programmierer tauschen dabei regelmäßig ihre Rollen. Durch dieses Vier-Augen-Prinzip sollen Fehler reduziert werden. Außerdem verringert sich durch die Verteilung des Know-hows auf mehrere Personen die Abhängigkeit von einem bestimmten Programmierer.

- Es erfolgen permanente Tests.

Die Tests sind ein zentraler Punkt des Extreme Programmings. Sie werden parallel mit dem Code oder sogar schon vor der eigentlichen Programmierung entworfen. Sobald ein Fehler entdeckt wird, werden sämtliche Tests erneut durchgeführt.

Um die Tests jederzeit nachvollziehen zu können, werden sie so weit wie möglich automatisiert.

Wenn Tests parallel zum Code oder vor der eigentlichen Programmierung erstellt werden, spricht man auch von **testgetriebener Entwicklung** (engl.: *test driven development*).



- Die Arbeitszeiten werden strikt eingehalten.

So soll sichergestellt werden, dass die Entwickler jederzeit motiviert und voll konzentriert bei der Arbeit sind.

Ein weiteres agiles Vorgehensmodell ist **Scrum**⁹. Hier werden mehrere kleinere Durchläufe – **Sprints** genannt – mit maximal 30 Tagen Dauer durchgeführt. Vor jedem Sprint erfolgen mindestens zwei Planungstreffen. Nach einem Sprint werden eine **Review**¹⁰ und eine **Retrospektive** durchgeführt. Bei der Review wird das Ergebnis des Sprints vorgestellt und bewertet. Die Retrospektive sucht unter anderem nach Verbesserungsmöglichkeiten im Prozess.

Das Entwicklungsteam organisiert sich bei Scrum weitgehend selbst. Es gibt allerdings feste Rollen und auch feste Vorgaben – wie zum Beispiel eine kurze tägliche Besprechung.

Es gibt noch einige weitere Modelle wie zum Beispiel Kanban, die wir Ihnen hier aber nicht vorstellen wollen.

Grundsätzlich betonen agile Modelle durch den weitestgehenden Verzicht auf Formalien sehr viel stärker als andere Modelle den kreativen Charakter der Software-Entwicklung. Außerdem kommen sie durch die starke Betonung des Programmcodes beziehungsweise durch die kurzen Zyklen dem Wunsch vieler Programmierer und auch Kunden nach einer möglichst schnellen Umsetzung entgegen.

Allerdings können agile Methoden keine Wunder vollbringen. Ohne Struktur, gute Vorbereitung und Disziplin funktioniert Software-Entwicklung nicht – auch dann nicht, wenn ein sehr flexibles Modell für die Entwicklung eingesetzt wird.

Zusammenfassung

Agile Vorgehensmodelle wie das Extreme Programming oder Scrum wollen den Aufwand bei der Entwicklung reduzieren und möglichst schnell zum Ergebnis kommen.

Es wird weitgehend auf Formalien verzichtet.

Aufgaben zur Selbstüberprüfung

- 8.1 Nennen Sie mindestens drei Punkte, in denen sich das Extreme Programming von klassischen Methoden zur Software-Entwicklung unterscheidet.

- 8.2 Was gehört bei Scrum in jedem Fall zu einem Sprint?

9. Wörtlich übersetzt bedeutet *scrum* so viel wie „Gedränge“.

10. *Review* bedeutet übersetzt so viel wie „Überprüfung“.

Schlussbetrachtung

Herzlichen Glückwunsch!

Sie haben zwei weitere Schritte bei der Umsetzung unserer Software Autovermietung 2030 gemeistert – die objektorientierte Analyse und den objektorientierten Entwurf.

Sie wissen jetzt, wie Sie Produktfunktionen über Anwendungsfall- und Aktivitätsdiagramme darstellen. Außerdem können Sie ein statisches Modell eines Systems über Klassendiagramme abbilden.

Wie Sie in diesem Studienheft gesehen haben, waren einige der erforderlichen Schritte durch die gute Vorarbeit – das Erstellen eines Lastenhefts – gar nicht so schwierig, wie es vielleicht im ersten Moment ausgesehen haben mag.

Möglicherweise haben Sie ja beim Durcharbeiten dieses Studienhefts auch die eine oder andere Schwachstelle bei der Software für die Autovermietung gefunden. So ist zum Beispiel die Verwaltung der Fahrzeuge bewusst sehr einfach gehalten. Spezielle Informationen – wie zum Beispiel technische Details – können wir in unserer Software nicht abbilden. Nehmen Sie ruhig in Eigenregie Erweiterungen vor. Die dazu nötigen Techniken haben Sie in diesem Studienheft gelernt.

Wir haben jetzt alle nötigen Vorarbeiten geleistet, um mit der Umsetzung der Anwendung zu beginnen. Als Nächstes werden wir das Programmgerüst erstellen, Datenbanktabellen anlegen und die ersten Methoden anlegen. Dabei machen wir wieder intensiv von den Möglichkeiten von Visual Studio Gebrauch.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Neben dem Verhalten des Systems muss auch die Struktur des Systems beschrieben werden.
- 1.2 Das Lastenheft beschreibt die Forderungen des Auftraggebers. Das Pflichtenheft beschreibt, wie der Auftragnehmer diese Forderungen umsetzen will.
- 1.3 Das Entwurfsmodell bildet ein System aus Sicht des Entwicklers ab.

Kapitel 2

- 2.1 Die Systemgrenzen sind die Stellen, an denen das System mit der Umwelt kommunizieren muss. Die Darstellung im Anwendungsfalldiagramm erfolgt als Rechteck.
- 2.2 Nein. Es werden nur die Akteure dargestellt, die auch tatsächlich am System den Anwendungsfall ausführen.
- 2.3 Eine **extend**-Beziehung stellt einen Sonderfall eines anderen Anwendungsfalls dar. Der Anwendungsfall wird nur unter bestimmten Bedingungen ausgeführt.

Bei der **include**-Beziehung greift ein Anwendungsfall auf einen anderen Anwendungsfall zu beziehungsweise ein Anwendungsfall ist Teil eines anderen Anwendungsfalls.

- 2.4 Die Darstellung erfolgt mit einem gestrichelten Pfeil und der Kennzeichnung **<<extend>>**. Die Pfeilspitze weist zu dem Anwendungsfall, der erweitert wird.

Kapitel 3

- 3.1 Das konzeptionelle Datenmodell beschreibt die Daten, die ein System verarbeiten soll, und ihre Beziehung untereinander.
- 3.2 Nein, ein Objektdiagramm muss nicht in jedem Fall erstellt werden.
- 3.3 Im ersten groben Klassendiagramm werden nur die Attribute dargestellt.
- 3.4 Es gibt zwei wesentliche Techniken:
 - Dokumentanalyse und
 - Analyse der Produktbeschreibung.
- 3.5 Eine Assoziation ist eine Beziehung zwischen Klassen.
- 3.6 Es handelt sich um ein Objektdiagramm. Bei einem Klassendiagramm würde der Name fett und nicht unterstrichen dargestellt.

Kapitel 4

- 4.1 Im Aktivitätsdiagramm kann auch die zeitliche Reihenfolge dargestellt werden.
- 4.2 Eine Verzweigung, die von einer Bedingung abhängt, wird durch eine Raute dargestellt. An den Verbindungslien wird in eckigen Klammern die Bedingung angegeben.
- 4.3 Eine Teilung wird *fork* oder *splitting* genannt. Eine Verbindung heißt im Fachjargon Synchronisation oder *join*.

Kapitel 5

- 5.1 Über einen Vorgabewert kann ein Attribut mit einem Wert vorbelegt werden. Dieser Wert wird automatisch gesetzt, wenn eine neue Instanz der Klasse erzeugt wird.
- 5.2 Die Multiplizität beschreibt, wie oft ein Attribut vorkommen kann beziehungsweise vorkommen muss.
- 5.3 Hinter dem Namen einer Methode werden üblicherweise runde Klammern () angegeben.

Kapitel 6

- 6.1 Die UML kennt folgende Sichtbarkeiten:
 - **private**,
 - **protected**,
 - **public** und
 - **package**.

Bei der Sichtbarkeit **private** kann nur die Klasse selbst auf die Attribute und Methoden zugreifen.

Bei der Sichtbarkeit **protected** können nur die Klasse selbst und abgeleitete Klassen auf die Attribute und Methoden zugreifen.

Bei der Sichtbarkeit **public** können alle Klassen auf die Attribute und Methoden zugreifen.

Bei der Sichtbarkeit **package** können nur Klassen aus demselben Paket auf die Attribute und Methoden zugreifen.

- 6.2 Eine Generalisierung wird durch einen Pfeil mit einem nicht ausgefüllten Dreieck als Spitze dargestellt. Der Pfeil weist dabei auf die übergeordnete Klasse.
- 6.3 Bei der Aggregation kann eine Klasse auch alleine existieren. Bei der Komposition hängt die Existenz der einen Klasse von der anderen Klasse ab.
- 6.4 Die Darstellung im Klassendiagramm könnte zum Beispiel so aussehen:

```
+suchen(Nr : int)
```

Kapitel 7

- 7.1 Ein Datensatz im relationalen Modell kann durch einen Primärschlüssel eindeutig identifiziert werden.
- 7.2 Der Primärschlüssel aus einer Tabelle wird in der anderen Tabelle als Fremdschlüssel benutzt.

Kapitel 8

- 8.1 Das Extreme Programming unterscheidet sich in den folgenden Punkten von klassischen Methoden zur Software-Entwicklung:
 - Die Entwicklung erfolgt in sehr kleinen und damit sehr schnellen Schritten.
 - Permanente Veränderungen werden zum Teil der Entwicklung.
 - Die Entwicklung berücksichtigt nur die aktuellen Anforderungen.
 - Zentrales Element der Entwicklung ist der Programmcode.

Für die richtige Antwort reicht es aus, wenn Sie drei dieser Punkte nennen.

- 8.2 Zu einem Sprint gehören in jedem Fall eine Review und eine Retrospektive.

B. Glossar

Agile Vorgehensmodelle	Agile Vorgehensmodelle versuchen bei der Software-Entwicklung, so schnell wie möglich zum Ziel zu kommen – dem fertigen Produkt. Dazu werden bewährte Verfahren aus der Software-Entwicklung eingesetzt und Formalien möglichst weit zurückgedrängt.
Akteur	Der Begriff Akteur beschreibt allgemein, wer oder was aktiv und direkt auf die Arbeit des Systems Einfluss nimmt. Ein Akteur kann eine Person beziehungsweise eine Rolle sein, aber auch ein anderes System.
Aktion	Eine Aktion ist Teil eines Aktivitätsdiagramms.
Aktivitätsdiagramm	Das Aktivitätsdiagramm wird für die Darstellung von Aktivitäten in einem Prozess benutzt. Eine Aktivität wird in Aktionen unterteilt. Die einzelnen Aktionen – dargestellt durch Rechtecke mit gerundeten Ecken – werden dabei durch Pfeile verbunden. So lässt sich auch die Reihenfolge der Aktionen ablesen.
Analysemodell	Das Analysemodell liefert die Antwort auf die Frage: „Was soll die Software machen?“
Anwendungsfalldiagramm	Das Anwendungsfalldiagramm stellt Beziehungen zwischen Akteuren und Prozessen – den Anwendungsfällen – dar. Die Akteure werden dabei als „Strichmännchen“ abgebildet und die Anwendungsfälle als Ellipsen.
Assoziation	Eine Assoziation ist – allgemein ausgedrückt – ein Zusammenschluss oder eine Vereinigung. In einem Klassendiagramm stellt eine Assoziation eine Beziehung zwischen den Klassen dar.
Attribut	Attribute beschreiben den Zustand eines Objekts. Bei C# werden die Attribute Felder genannt.
Attributwert	Der Attributwert ist der Wert, den ein Attribut aktuell hat.
Datenbank-Management-System	Ein Datenbank-Management-System übernimmt die Verwaltung von Datenbanken und stellt Anwendungsprogramme für die Verwaltung zur Verfügung.
Datenfeld	Ein Datenfeld speichert Teileinformationen eines Datensatzes. Jedes Datenfeld hat bestimmte Eigenschaften – zum Beispiel die Länge oder den Datentyp.

Datenkapselung	Die Datenkapselung ist ein Prinzip der objektorientierten Programmierung.
Datensatz	Eine Klasse sollte nie direkt auf die Attribute einer anderen Klasse zugreifen, sondern immer nur auf die Methoden. Die Methoden ändern dann die Attribute.
Datentyp	Ein Datensatz speichert logisch zusammengehörige Informationen in einer Datenbank beziehungsweise einer Datenbanktabelle.
DBMS	Der Datentyp beschreibt, welche Informationen mit einem Datenobjekt verarbeitet werden. C# kennt unter anderem verschiedene Datentypen für ganze Zahlen, Gleitkommazahlen und Zeichen. Zusätzlich können beliebige Datentypen durch den Programmierer definiert werden.
Dokumentanalyse	Abkürzung für Datenbankmanagementsystem .
Domäne	Die Dokumentanalyse ist eine Technik zur Ermittlung von Klassenkandidaten. Dazu werden Formulare oder andere Dokumente aus dem Einsatzgebiet des Systems auf mögliche Klassenkandidaten geprüft.
Domänenmodell	Eine Domäne ist – allgemein ausgedrückt – ein eindeutig abgegrenzter Bereich.
Dynamisches Modell (UML)	Domänenmodell ist ein anderer Ausdruck für das konzeptionelle Datenmodell. Im dynamischen Modell der UML wird vor allem das Verhalten eines Systems beschrieben. Dazu werden unter anderem folgende Diagramme verwendet:
Entwurfsmodell	<ul style="list-style-type: none"> • das Anwendungsfalldiagramm – auch <i>Use-Case-Diagramm</i> genannt, • das Aktivitätsdiagramm, • das Zustandsdiagramm und • das Sequenzdiagramm.
Extreme Programming (XP)	Das Entwurfsmodell liefert die Antwort auf die Frage: „Wie soll die Software etwas machen?“ Das Entwurfsmodell setzt ein Analysemodell voraus.
Fremdschlüssel	Das Extreme Programming (XP) ist ein agiles Vorgehensmodell für die Software-Entwicklung.
	Ein Fremdschlüssel ist ein Primärschlüssel einer Datenbanktabelle, der in einer anderen Tabelle zum Herstellen einer Beziehung zwischen den Tabellen genutzt wird.

Klasse	In einer Klasse werden Objekte mit ähnlichen Eigenschaften und ähnlichem Verhalten zusammengefasst. Klassen stehen in einem hierarchischen Verhältnis zueinander. Eine Klasse bildet den Bauplan für ein Objekt.
Klassendiagramm	Im Klassendiagramm werden Klassen und ihre Beziehungen untereinander beschrieben. Die Klassen werden dabei durch ein Rechteck dargestellt. Die Beziehungen werden durch verschiedene Linien dargestellt.
Konzeptionelles Datenmodell	Das konzeptionelle Datenmodell beschreibt die Daten, die ein System verarbeiten soll, und ihre Beziehung untereinander. Wie diese Daten tatsächlich gespeichert werden, spielt im konzeptionellen Datenmodell keine Rolle.
Konzeptuelles Datenmodell	Siehe konzeptionelles Datenmodell.
Lastenheft	Das Lastenheft beschreibt die Anforderungen an eine Software aus Sicht des Kunden.
Leichtgewichtiges Prozessmodell	Leichtgewichtiges Prozessmodell ist eine andere Bezeichnung für die ersten agilen Vorgehensmodelle in der Software-Entwicklung.
Multiplizität	Die Multiplizität beschreibt, wie oft ein Attribut vorkommen kann beziehungsweise vorkommen muss.
Objekt	Ein Objekt in der objektorientierten Programmierung ist jede in sich geschlossene Einheit. Ein Objekt besteht aus dem Zustand und dem Verhalten.
Objektdiagramm	Im Objektdiagramm werden Objekte und ihre Beziehungen untereinander beschrieben. Die Objekte werden dabei durch ein Rechteck dargestellt.
Pflichtenheft	Das Pflichtenheft beschreibt, wie ein Auftragnehmer die Anforderungen aus dem Lastenheft realisieren will.
Primärschlüssel	Über den Primärschlüssel kann ein Datensatz in einer Datenbanktabelle eindeutig identifiziert werden.
Prototyp	Ein Prototyp ist – allgemein ausgedrückt – eine erste Ausprägung eines Produkts, die wesentliche Eigenschaften des späteren Endprodukts darstellt.
Relation	Eine Relation ist im relationalen Datenmodell eine Tabelle.

Relationales Datenmodell	Beim relationalen Datenmodell werden Daten in Tabellen abgelegt, die miteinander verbunden werden. Die Spalten der Tabelle – im relationalen Datenmodell Attribut genannt – beschreiben dabei die Eigenschaften der Daten.
Scrum	Scrum ist ein agiles Vorgehensmodell für die Software-Entwicklung. Die Entwicklung erfolgt in mehreren Iterationen – den Sprints.
Sprint	Ein Sprint ist eine Wiederholung bei Scrum.
SQL	SQL steht für <i>Structured Query Language</i> . SQL ist ein Sprachstandard für die Kommunikation zwischen Anwendungsprogrammen und relationalen Datenbankmanagementsystemen.
Statisches Modell (UML)	Im statischen Modell der UML wird vor allem die Struktur eines Systems dargestellt. Dazu werden unter anderem folgende Diagramme verwendet: <ul style="list-style-type: none">• Klassendiagramm,• Komponentendiagramm,• Paketdiagramm und• Verteilungsdiagramm.
Steuerelement	Die Steuerelemente von Visual Studio Community 2015 stellen vorgefertigte Bedienelemente, Objekte und Methoden zur Verfügung.
Structured Query Language	Siehe SQL.
Transaktion	Eine Transaktion fasst oft mehrere Einzelschritte zu einem einzigen logischen Schritt zusammen.
Tupel	Ein Tupel ist im relationalen Datenmodell eine Zeile in einer Tabelle.
UML	UML steht für <i>Unified Modeling Language</i> . UML ist eine objektorientierte Standardmodellierungssprache zur Beschreibung der Struktur und des Verhaltens von Objekten in Anwendungsbereichen und Datenverarbeitungssystemen.
Unified Modeling Language	Siehe UML.
XP	XP ist die Abkürzung für Extreme Programming.

C. Literaturverzeichnis

Verwendete Literatur

- Balzert, H. (2004). *Lehrbuch Grundlagen der Informatik*. Konzepte und Notationen in UML 2, Java 5, C# und C++, Algorithmik und Software-Technik. Anwendungen. 2. Aufl., Heidelberg: Spektrum Akademischer Verlag.
- Balzert, H. (2008). *Lehrbuch der Softwaretechnik*. Softwaremanagement. 2. Aufl., Heidelberg: Spektrum Akademischer Verlag.
- Balzert, H. (2009). *Lehrbuch der Softwaretechnik*. Basiskonzepte und Requirements Engineering. 3. Aufl., Heidelberg: Spektrum Akademischer Verlag.

Empfohlene Literatur

- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2014). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Wachtendonk: mitp.
- Krypczyk, V. & Bochkor, O. (2018). *Handbuch für Softwareentwickler: Das Standardwerk zu professionellem Software Engineering*. Bonn: Rheinwerk Computing.
- Oestereich, B. & Scheithauer, A. (2013). *Analyse und Design mit der UML 2.5*. Objektorientierte Softwareentwicklung. 11. Aufl., Berlin: Oldenbourg.

D. Abbildungsverzeichnis

Abb. 1.1	Analyse- und Entwurfsmodell	3
Abb. 1.2	Von der Realität zum Pflichtenheft.....	5
Abb. 2.1	Die Systemgrenzen	9
Abb. 2.2	Ein erster Entwurf der Oberflächen	10
Abb. 2.3	Ein Akteur (links) und ein Anwendungsfall (rechts)	11
Abb. 2.4	Das erste Anwendungsfalldiagramm für Autovermietung 2030	12
Abb. 2.5	Das verfeinerte Anwendungsfalldiagramm für „Fahrzeug vermieten“	13
Abb. 2.6	Das verfeinerte Anwendungsfalldiagramm für „Fahrzeug zurückgeben“	14
Abb. 3.1	Von der Realität zum Klassendiagramm	19
Abb. 3.2	Ein analysierter Mietvertrag	21
Abb. 3.3	Das erste Klassendiagramm für Autovermietung 2030	25
Abb. 3.4	Darstellung eines Objekts	26
Abb. 3.5	Ein Objektdiagramm	26
Abb. 4.1	Darstellung einer Aktion	28
Abb. 4.2	Verbindung von Aktionen durch Linien	28
Abb. 4.3	Anfang und Ende in einem Aktivitätsdiagramm	29
Abb. 4.4	Einfaches Aktivitätsdiagramm für den Anwendungsfall „Fahrzeug zurückgeben“	29
Abb. 4.5	Eine Entscheidung in einem Aktivitätsdiagramm.....	29
Abb. 4.6	Das Aktivitätsdiagramm für den Anwendungsfall „Fahrzeug zurückgeben“ mit einer Verzweigung	30
Abb. 4.7	Teilung (links) und Vereinigung (rechts) in einem Aktivitätsdiagramm	30
Abb. 5.1	Das Klassendiagramm für Autovermietung 2030 mit den Attributtypen	35
Abb. 5.2	Das Klassendiagramm für Autovermietung 2030 mit Vorgabewerten und Multiplizitäten	37
Abb. 5.3	Das Klassendiagramm mit den Methoden	40
Abb. 6.1	Das Klassendiagramm mit den präzisierten Methoden	44
Abb. 6.2	Das Klassendiagramm mit den neuen Methoden.....	45
Abb. 6.3	Das Klassendiagramm mit den Sichtbarkeiten	47
Abb. 6.4	Eine einfache Assoziation	48
Abb. 6.5	Ein Kunde hat beliebig viele Verträge	48
Abb. 6.6	Die Leserichtung für eine Assoziation	48
Abb. 6.7	Eine Generalisierung im Klassendiagramm	49

Abb. 6.8	Eine Aggregation im Klassendiagramm	50
Abb. 6.9	Eine Komposition im Klassendiagramm.....	50
Abb. 6.10	Das Klassendiagramm von Autovermietung 2030 mit Multiplizitäten und Leserichtungen	51
Abb. 6.11	Ein Entwurfsmuster	52
Abb. 7.1	Datenablage in einem Karteikasten (oben) und in einem Datenbank-Management-System (unten).....	55
Abb. 7.2	Darstellung von Daten in einer Relation	56
Abb. 7.3	Die Kundentabelle mit Primärschlüssel	56
Abb. 7.4	Verbindung von Tabellen über Schlüssel	57
Abb. 7.5	Die Tabellen für die Autovermietung	58
Abb. G.1	Ein Anwendungsfalldiagramm mit Fehlern	77
Abb. G.2	Klassendiagramm	78
Abb. G.3	Seminaranmeldebogen	79

E. Tabellenverzeichnis

Tab. 2.1	Beschreibung für den Anwendungsfall „Fahrzeug zurückgeben“	15
Tab. 3.1	Die Attribute für die Klassen aus der Dokumentanalyse	22
Tab. 3.2	Die ergänzten Attribute für die Klassen	23
Tab. 3.3	Die Attribute für die Klassen	23
Tab. 4.1	Ablaufbeschreibung für „Fahrzeug zurückgeben“	31
Tab. 4.2	Ablaufbeschreibung für „Fahrzeug zurückgeben“ mit der Ausnahme „Fahrzeug ist beschädigt oder stark verschmutzt“	31
Tab. 5.1	Die Attributtypen für die Klassen unserer Autovermietung	34
Tab. 5.2	Beispiele für die Multiplizität	36
Tab. 5.3	Eine erste Zuordnung der Methoden für die Klassen	38
Tab. 5.4	Die zusätzlichen Methoden für die Klassen	38
Tab. 5.5	Weitere Methoden für die Klassen	39
Tab. 6.1	Sichtbarkeiten der UML	46
Tab. 6.2	UML-Symbole für die Sichtbarkeiten	46
Tab. G.1	Ablaufbeschreibung für „Fahrzeug reservieren“	78

F. Sachwortverzeichnis

A

Aggregation	49
Akteur	11
Aktion	28
Aktivität	
Beschreibung	31
Aktivitätsdiagramm	28
Analyse	
Aufgaben und Ergebnisse der	3
Analysemodell	3
Antwortzeit	5
Anwendungsfall	11
Beschreibung	15
Kategorisierung	16
Anwendungsfalldiagramm	10
Verfeinerung	13
Argument	42
Assoziation	21
Attribut	55
ermitteln	22
Attributtyp	
festlegen	33

B

Benutzeroberfläche	
erster grober Entwurf	9
Beziehung	
ermitteln	21
extend-	14
include-	13

D

Datenbank	54
Datenbank-Management-System	54
Datenfeld	54
Datenhaltung	54
Datenkapselung	39
Datenmodell	55
konzeptionelles	18
konzeptuelles	18
relationales	55

Datensatz	54
-----------------	----

Datentyp	54
----------------	----

DBMS	54
------------	----

default value	35
---------------------	----

Design	1
--------------	---

Dokumentanalyse	20
-----------------------	----

Domänenmodell	18
---------------------	----

E

Entwicklung	
testgetriebene	61
Entwurfsmodell	3, 5
Entwurfsmuster	51

F

Feldeigenschaft	54
fork	30
Fremdschlüssel	57

G

Generalisierung	49
-----------------------	----

J

join	30
------------	----

K

Klassenbeziehung	
Präzisierung der	48
Klassendiagramm	23
erstellen	23
Verfeinerung des	42
Komposition	50

L

Leserichtung	48
--------------------	----

M

Methode	
ermitteln	37
im Klassendiagramm	39
Präzisierung der	42

Modell	
dynamisches	3
erstes grobes dynamisches	7
erstes grobes statisches	18
statisches	3
Verfeinerung des dynamischen ...	28
Verfeinerung des statischen	33
Multiplizität	36, 48
 O	
Objekt- und Klassenkandidat	
ermitteln	20
Objektdiagramm	25
 P	
Parameter	42
Pattern	51
Pflichtenheft	4
Primärschlüssel	56
Produktfunktion	
wesentliche ermitteln	7
Programming	
eXtreme	60
Prozessmodell	
leichtgewichtiges	60
 R	
Relation	55
Retrospektive	62
Review	62
Rückgabetyp	42
 S	
Scrum	62
Sichtbarkeit	45
der UML	46
Festlegen der	45
Software-Architektur	6
splitting	30
Sprint	62
SQL	57
Synchronisation	30
 System	
redundantes	5
Systemgrenze	9
Ermittlung	8
 T	
Tabelle	
für die Autovermietung	57
Teilung	30
Tupel	55
 U	
Use-Case-Diagramm	10
 V	
Vereinigung	30
Verzweigung	29
Vorgabewert	35
Vorgehensmodell	
agiles	60

G. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH20D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

- Das folgende allgemeine Anwendungsfalldiagramm enthält zwei grobe Darstellungsfehler. Beschreiben Sie die Fehler und zeichnen Sie das Diagramm korrekt.

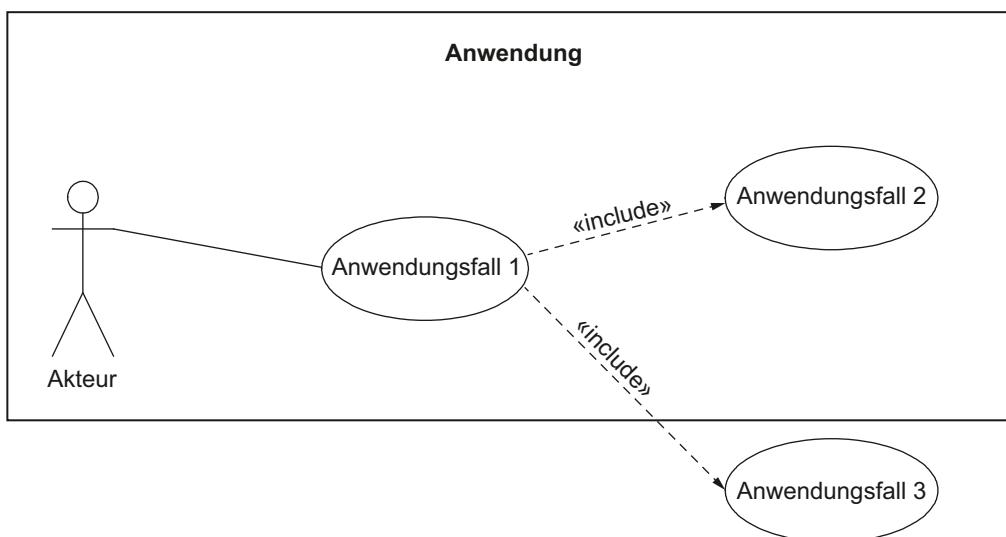


Abb. G.1: Ein Anwendungsfalldiagramm mit Fehlern

10 Pkt.

2. Was drückt das folgende Klassendiagramm aus? Welche Beziehungen bestehen zwischen den Klassen „Gebäude“ und „Zimmer“ beziehungsweise zwischen den Klassen „Gebäude“ und „Wand“?

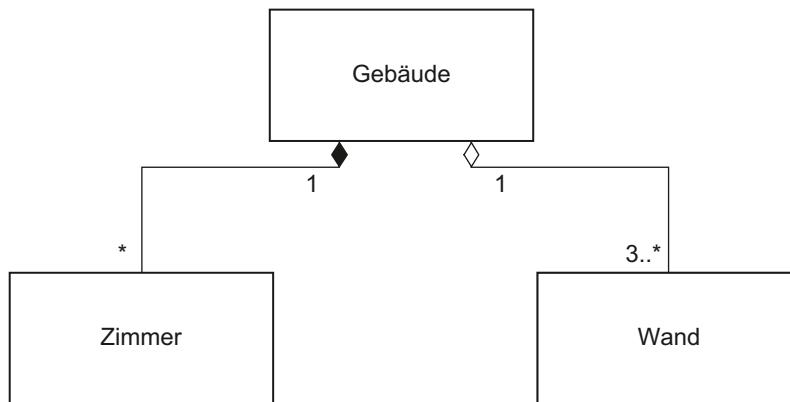


Abb. G.2: Klassendiagramm

10 Pkt.

3. Welche Informationen liefert Ihnen die folgende Attributbeschreibung?

`kWert : int [1..*] = 5`

10 Pkt.

4. Welche Informationen liefert Ihnen die folgende Beschreibung einer Methode aus einem Klassendiagramm?

`+sucheTreffer(fNr: int = 0, mNr: int): bool`

10 Pkt.

5. Die folgende Ablaufbeschreibung soll den Vorgang für die Reservierung eines Autos in einer Autovermietung darstellen. Die Beschreibung enthält einen groben logischen Fehler und eine nicht sonderlich sinnvolle Funktion. Finden Sie diese Stellen, und nehmen Sie Korrekturen vor.

Tab. G.1: Ablaufbeschreibung für „Fahrzeug reservieren“

Aktion	Systemantwort
1. Start	2. Die Kundennummer wird ermittelt.
3. Die Kundennummer wird geprüft.	4. Der Kunde wird gesucht und der Reservierungszeitraum wird vorgeschlagen.
5. Der Reservierungszeitraum wird geprüft und gegebenenfalls korrigiert.	6. Die gewünschte Fahrzeugkategorie wird erfragt.
7. Die gewünschte Fahrzeugkategorie wird eingegeben.	8. Die verfügbaren Fahrzeuge der gewünschten Fahrzeugkategorie und des Reservierungszeitraums werden ermittelt.

15 Pkt.

6. Führen Sie für den folgenden Seminaranmeldebogen eine Dokumentanalyse durch. Erstellen Sie im ersten Schritt eine Tabelle mit den Klassen und den zugeordneten Attributen. Bilden Sie dann die Klassen mit den Attributen als Klassendiagramm ab. Geben Sie bei den Attributen bitte neben einem Namen auch noch den Attributtyp an.

Seminaranmeldung

Teilnehmer				
Name, Vorname: _____				
Anschrift: _____ _____				
Telefon: _____				
Ich melde mich für folgende Seminare an:				
Nummer	Titel	vom	bis	
1	_____	_____	_____	
2	_____	_____	_____	
3	_____	_____	_____	
4	_____	_____	_____	
Abweichender Rechnungsempfänger				
Bitte senden Sie die Rechnung an:				
Firma:	_____			
Name, Vorname:	_____			
Anschrift:	_____			
Telefon:	_____			

Abb. G.3: Seminaranmeldebogen

25 Pkt.

7. Erstellen Sie ein Aktivitätsdiagramm, das folgende Aktivitäten abbildet:
- Eine Zahl soll eingelesen werden.
 - Es wird geprüft, ob die eingegebene Zahl kleiner, gleich oder größer als 100 ist.
 - Für jeden der drei Fälle soll eine eigene Meldung erscheinen, die das Ergebnis der Prüfung ausgibt – also zum Beispiel „Die Zahl ist kleiner als 100“.
 - Am Ende soll der Text „Auf Wiedersehen“ ausgegeben werden. Diese Ausgabe soll unabhängig vom Wert der Zahl erfolgen.

Denken Sie bei der Darstellung bitte auch an den Start und das Ende. **15 Pkt.**

8. Sie wollen eine Postleitzahl für Deutschland als Attribut einer Klasse darstellen. Welchen Attributtyp verwenden Sie? Denken Sie bitte daran, dass deutsche Postleitzahlen führende Nullen haben können.

5 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt:
Einführung in die Datenbankprogrammierung

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1119N01

CSHP21D

Objektorientierte Software-Entwicklung mit C#

**Beispielprojekt:
Einführung in die Datenbankprogrammierung**

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Einführung in die Datenbankprogrammierung

Inhaltsverzeichnis

Einleitung	1
1 Vorüberlegungen und Vorbereitungen	3
1.1 Vorüberlegungen	3
1.2 Vorbereitungen	5
2 Erstellen der Datenbank	8
Zusammenfassung	17
3 Das Zusammenspiel zwischen Datenbank und Anwendung	19
3.1 Ein wenig Theorie	19
3.2 Das Anlegen der Datenquelle	20
3.3 Das Steuerelement DataGridView	24
Zusammenfassung	30
4 Die Dateneingabe	32
4.1 Festlegen der maximalen Eingabelänge	32
4.2 Auffangen von leeren Eingaben	34
4.3 Überprüfung der Postleitzahl	37
4.4 Das Speichern	39
4.5 Ein wenig Feinschliff	40
Zusammenfassung	44
5 Das Formular zur Einzelanzeige	47
Zusammenfassung	49
6 Die Tabelle und die Formulare für die Fahrzeuge	50
6.1 Das Anlegen der Tabelle.....	50
6.2 Die Formulare	52
6.3 Eine Fahrzeugliste	53
Schlussbetrachtung	56

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	57
B.	Glossar	59
C.	Literaturverzeichnis	61
D.	Abbildungsverzeichnis	62
E.	Tabellenverzeichnis	64
F.	Codeverzeichnis	65
G.	Sachwortverzeichnis	66
H.	Einsendeaufgabe	67

Einleitung

Bisher haben wir uns vor allem theoretisch mit unserer Beispielanwendung Autovermietung 2030 beschäftigt. In diesem Studienheft werden wir mit der praktischen Umsetzung mit Visual Studio beginnen. Dabei benutzen wir das Datenbank-Management-System SQL Server, das zum Lieferumfang von Visual Studio gehört.

In diesem Studienheft werden wir uns zunächst einmal um die Grundlagen für das Programm kümmern. Wir erstellen die Datenbank und legen dort einige Tabellen an. Danach sorgen wir dann dafür, dass die Daten für diese Tabellen über verschiedene Steuerelemente in einem Formular erfasst, bearbeitet und angezeigt werden können.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie mit Visual Studio eine Datenbank für den SQL Server anlegen,
- wie Sie Tabellen in einer Datenbank erstellen,
- wie das Datenbank-Management-System SQL Server und Anwendungsprogramme von Visual Studio miteinander kommunizieren,
- wie Sie eine Datenquelle anlegen,
- wie Sie aus einer Datenquelle über das Steuerelement **DataGridView**¹ ein Formular mit einer Tabellenansicht für die Daten erstellen lassen,
- wie Sie Eingabeprüfungen vornehmen,
- wie Sie Fehlerhinweise in einer Zelle des Steuerelements **DataGridView** anzeigen,
- wie Sie Fehler beim Speichern der Daten abfangen,
- wie Sie die Anzeige in einer DataGridView an Ihre eigenen Bedürfnisse anpassen und
- wie Sie ein Formular zur Anzeige einzelner Datensätze aus einer Datenquelle erzeugen.

Christoph Siebeck

Hinweis:

Bitte sichern Sie beim Bearbeiten des Studienhefts nicht nur das Projekt regelmäßig, sondern auch die Datenbankdatei. Erstellen Sie am besten Kopien des kompletten Projektordners.

1. Wörtlich übersetzt bedeutet **DataGridView** so viel wie „Datengitteransicht“.

1 Vorüberlegungen und Vorbereitungen

Bevor wir uns an die praktische Umsetzung machen, wollen wir noch einmal kurz wiederholen, welche Funktionen das Programm haben soll.

1.1 Vorüberlegungen

Das letzte Klassendiagramm aus dem Entwurf sah so aus:

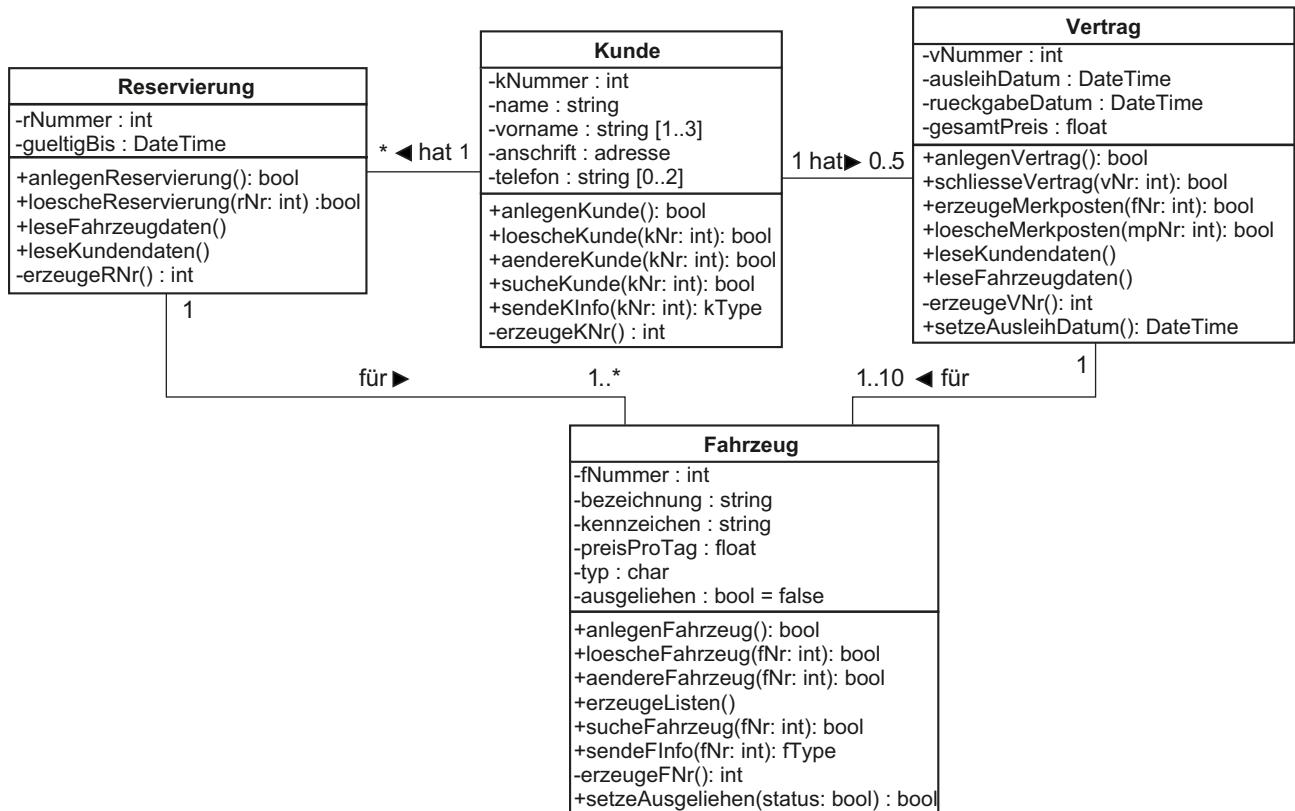


Abb. 1.1: Das Klassendiagramm von Autovermietung 2030

Wir benötigen insgesamt vier Klassen, die über verschiedene Felder und Methoden zusammenarbeiten sollen.

Die Speicherung der Daten erfolgt in einer Datenbank, deren Tabellen so aussehen sollen:

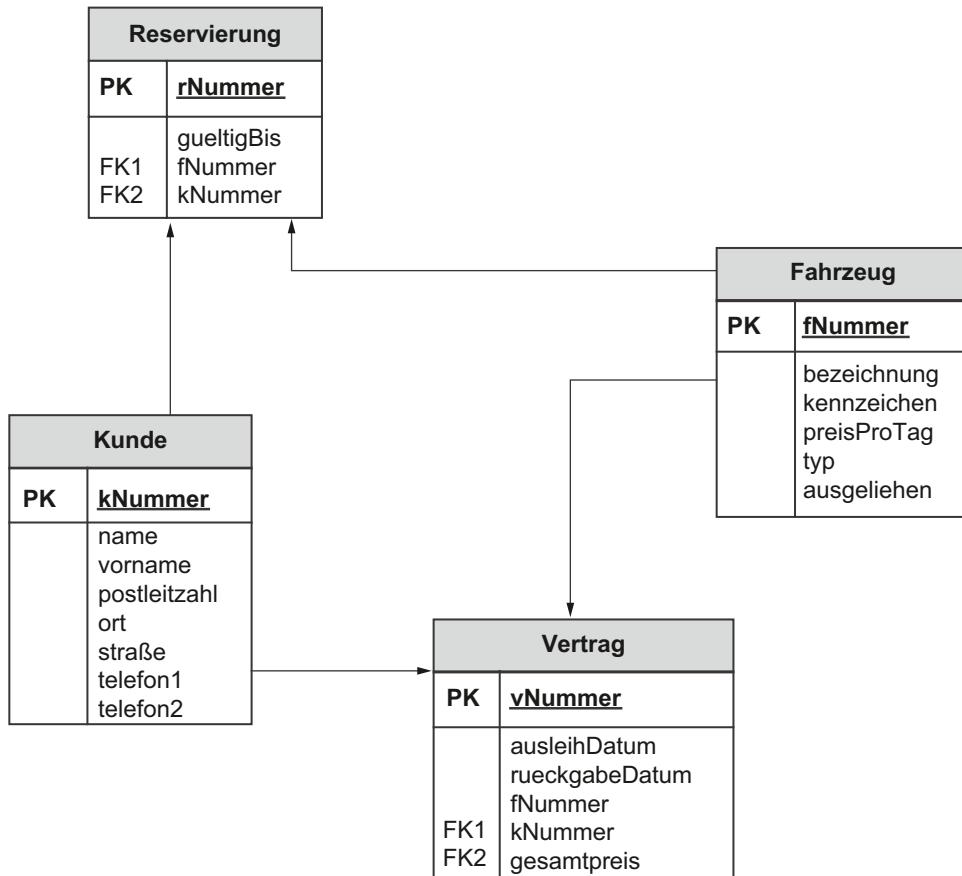


Abb. 1.2: Die Datenbanktabellen für die Autovermietung

In diesem Studienheft werden wir uns vor allem um die Funktionen zum Anlegen, Ändern und Löschen der Kunden- und Fahrzeugdaten kümmern. Diese Daten sind ja auch für das Anlegen der Reservierungen und der Verträge wichtig.

Für das Anzeigen und Bearbeiten der Daten stellen wir dem Anwender sowohl eine Listenansicht als auch eine Detailansicht zur Verfügung. In der Listenansicht werden alle vorhandenen Daten angezeigt, in der Detailansicht immer nur ein Eintrag.

Die Navigation durch die verschiedenen Datensätze soll vor allem über eine Symbolleiste erfolgen. Über diese Symbolleiste sollen auch neue Datensätze angelegt und vorhandene Datensätze gelöscht werden können. Dabei greifen wir im Wesentlichen auf vorgefertigte Funktionen von Visual Studio zu. Wir werden also zum Beispiel die Methoden `anlegenKunde()` und `anlegenFahrzeug()` aus dem Klassendiagramm nicht selbst programmieren. Auch für das Erzeugen der verschiedenen Nummern wie der Kunden- oder Fahrzeugnummer benutzen wir Funktionalitäten des Datenbank-Management-Systems. Das heißt, wir werden die entsprechenden Methoden ebenfalls nicht selbst erstellen.

Damit die Anwendung überschaubar bleibt und technisch einfach umzusetzen ist, werden wir folgende Anforderungen aus dem Entwurf nicht oder nicht komplett umsetzen:

- Wir erstellen nicht zwei getrennte Anwendungen beziehungsweise Oberflächen für die Kunden und die Mitarbeiter der Autovermietung, sondern fassen alle Funktionen in einem Programm zusammen. Für jeden Anwender stehen damit immer sämtliche Funktionen zur Verfügung.
- Die Listenfunktionen programmieren wir ebenfalls nicht komplett aus, sondern benutzen lediglich eine Standardlistendarstellung über ein DataGridView-Steuerelement.
- Auf die Einschränkungen für die maximale Anzahl von Verträgen und Reservierungen verzichten wir. In unserem Programm kann ein Kunde also beliebig viele Verträge und Reservierungen haben.
- Auf das Erzeugen und Verwalten von Merkposten verzichten wir ebenfalls. Bei der Rückgabe eines Fahrzeugs setzen wir lediglich ein Kennzeichen in der Datenbank zurück.

Die Daten unserer Anwendung legen wir in einer Datenbank des SQL Servers ab, die in einer Datenbankdatei gespeichert wird. Der Zugriff auf die Daten aus der eigentlichen Anwendung erfolgt über spezielle Datenbankschnittstellen. Damit besteht unser Programm also aus zwei getrennten Teilen – der Datenbank auf der einen Seite und der eigentlichen Anwendung auf der anderen Seite.

1.2 Vorbereitungen

Im ersten Schritt legen wir jetzt ein Formular an, über das wir später die verschiedenen Funktionen des Programms aufrufen können. Außerdem installieren wir die Datenbankkomponenten für Visual Studio.

Starten Sie bitte Visual Studio und erstellen Sie ein neues Projekt für eine Windows Forms-Anwendung. Als Namen können Sie **Autovermietung2030** verwenden.

Legen Sie dann in dem Formular zwei GroupBoxen an. Sie finden das entsprechende Steuerelement in der Gruppe **Container** der Toolbox. Die erste GroupBox soll den Text **Kunden** erhalten und die zweite den Text **Fahrzeuge**. Nehmen Sie danach in die erste GroupBox eine Schaltfläche mit dem Text **Listenansicht** auf. Setzen Sie anschließend den Titel in der Titelleiste des Formulars auf **Autovermietung2030** und verbreitern Sie das Formular ein wenig.

Hinweis:

Die weiteren Schaltflächen werden wir nach und nach ergänzen.

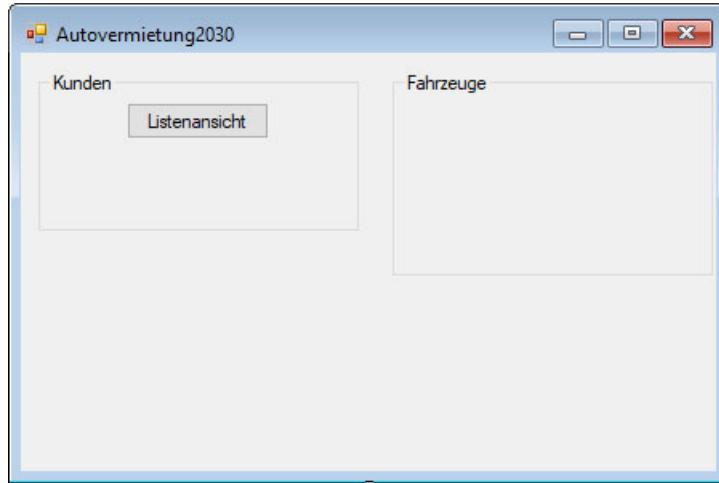


Abb. 1.3: Die Rohform des ersten Formulars

Legen Sie dann ein weiteres Formular unter dem Namen **Kundenliste** an. Setzen Sie den Text in der Titelleiste dieses Formulars auf **Kunden – Listenansicht**. Sorgen Sie anschließend dafür, dass das zweite Formular beim Anklicken der Schaltfläche **Listenansicht** im ersten Formular modal angezeigt wird.

Speichern Sie abschließend alle Änderungen und schließen Sie Visual Studio.

Installieren wir jetzt noch die Datenbankkomponenten. Starten Sie dazu den Visual Studio Installer. Sie finden ihn zum Beispiel im Startmenü. Bestätigen Sie danach die Abfrage durch die Benutzerkontensteuerung.

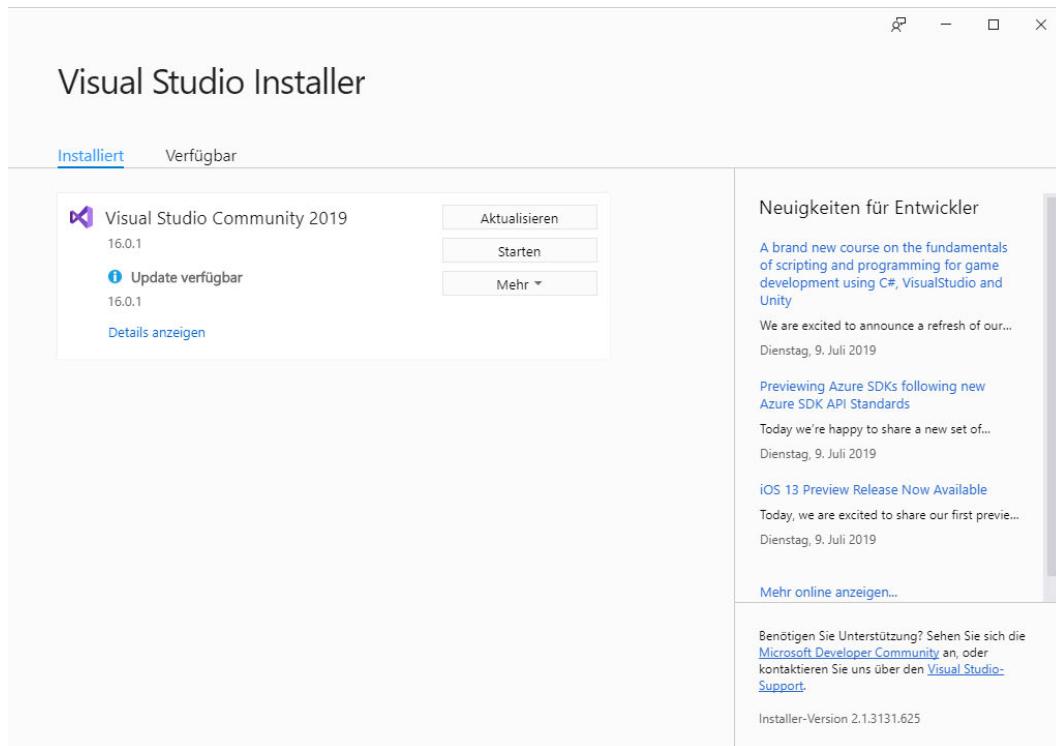


Abb. 1.4: Der Visual Studio Installer

Im Visual Studio Installer klicken Sie auf die Schaltfläche **Mehr** neben der installierten Version und wählen im Menü den Eintrag **Ändern**.

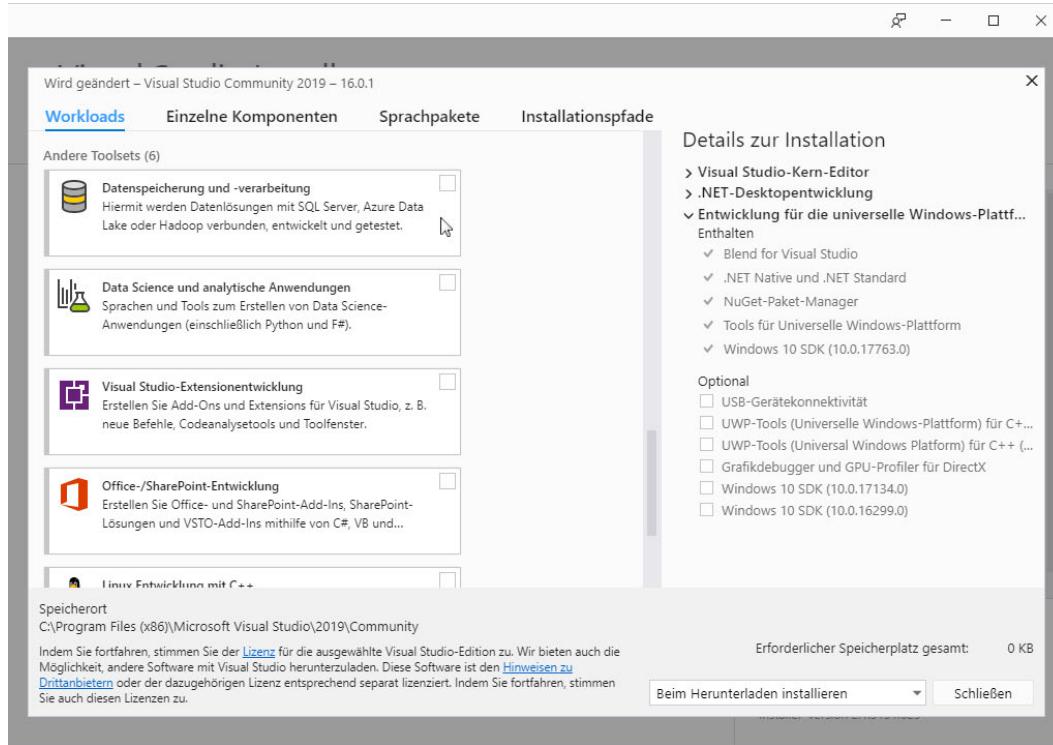


Abb. 1.5: Der Workload Datenspeicherung und -verarbeitung (links in der Abbildung am Mauszeiger)

Im folgenden Fenster markieren Sie anschließend den Workload **Datenspeicherung und -verarbeitung** im Bereich **Andere Toolsets**. Sie finden diesen Bereich unten bei der Auswahl der Workloads.

Klicken Sie dann auf die Schaltfläche **Ändern** rechts unten im Fenster, um die Installation zu starten. Danach werden die Dateien heruntergeladen und installiert. Das kann durchaus einige Zeit dauern. Schließen Sie danach den Visual Studio Installer bitte wieder.

Damit sind die wichtigsten Vorbereitungen bereits erledigt. Im nächsten Kapitel werden wir die Datenbank erstellen.

2 Erstellen der Datenbank

In diesem Kapitel werden wir die Datenbank anlegen, in der die Daten unserer Anwendung gespeichert werden. Dazu verwenden wir das Datenbank-Management-System SQL Server mit einer Datenbankdatei und den Server-Explorer von Visual Studio.



SQL Server arbeitet mit dem relationalen Datenmodell. Bei diesem Modell werden die Daten – wie Sie ja bereits wissen – in Tabellenform abgelegt.

Starten Sie Visual Studio und laden Sie das Projekt für unsere Autovermietung. Öffnen Sie danach bitte den Server-Explorer. Klicken Sie dazu auf die Registerzunge **Server-Explorer**. Sie finden sie links oben bei der Registerzunge für die Toolbox.

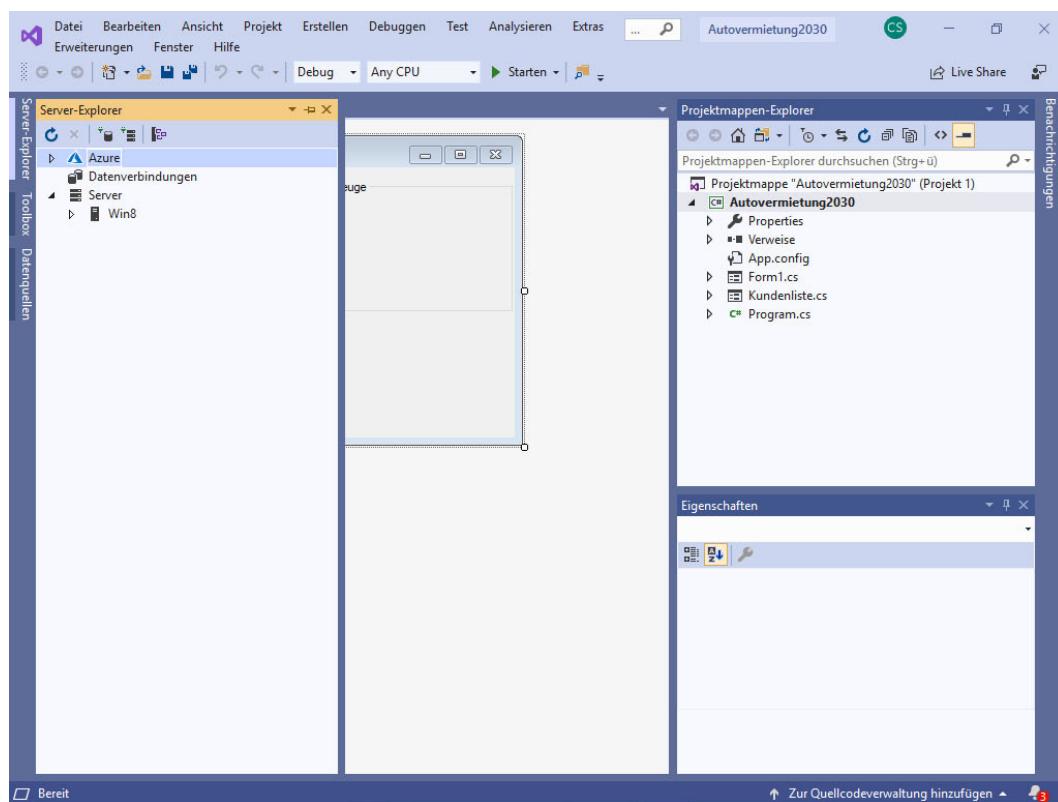


Abb. 2.1: Das eingeblendete Register **Server-Explorer** (links in der Abbildung)

Im Server-Explorer klicken Sie dann auf das Symbol **Mit Datenbank verbinden**.

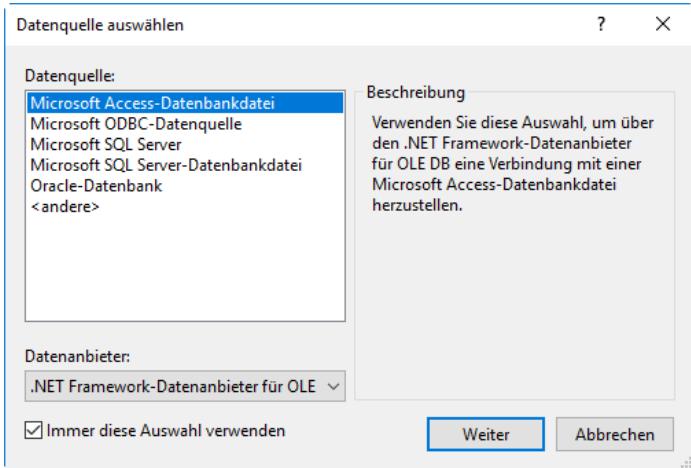


Abb. 2.2: Der Dialog **Datenquelle auswählen**

Im Dialog **Datenquelle auswählen** markieren Sie den Eintrag **Microsoft SQL Server-Datenbankdatei**. Klicken Sie dann auf **Weiter**.

Tipp:

Über das Feld **Immer diese Auswahl verwenden** können Sie festlegen, ob neue Datenbanken immer mit der gerade ausgewählten Datenquelle erstellt werden sollen. Der Dialog **Datenquelle auswählen** erscheint dann nicht mehr.

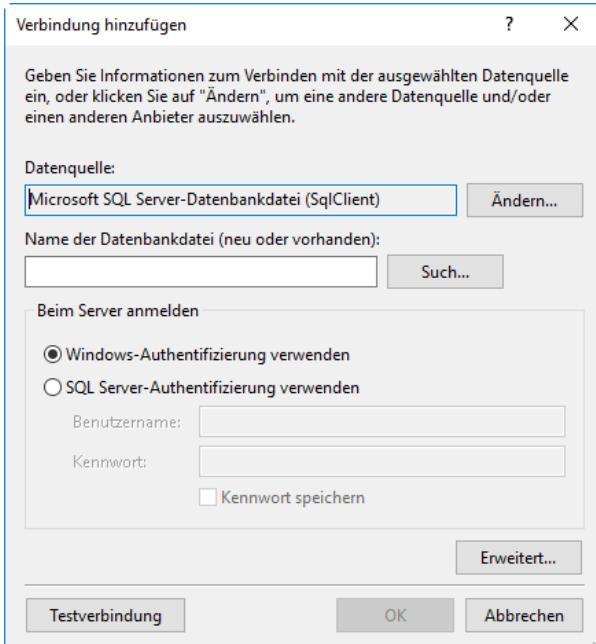


Abb. 2.3: Der Dialog **Verbindung hinzufügen**

Überprüfen Sie im Dialog **Verbindung hinzufügen** noch einmal, ob bei der Datenquelle eine Microsoft SQL Server-Datenbankdatei angegeben ist. Wenn das nicht der Fall ist, wählen Sie den entsprechenden Eintrag bitte über die Schaltfläche **Ändern...** aus. Geben Sie anschließend einen Namen für die Datenbankdatei in das Feld **Name der Datenbankdatei:** ein. Wir verwenden in unserem Beispiel den Namen **auto2030**.

Hinweis:

Die Datenbankdatei wird in der Standardeinstellung in Ihrem Ordner **Dokumente** angelegt. Wenn Sie einen anderen Ordner verwenden möchten, wählen Sie ihn vor dem Anlegen mit der Schaltfläche **Such...** neben dem Feld für den Namen aus.

Die Einstellungen für die Anmeldung beim Server können Sie unverändert stehen lassen. Klicken Sie anschließend auf **OK** und bestätigen Sie die Abfrage, ob die Datenbankdatei erstellt werden soll.

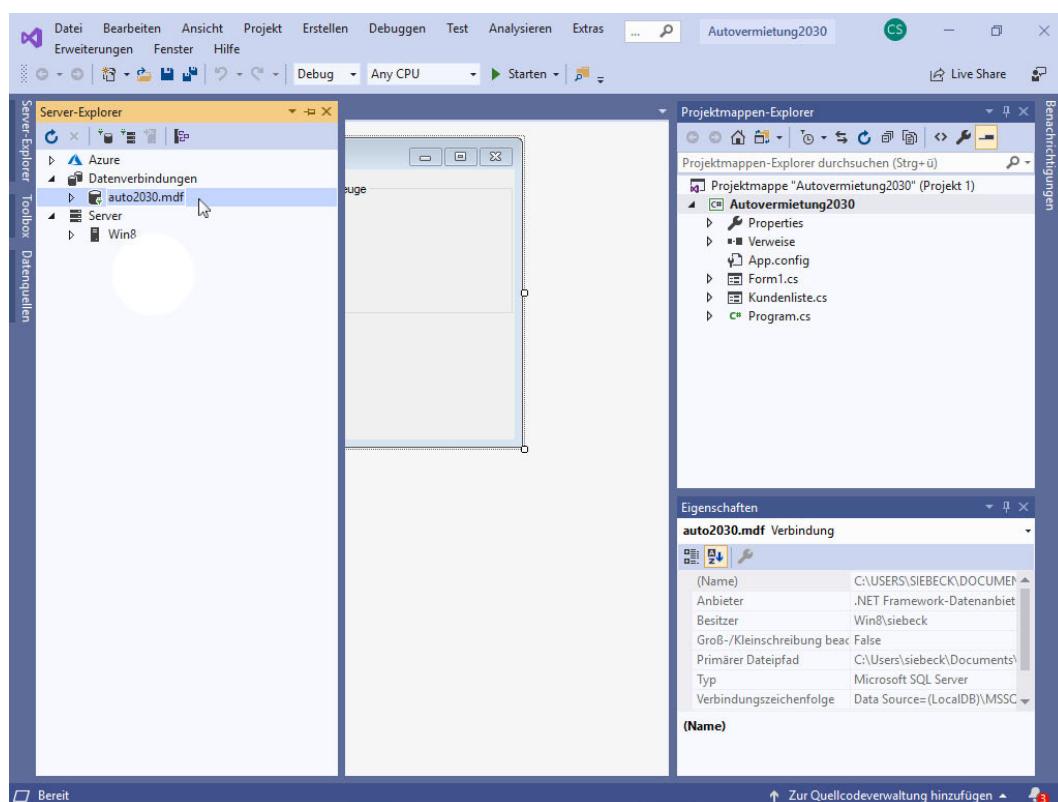


Abb. 2.4: Die neue Datenbankdatei (links oben im Server-Explorer am Mauszeiger)

Nach dem Anlegen der Datenbankdatei erscheint auch eine neue Datenverbindung im Server-Explorer.

Im nächsten Schritt müssen wir nun noch eine Tabelle in der gerade erstellten Datenbank anlegen. Dazu benutzen wir ebenfalls den Server-Explorer von Visual Studio.

Klicken Sie im Server-Explorer auf das Symbol vor dem Eintrag **auto2030.mdf**. Dabei wird gegebenenfalls auch automatisch eine Verbindung zu der Datenbank hergestellt. Das kann durchaus einen Moment dauern.

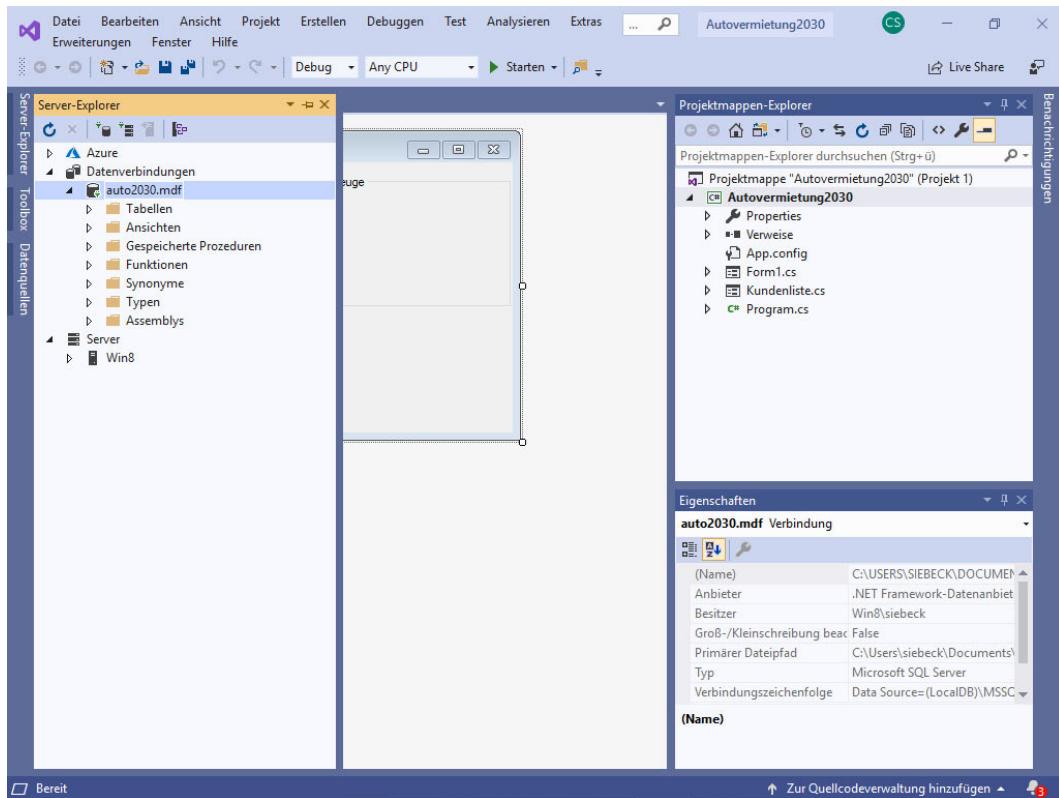


Abb. 2.5: Die Objekte einer Datenbank (links in der Abbildung)

In der Baumstruktur klicken Sie anschließend mit der rechten Maustaste auf den Eintrag **Tabellen** und wählen im Kontext-Menü die Funktion **Neue Tabelle hinzufügen**.

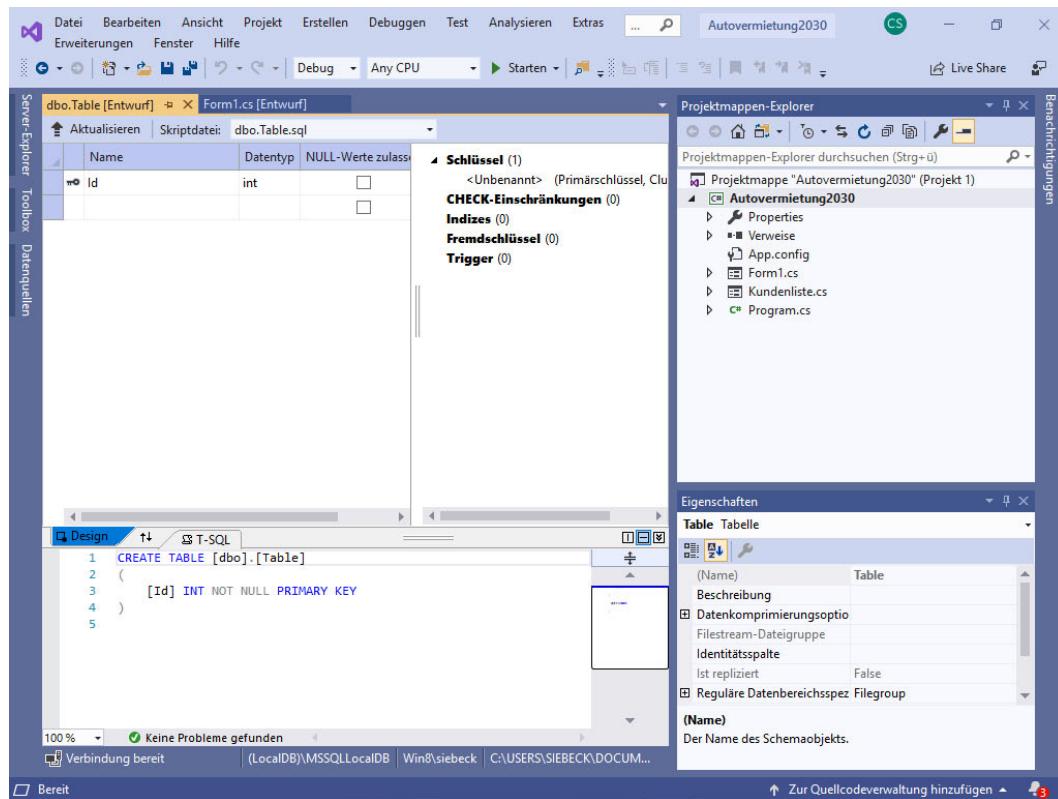


Abb. 2.6: Das Anlegen einer neuen Tabelle

Danach legen Sie dann die verschiedenen Spalten – die Datenfelder – für die Datenbanktabelle an.



Noch einmal zur Erinnerung:

Im relationalen Datenmodell beschreiben die Spalten der Tabelle die Eigenschaft der Daten. Jede Spalte hat einen eindeutigen Namen und einen Datentyp.

Geben Sie dazu jeweils den Namen in das Feld **Name** ein und legen Sie anschließend über das Feld **Datentyp** den gewünschten Datentyp fest. Das Feld **NULL-Werte zulassen** steuert, ob das Datenfeld auch leer bleiben darf. Im Feld **Standard** können Sie einen Standardwert für das Feld festlegen. Das Feld ganz links schließlich zeigt an, ob es sich bei dem Feld um einen Primärschlüssel handelt. Wenn das der Fall ist, wird hier ein kleiner Schlüssel angezeigt.

Die Spalten zur Speicherung der Kundendaten lassen sich recht einfach aus dem Datenbankmodelldiagramm und auch dem Klassendiagramm übernehmen.

Kunde	
PK	<u>kNummer</u>
	name vorname postleitzahl ort straße telefon1 telefon2

Kunde	
-kNummer : int -name : string -vorname : string [1..3] -anschrift : adresse -telefon : string [0..2]	+anlegenKunde(): bool +loescheKunde(kNr: int): bool +aendereKunde(kNr: int): bool +sucheKunde(kNr: int): bool +sendeKInfo(kNr: int): kType -erzeugeKNr() : int

Abb. 2.7: Die Tabelle für die Kundendaten (links) und die Klasse **Kunde** (rechts)

Die Kundennummer soll dabei automatisch vom System vergeben werden und als Primärschlüssel dienen. Alle anderen Daten speichern wir als Zeichenketten mit unterschiedlichen Längen.

Hinweis:

Die Vornamen haben wir beim Entwurf zu einer Spalte in der Tabelle zusammengefasst, da wir die Daten nicht getrennt verarbeiten wollen. Wir werden die Spalte aber gleich so breit anlegen, dass mehrere Vornamen erfasst werden können.

Sie können die Spalte für die zweite Telefonnummer auch weglassen. Denken Sie dann aber bitte im weiteren Verlauf des Studienhefts daran, dass Sie diese Spalte nicht übernommen haben.

Die Postleitzahl und auch die Telefonnummer speichern wir als Zeichenketten und nicht als numerische Werte. Andernfalls könnten wir ja keine führenden Nullen erfassen.

Beginnen wir mit dem Anlegen der Kundennummer.

Ändern Sie bitte den Eintrag in der Spalte **Name** in `kNummer`. Den Datentyp und auch die anderen Einstellungen in der Zeile können Sie unverändert übernehmen.

Bitte beachten Sie:

In dem Kombinationsfeld **Datentyp** werden Ihnen nur die Datentypen des SQL Server angeboten. Die Bezeichnung dieser Datentypen entspricht **nicht** den Bezeichnungen der Datentypen des .NET Frameworks. Der Datentyp `int` des SQL Server steht zum Beispiel für den Datentyp `Int32` des .NET Frameworks. Eine ausführlichere Tabelle mit den jeweiligen Entsprechungen finden Sie weiter unten.



Damit haben wir zunächst einmal eine Spalte **kNummer** für numerische Daten angelegt. Über die Spalteneigenschaften unten rechts im Eigenschaftenfenster müssen wir nun noch festlegen, dass der Wert automatisch vergeben werden soll. Das erfolgt über die Eigenschaft (**Ist Identity**)² in der Gruppe **Identitätsspezifikation**.

2. *Identity* lässt sich mit „Identität“ übersetzen.

Ändern Sie hier den Wert von `False` in `True`. Dadurch werden auch die Eigenschaften **ID-Inkrement** und **Identitätsseed³** automatisch auf den Wert 1 gesetzt. Unsere automatisch vergebene Kundennummer beginnt damit bei 1 und wird für jeden neuen Datensatz um den Wert 1 erhöht.



Bitte beachten Sie:

Unsere Kundennummer wird jetzt automatisch vom Datenbank-Management-System verwaltet. Wir benötigen daher die Methode `erzeugekNr()` der Klasse `Kunde`, die diese Aufgabe eigentlich übernehmen sollte, nicht mehr.

Damit wäre die erste Spalte für die eindeutige Kundennummer erstellt. Sie dient auch als Primärschlüssel.

Im nächsten Schritt legen wir eine Spalte für den Kundennamen an. Klicken Sie dazu mit der Maus in die Spalte **Name** in die leere Zeile unten in der Tabelle für die Spalten. Geben Sie dann als Namen `kName` ein. Für den Datentyp wählen Sie den Eintrag `nvarchar(50)`. Damit können Zeichenketten mit 50 Zeichen gespeichert werden. Die Anzahl der Zeichen können Sie über die Eigenschaft **Length** gezielt verändern. Sie können aber auch einfach die Zahl in den Klammern mit der gewünschten Länge überschreiben.

Da der Kundennname nicht leer bleiben darf, ändern Sie abschließend noch die Markierung in der Spalte **NUL-Werte zulassen**. Die Tabelle sollte nun so aussehen:

Name	Datentyp	NULL-Werte zulassen
<code>kNummer</code>	<code>int</code>	<input type="checkbox"/>
<code>kName</code>	<code>nvarchar(50)</code>	<input checked="" type="checkbox"/>

```

CREATE TABLE [dbo].[Table]
(
    [kNummer] INT NOT NULL PRIMARY KEY IDENTITY,
    [kName] NVARCHAR(50) NOT NULL
)

```

Abb. 2.8: Die ersten beiden Spalten in der Datenbanktabelle

3. Seed lässt sich hier am besten mit „Startwert“ übersetzen.

Legen Sie anschließend die weiteren Spalten an. Die erforderlichen Daten finden Sie in der Tab. 2.1.

Tab. 2.1: Die weiteren Spalten für die Datenbanktabelle

Spaltenname	Datentyp	NULL-Werte zulassen	Länge
vorname	nvarchar	nein	50
strasse	nvarchar	nein	50
postleitzahl	nvarchar	nein	5
ort	nvarchar	nein	50
telefon1	nvarchar	ja	20
telefon2	nvarchar	ja	20

Bitte beachten Sie:

Legen Sie die Spalten exakt in der Reihenfolge wie im Studienheft an. Andernfalls funktioniert unter Umständen später das Überprüfen der Daten nicht korrekt.

Sie sollten keine Spaltennamen verwenden, die auch vom SQL Server beziehungsweise von Visual Studio Express selbst intern benutzt werden. Wenn Sie zum Beispiel eine Spalte **name** anlegen, erhalten Sie unter Umständen später beim Ausführen des Programms seltsame Fehlermeldungen.



Jetzt setzen wir noch den Namen der Tabelle auf **Kunde**. Dazu ändern Sie den Eintrag `[Table]` in der Anweisung

```
CREATE TABLE [dbo]. [Table]
```

oben im Register **T-SQL** in `[Kunde]`. Die Anweisung sollte also so aussehen:

```
CREATE TABLE [dbo]. [Kunde]
```

Damit ist unsere erste Datenbanktabelle vollständig erfasst.

Wir müssen jetzt noch die Datenbank aktualisieren lassen, damit die Änderungen übernommen werden. Klicken Sie dazu auf die Schaltfläche **Aktualisieren** links oberhalb der Tabelle. Im Fenster **Vorschau der Datenbankupdates** klicken Sie anschließend auf die Schaltfläche **Datenbank aktualisieren**. Der Server-Explorer führt dann die Änderungen an der Datenbank durch und zeigt Ihnen die Ergebnisse an.

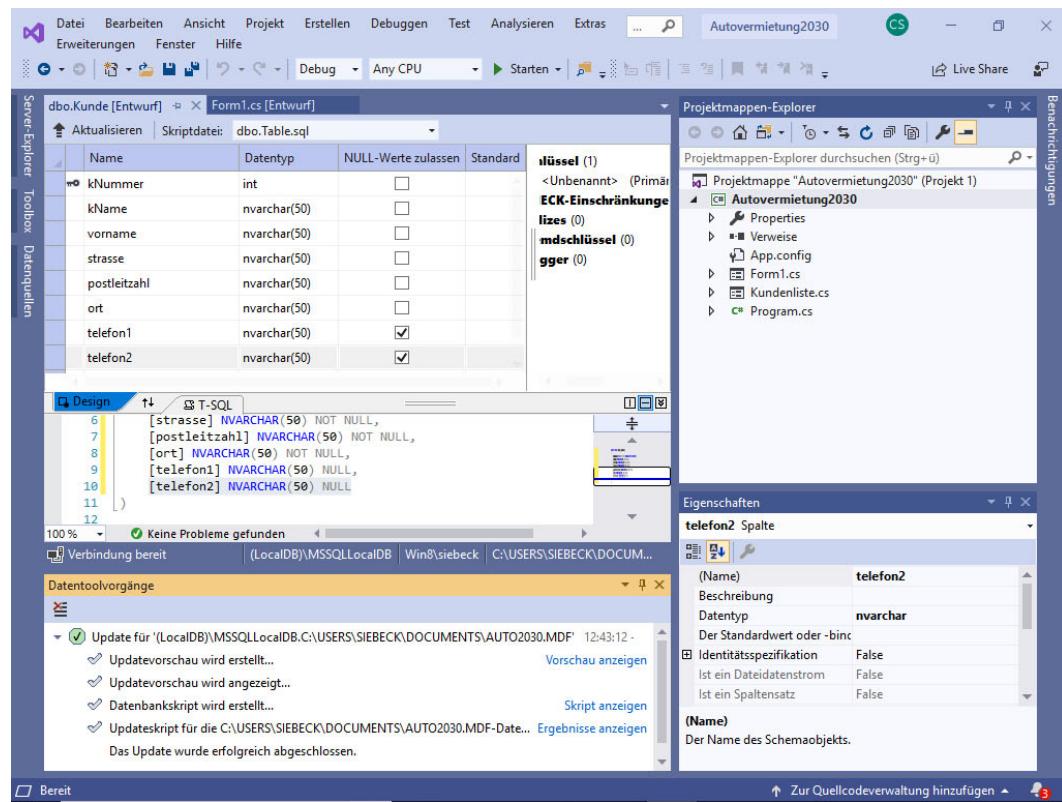


Abb. 2.9: Die aktualisierte Datenbank

Im nächsten Kapitel werden wir die Verbindung zwischen der Tabelle und der Anwendung herstellen.

Exkurs: Die Datentypen des SQL Server und ihre Entsprechungen im .NET Framework

Wie Sie ja bereits wissen, arbeitet der SQL Server mit eigenen Datentypen, die sich sowohl in den darstellbaren Werten als auch im Namen zum Teil von den .NET Framework-Datentypen unterscheiden. Die wichtigsten Datentypen mit der jeweiligen Entsprechung finden Sie in der folgenden Tabelle.

SQL Server Datentyp	.NET Framework/C#-Entsprechung
nvarchar	String/string beziehungsweise Char/char
bit	Boolean/bool
tinyint	Byte/byte
smallint	Int16/short
int	Int32/int
bigint	Int64/long
real	Single/float
float	Double/double

Außerdem kennt der SQL Server noch weitere spezialisierte Datentypen wie `image` zum Speichern von Bildern oder `datetime` zum Speichern von Datum und Uhrzeit. Den Datentyp `datetime` werden wir später unter anderem zum Speichern der Datumswerte verwenden.

Zusammenfassung

Mit dem Datenbank-Management-System SQL Server können Sie Daten strukturiert in Datenbanken verwalten. SQL Server arbeitet mit dem relationalen Datenmodell.

Um eine neue Datenbank anzulegen, benutzen Sie den Server-Explorer von Visual Studio. Über den Server-Explorer können Sie dann auch Tabellen für die Datenbank erstellen.

Für jede Spalte in einer Datenbanktabelle müssen Sie einen Namen und den Datentyp festlegen.

Der SQL Server verwendet andere Datentypen als Visual Studio.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 2.1 Beschreiben Sie, wie Sie mit dem Server-Explorer von Visual Studio eine neue Datenbank für den SQL Server anlegen.

- 2.2 Wie können Sie für eine bereits vorhandene Datenbank eine neue Tabelle erstellen?

- 2.3 Welche Bedeutung hat das Feld **NULL-Werte zulassen** beim Anlegen einer Spalte für eine Datenbanktabelle?

- 2.4 Sie wollen für eine Spalte in einer Datenbanktabelle die Daten automatisch fortschreiben lassen. Der Startwert soll 10 sein. Für jeden neuen Datensatz soll dieser Wert um 5 erhöht werden. Beschreiben Sie, welche Einstellungen Sie für die Spalte setzen müssen.

3 Das Zusammenspiel zwischen Datenbank und Anwendung

In diesem Kapitel werden wir die Datenbank und unsere Anwendung verbinden. Dazu legen wir eine neue Datenquelle an und lassen dann die Daten aus der Datenbanktabelle **Kunde** in einem speziellen Steuerelement anzeigen – der *DataGridView*.

Zuerst wollen wir uns aber kurz ansehen, wie das Zusammenspiel zwischen einer Datenbank und einer Anwendung grundsätzlich erfolgt.

3.1 Ein wenig Theorie

Der Zugriff aus einer .NET-Anwendung auf Daten in einer Datenbank erfolgt nicht direkt, sondern über die Datenzugriffsschnittstelle **ADO.NET**⁴. Diese Datenzugriffs-schnittstelle ist Bestandteil des .NET Frameworks.

ADO.NET selbst besteht aus verschiedenen **Datenprovidern**, die den Zugriff auf unterschiedliche Datenbank-Management-Systeme ermöglichen. Einen dieser Datenprovider – nämlich denjenigen für den SQL Server – haben Sie ja auch bereits beim Anlegen der neuen Datenbank eingesetzt.

Die Datenprovider bieten dann – etwas vereinfacht dargestellt – die Daten aus einer Datenbanktabelle als **Datenquelle** an. Diese Datenquelle wiederum können Sie mit Steuer-elementen in einem Formular verbinden und so die Daten aus der Datenbanktabelle in einer Anwendung anzeigen lassen und auch bearbeiten.

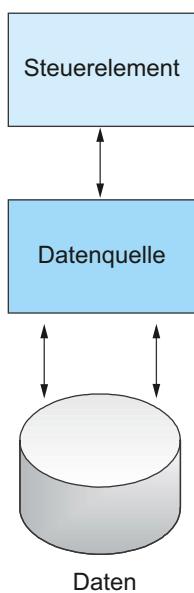


Abb. 3.1: Der Zugriff aus einer Anwendung auf eine Datenbanktabelle

4. ADO steht für *Active X Data Objects*.



Der Zugriff auf eine Datenbanktabelle aus einer Anwendung erfolgt in der Regel nicht direkt, sondern über eine Datenquelle.

Dieser scheinbare Umweg hat seinen guten Grund. Denn beim direkten Zugriff müssten Sie für nahezu jedes Datenbank-Management-System auch ein eigenes Zugriffsverfahren benutzen, das nur für dieses eine Datenbank-Management-System funktionieren würde. Beim Einsatz eines anderen Datenbank-Management-Systems müssten Sie sich wieder neu in das Zugriffsverfahren eindenken und einarbeiten.

Durch den Zugriff über die Datenquelle dagegen ist sichergestellt, dass Sie eine einheitliche Schnittstelle verwenden können – egal, welches Datenbank-Management-System nun die Daten liefert. Das heißt, Sie müssen sich nur einmal in den Umgang mit der Schnittstelle einarbeiten. Um die Umsetzung der Anweisungen in die konkreten Befehle für ein bestimmtes Datenbank-Management-System dagegen müssen Sie sich nicht mehr kümmern. Diese Aufgabe nimmt Ihnen die Datenquelle ab.

3.2 Das Anlegen der Datenquelle

Schauen wir uns jetzt das Anlegen einer Datenquelle am Beispiel unserer Anwendung an. Es erfolgt sehr komfortabel über einen Assistenten.

Wechseln Sie in ein beliebiges Formular der Anwendung. Öffnen Sie dann das Menü **Projekt** und klicken Sie auf **Neue Datenquelle hinzufügen...**

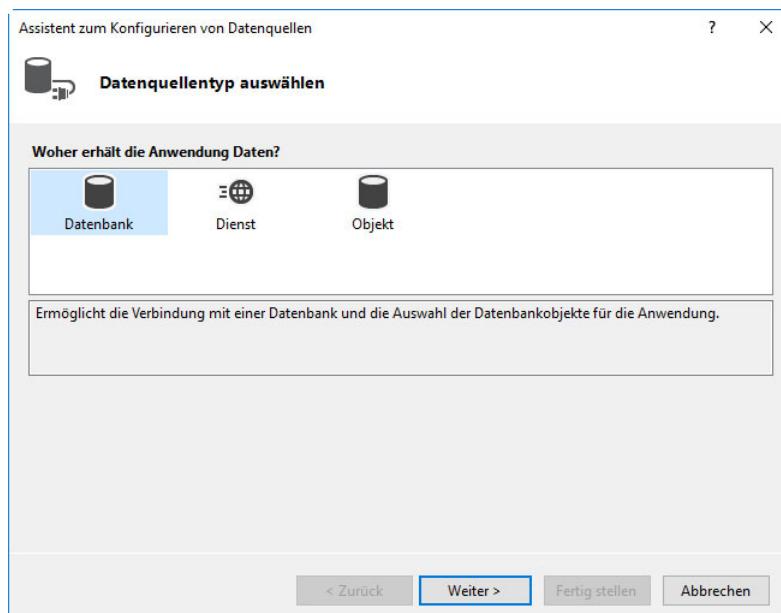


Abb. 3.2: Der Assistent zum Konfigurieren von Datenquellen

Im ersten Schritt des Assistenten wählen Sie aus, woher die Daten stammen sollen. In der Regel ist der Eintrag **Datenbank** bereits markiert und Sie können über die Schaltfläche **Weiter >** zum nächsten Schritt gehen.

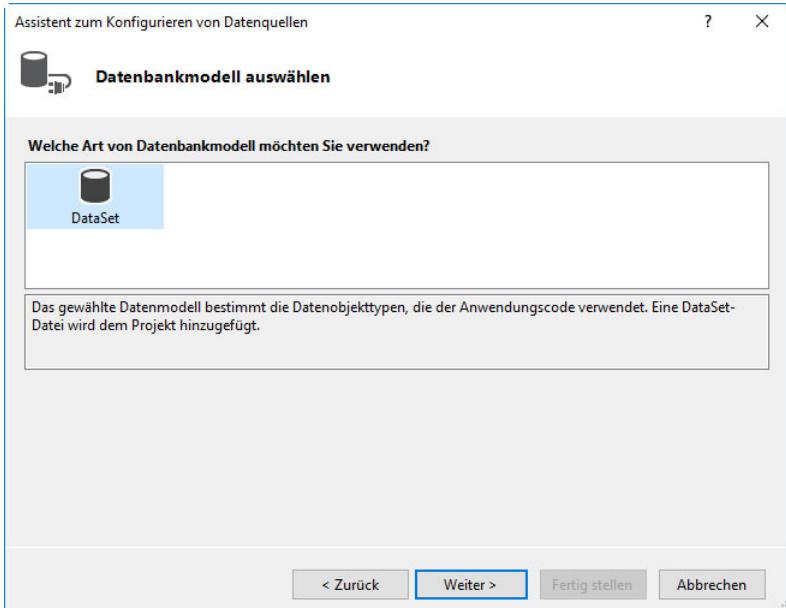


Abb. 3.3: Die Auswahl des Datenbankmodells

Im zweiten Schritt legen Sie das Datenbankmodell fest. Wir arbeiten in unserem Beispiel mit einem DataSet. Da kein anderer Eintrag angeboten wird, können Sie also wieder mit der Schaltfläche **Weiter >** zum nächsten Schritt gehen.

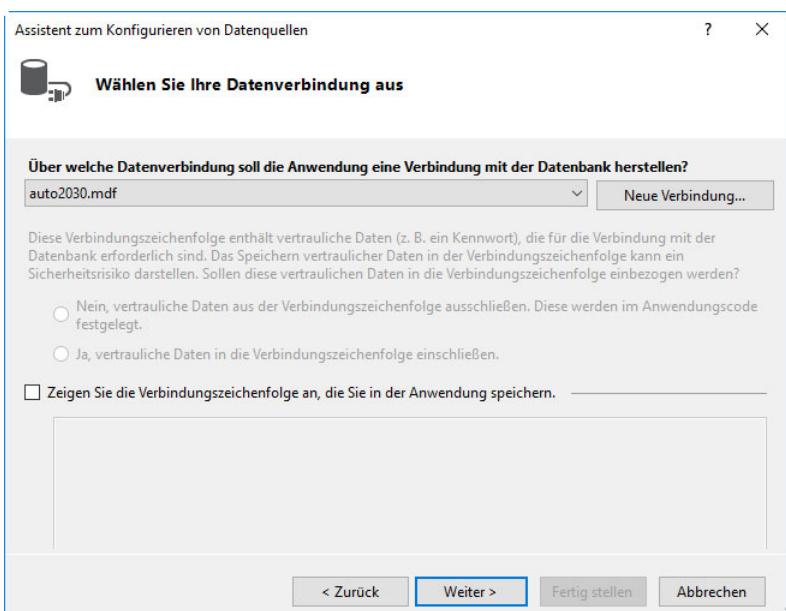


Abb. 3.4: Die Auswahl der Datenverbindung

Im dritten Schritt wählen Sie die Datenverbindung aus. Da wir bisher nur eine einzige Verbindung angelegt haben, schlägt der Assistent direkt den richtigen Eintrag **auto2030.mdf** vor. Sie können also mit der Schaltfläche **Weiter >** zum nächsten Schritt gehen.

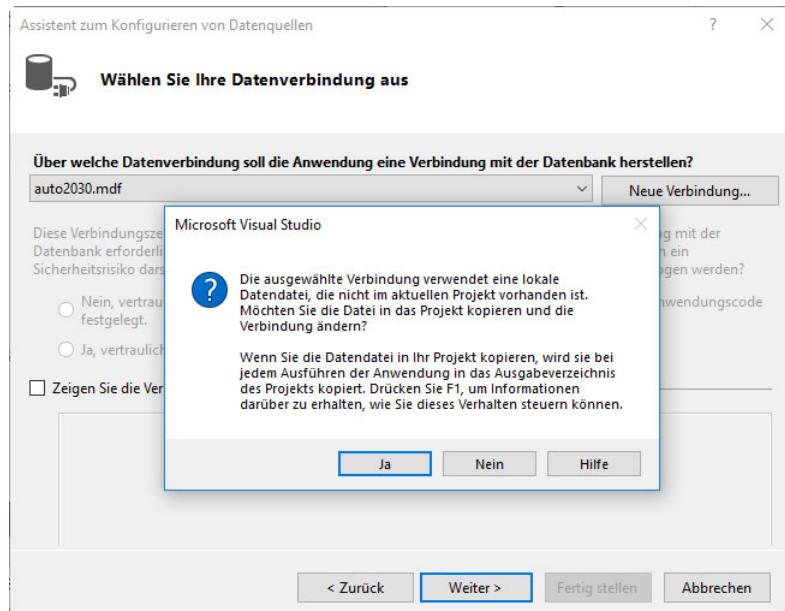


Abb. 3.5: Die Rückfrage zum Kopieren

Im folgenden Dialog fragt Sie Visual Studio, ob Sie die Datendatei in das Projekt kopieren möchten. Das erleichtert die Pflege und auch die Sicherung der Daten deutlich, da Sie alle Dateien des Projekts in einem Ordner zusammenhalten können. Sie sollten daher hier auf **Ja** klicken.



Bitte beachten Sie:

Sie müssen keine Kopie erstellen lassen. Sie können auch ausschließlich mit dem Original arbeiten. Denken Sie dann aber bitte daran, dass sich dieses Original in unserem Beispiel nicht im Projektordner befindet.

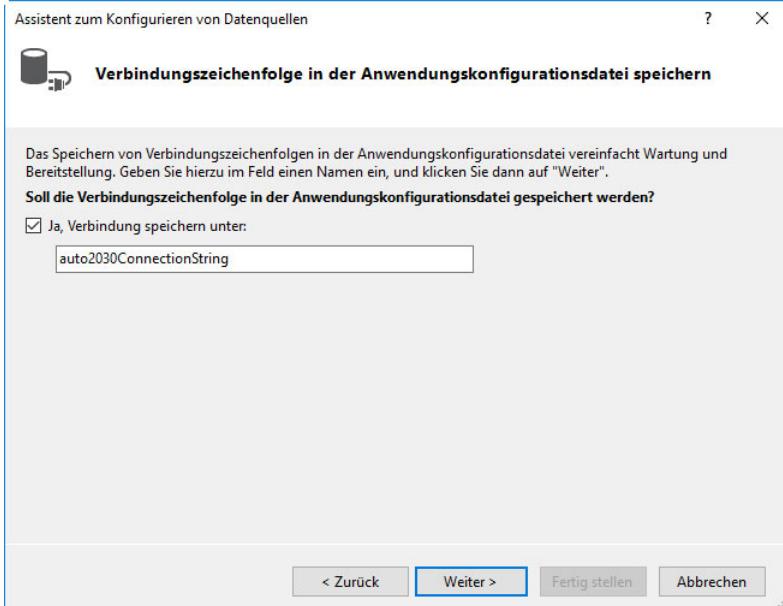


Abb. 3.6: Das Speichern der Verbindungszeichenfolge

Im nächsten Schritt sind keine Änderungen erforderlich. Überprüfen Sie zur Sicherheit noch einmal, ob das Kontrollkästchen **Ja, Verbindung speichern unter** markiert ist, und klicken Sie auf **Weiter >**.

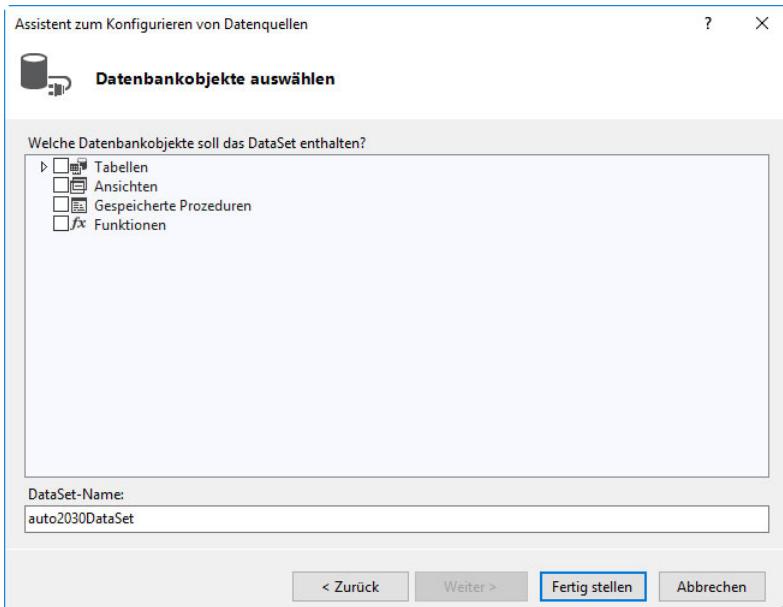


Abb. 3.7: Die Auswahl der Datenbankobjekte

Der Assistent ruft dann einige Informationen aus der Datenbank ab und zeigt Ihnen nach kurzer Zeit eine Baumstruktur mit den Datenbankobjekten an. In unserem Fall gibt es bisher lediglich eine Tabelle – und zwar für die Kunden. Aus dieser Tabelle sollen auch die Daten beschafft werden.

Öffnen Sie den Zweig für die Tabellen mit einem Mausklick auf das Symbol vor dem Eintrag. Markieren Sie dann die Tabelle **Kunde** durch einen Mausklick in das Kontrollkästchen vor dem Eintrag. Das Fenster sollte nun so aussehen:

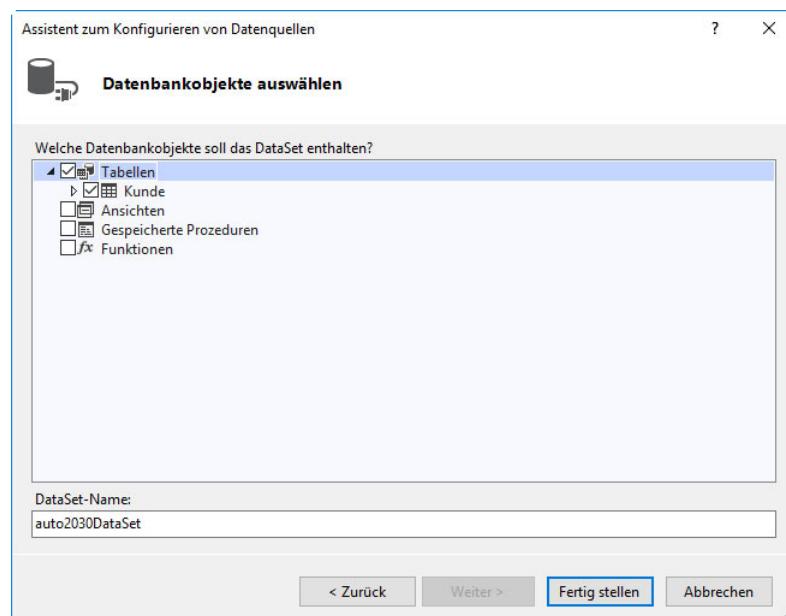


Abb. 3.8: Die ausgewählte Tabelle **Kunde**

Beenden Sie anschließend das Anlegen der Datenquelle über die Schaltfläche **Fertig stellen**. Der Assistent legt nun automatisch verschiedene Dateien an.

3.3 Das Steuerelement DataGridView

Nachdem wir die Datenquelle angelegt haben, können wir jetzt auch ein Steuerelement für die Anzeige und das Bearbeiten der Daten in das zweite Formular einbauen. Das erfolgt mit einigen wenigen Mausklicks.

Wechseln Sie bitte in das Formular **Kundenliste**. Lassen Sie dann über die Registerzunge **Datenquellen** links im Fenster die vorhandenen Datenquellen anzeigen.

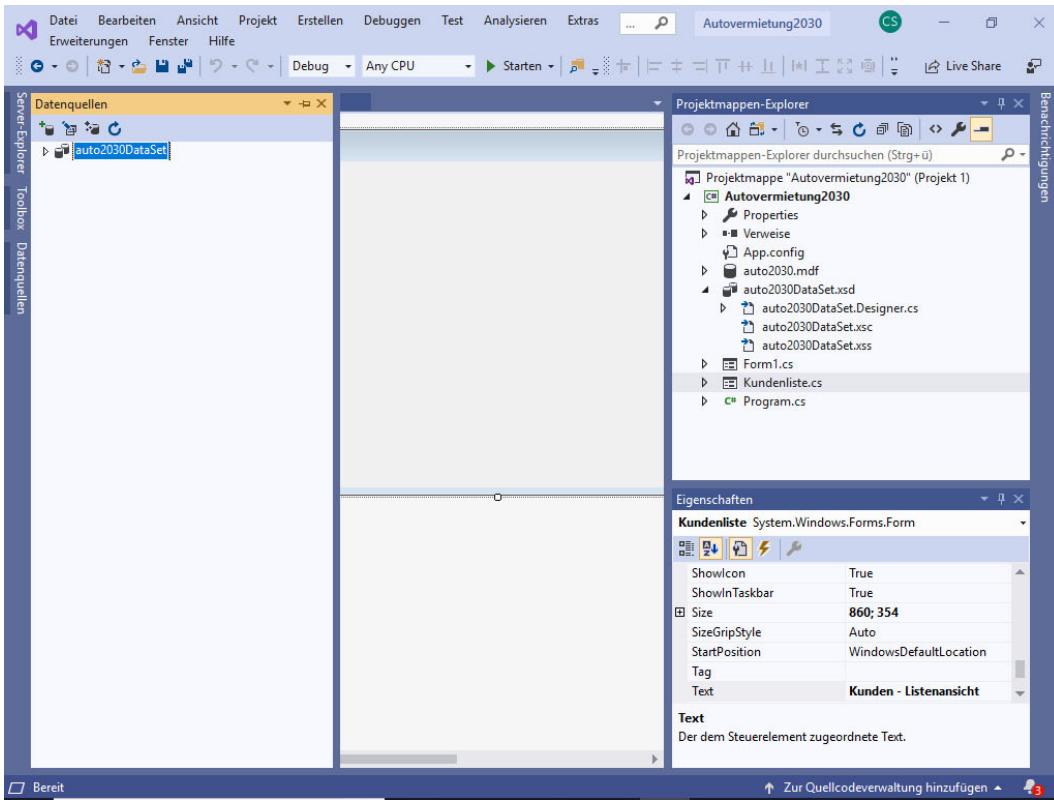


Abb. 3.9: Die Anzeige der Datenquellen (links oben in der Abbildung)

Im linken Bereich des Fensters erscheint jetzt die Datenquelle **auto2030DataSet** mit unserer Tabelle **Kunde**.

Hinweis:

Unter Umständen müssen Sie einmal auf das Symbol vor dem Eintrag der Datenquelle klicken, um die untergeordneten Objekte anzuzeigen.

Sie müssen nun nichts weiter machen, als den Eintrag der Tabelle mit der Maus in das Formular zu ziehen. Sobald Sie die Maustaste loslassen, erstellt Visual Studio Express automatisch verschiedene Steuerelemente. Eines davon ist eine DataGridView – ein Steuerelement mit einer Tabelle, das die Spalten und Zeilen – also die Datenfelder und die Datensätze – in der Datenbanktabelle anzeigt.

Probieren Sie das jetzt bitte aus. Ziehen Sie die Tabelle **Kunde** mit gedrückter linker Maustaste in das Formular und lassen Sie die Maustaste dann los.

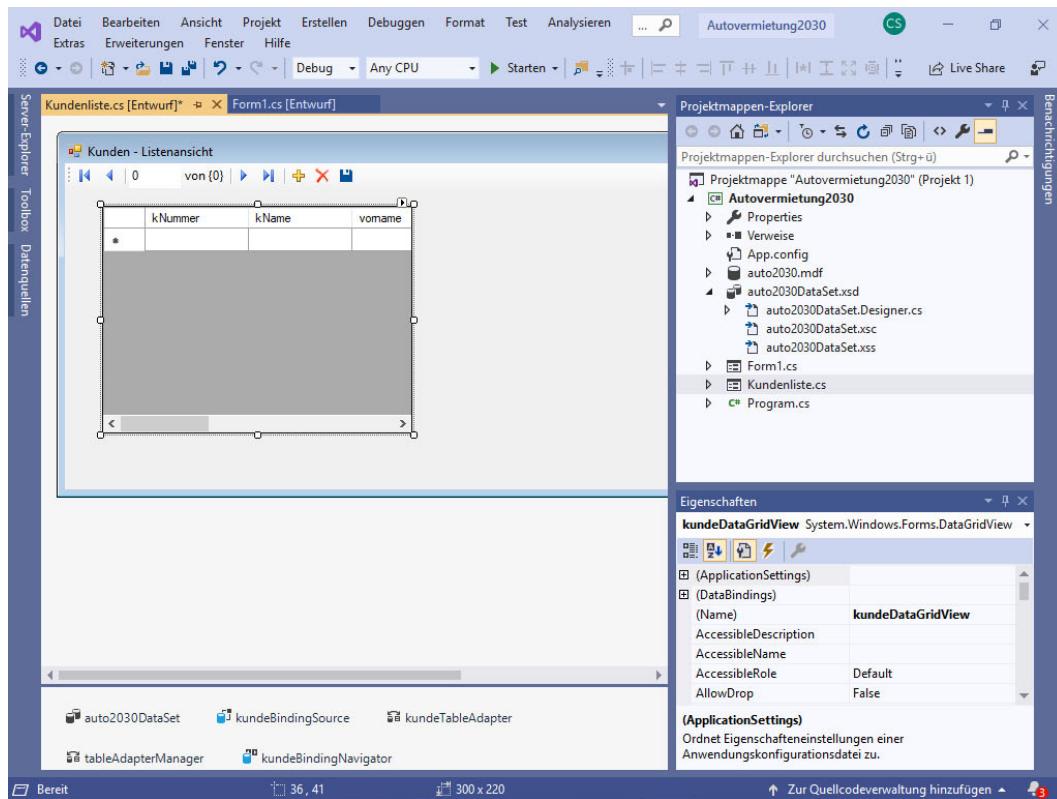


Abb. 3.10: Die Steuerelemente für den Zugriff auf die Datenquelle

Nach einiger Zeit finden Sie oben im Formular eine Art Navigationsleiste und darunter die Tabelle. Wenn Sie sich die Tabelle einmal genau ansehen, werden Sie feststellen, dass die Texte in den Spaltenköpfen den Namen der Spalten in der Datenbanktabelle entsprechen.

Unten im Formular werden außerdem mehrere nicht visuelle Steuerelemente für die Datenquelle angezeigt. Die vier wichtigsten sind:

- `auto2030DataSet` für den Zugriff auf die eigentliche Datenbank,
- `kundeTableAdapter` für den Zugriff auf die Tabelle **Kunde**,
- `kundeBindingSource` für die Kommunikation zwischen den Steuerelementen und der Datenbank und
- `kundeBindingNavigator` für die Navigationsleiste.

Nun wird es auch endlich Zeit für einen ersten Test. Setzen Sie die Eigenschaft **Dock** für das Steuerelement `kundeDataGridView` auf `Fill`, damit es das gesamte Formular ausfüllt. Passen Sie dann die Größe des Formulars an, bis alle Spalten vollständig angezeigt werden können. Speichern Sie die Änderungen und lassen Sie das Programm ausführen.

Bitte beachten Sie unbedingt:

Beim Ausführen des Programms wird automatisch eine weitere Kopie der Datenbankdatei in dem Ordner mit dem ausführbaren Programm erstellt. In dieser Kopie werden dann auch die Datensätze bearbeitet.

Die Kopien im Ordner mit dem ausführbaren Programm werden allerdings ohne Rückfrage überschrieben und bereits erfasste Daten gehen verloren. Wenn Sie keine Kopien im Ordner mit dem ausführbaren Programm erstellen lassen wollen, markieren Sie die Datenbankdatei **auto2030.mdf** im Projektmappen-Explorer und setzen Sie die Eigenschaft **In Ausgabeverzeichnis kopieren** im Eigenschaftenfenster auf **Nicht kopieren**.

Sie müssen dann allerdings die Datenbankdatei **auto2030.mdf** selbst aus dem Ordner mit der ausführbaren Datei^{a)} in den eigentlichen Entwicklungsordner kopieren und auch nach jeder Änderung wieder vor dem Ausführen der Anwendung in den Ordner mit der ausführbaren Datei kopieren. Und das gerät schnell in Vergessenheit beziehungsweise es entsteht sehr schnell großes Durcheinander.

Alternativ können Sie auch Kopien erstellen lassen und die Datenbankdateien aus dem Ordner mit der ausführbaren Anwendung nach dem Erfassen von Daten in den eigentlichen Entwicklungsordner zurückkopieren. Aber auch hier besteht die Gefahr, dass Sie das Kopieren vergessen beziehungsweise aus Versehen Änderungen wieder überschreiben.

Wenn Ihnen der Aufwand zu groß ist, können Sie auch auf das Einbinden der Datenbankdateien in das Projekt verzichten und einen festen Pfad benutzen. Dann müssen Sie allerdings sicherstellen, dass sich die Datenbankdateien dort auch in jedem Fall befinden.

In unserem Beispielprojekt, das Sie auf der Online-Lernplattform finden, werden die Datenbankdateien kopiert, wenn Änderungen erfolgt sind. Damit ist in jedem Fall sichergestellt, dass der Zugriff funktioniert und immer die aktuellste Version benutzt wird.

a) Dieser Ordner befindet sich in der Regel im Unterordner `\bin` des Projekts und trägt den Namen **Debug** beziehungsweise **Release**.

Lassen Sie anschließend das Formular mit der Listenansicht für die Kunden anzeigen. Es sollte ungefähr so aussehen:



Abb. 3.11: Das Formular für die Listenansicht der Kunden

In die einzelnen Felder können Sie nun Daten eingeben. In das nächste Feld gelangen Sie dabei entweder mit der Taste oder durch einen Mausklick. Lediglich das Feld **kNummer** ganz links in der Tabelle ist gegen Eingaben gesperrt, da die Kundennummer ja automatisch vergeben wird.

Probieren Sie die Dateneingabe jetzt einmal aus. Klicken Sie dann auf das Symbol **Daten speichern** ganz rechts in der Symbolleiste des Formulars.



Abb. 3.12: Der erste gespeicherte Datensatz

Bitte beachten Sie:

Die Daten werden nicht automatisch gesichert – auch dann nicht, wenn Sie eine neue Zeile anlegen oder das Formular schließen. Sie müssen das Speichern immer selbst durchführen.

Achten Sie vor dem Speichern darauf, dass Sie für alle Felder außer den Telefonnummern Daten erfassen müssen. Außerdem darf die Postleitzahl nicht länger sein als fünf Stellen. Andernfalls erzeugt das Programm nämlich beim Speichern eine Ausnahme. Wir werden uns gleich noch darum kümmern, dass die Daten bei der Eingabe automatisch überprüft werden.

Beim Löschen von Datensätzen über das Symbol **Löschen** wird zunächst nur der Datensatz aus der Anzeige entfernt. Um die Änderungen in die Datenbanktabelle zu übernehmen, müssen Sie nach dem Löschen noch auf das Symbol **Daten speichern** klicken.

Erfassen Sie jetzt noch einige weitere Datensätze und experimentieren Sie dann ein wenig mit der Tabelle. So können Sie die Daten zum Beispiel durch einen Mausklick in die Spaltenköpfe auf- und absteigend sortieren lassen oder die Breite der einzelnen Spalten mit der Maus nahezu beliebig verändern. Testen Sie auch einmal die Funktionen zum Navigieren in der Symbolleiste. Alles sollte problemlos funktionieren – obwohl Sie bisher noch keine einzige Zeile programmiert haben.

Exkurs: Eine DataGridView manuell mit einer Datenquelle verbinden

Sie können eine DataGridView auch selbst erstellen und mit einer Datenquelle verbinden.

Legen Sie dazu zunächst ein Steuerelement **DataGridview** im Formular ab. Sie finden es in der Toolbox in der Gruppe **Daten**. Danach stellen Sie dann die Verbindung zur gewünschten Datenquelle und zur gewünschten Tabelle her. Dazu benutzen Sie entweder das Kombinationsfeld **Datenquelle auswählen** bei den DataGridView-Aufgaben oder die Eigenschaft **DataSource^{a)}** in der Gruppe **Daten** im Eigenschaftenfenster. Dabei werden auch die Spalten aus der Datenquelle automatisch in die DataGridView eingefügt.

Abschließend müssen Sie die Navigationsleiste noch selbst über das Steuerelement **BindingNavigator** einfügen und es über die Eigenschaft **BindingSource** mit der DataGridView verbinden.

Sie sehen selbst, welche Arbeiten Ihnen Visual Studio beim automatischen Einfügen abnimmt.

a) **DataSource** bedeutet übersetzt so viel wie „Datenquelle“.

Zusammenfassung

Der Zugriff auf Daten in einer Datenbanktabelle erfolgt nicht direkt, sondern über spezielle Datenzugriffsschnittstellen und Datenquellen.

Das Anlegen einer neuen Datenquelle erfolgt sehr komfortabel über einen Assistenten von Visual Studio. Diesen Assistenten starten Sie über die Funktion **Projekt/Neue Datenquelle hinzufügen...**

Über das Steuerelement **DataGridView** können Sie Daten aus einer Datenbanktabelle anzeigen und bearbeiten. Ziehen Sie dazu die gewünschte Tabelle aus der Ansicht der Datenquellen in das Formular. Visual Studio erstellt dann alle erforderlichen Steuerelemente und legt auch verschiedene Dateien automatisch an.

Die Daten, die Sie in einem DataGridView erfassen, werden nicht automatisch gespeichert. Sie müssen alle Änderungen selbst über das Symbol **Daten speichern**  sichern.

Sie können das Steuerelement **DataGridView** auch selbst in ein Formular einfügen und dann per Hand über die Eigenschaft **DataSource** mit der Datenquelle verbinden.

Aufgaben zur Selbstüberprüfung

- 3.1 Was ist ADO.NET?

- 3.2 Sie wollen in einem Steuerelement Daten aus einer Datenbanktabelle anzeigen lassen. Womit verbinden Sie das Steuerelement?

- 3.3 Sie wollen für eine Tabelle **versuch** in der Datenbankdatei **test.mdf** eine Datenquelle erstellen. Beschreiben Sie die dazu erforderlichen Schritte.

- 3.4 Wie viele Steuerelemente werden in einem Formular automatisch erzeugt, wenn Sie eine Datenbanktabelle aus der Ansicht der Datenquellen in das Formular ziehen? Welche Bedeutung haben diese Steuerelemente?

4 Die Dateneingabe

In diesem Kapitel beschäftigen wir uns intensiver mit der Dateneingabe in das Formular. Wir sorgen unter anderem dafür, dass keine ungültigen Daten eingegeben werden können und dass alle Felder, für die kein Nullwert zugelassen ist, gefüllt werden.

Bisher ist die Eingabe beziehungsweise das Speichern der Daten in unserem Formular noch recht „wackelig“. Wenn Sie zum Beispiel keinen Eintrag für die Spalte **kName** machen und dann versuchen, den Datensatz zu speichern, löst das Programm eine Ausnahme aus. Gleicher geschieht, wenn Sie zum Beispiel versuchen, eine Postleitzahl mit mehr als fünf Stellen zu speichern. Auch hier wird eine Ausnahme ausgelöst, da die entsprechende Spalte in der Datenbanktabelle lediglich fünf Zeichen aufnehmen kann.

Damit das Programm stabiler wird, müssen wir folgende mögliche Bedienfehler abfangen:

- Die Eingabe in die Felder darf nicht länger werden als die maximale Länge der entsprechenden Spalte in der Datenbanktabelle.
- Für Spalten in der Datenbanktabelle, die keinen Nullwert zulassen, muss auch im entsprechenden Feld des DataGridView-Steuerelements ein Eintrag gemacht werden.
- Die Postleitzahl muss exakt fünf Stellen haben. Es dürfen weder mehr noch weniger Stellen eingegeben werden. Außerdem müssen wir überprüfen, ob für die Postleitzahl nur Ziffern verwendet wurden.

Sehen wir uns der Reihe nach an, wie Sie die Probleme angehen können.

4.1 Festlegen der maximalen Eingabelänge

Der erste kritische Punkt – die Länge der Eingabe – lässt sich recht einfach beheben, und zwar über die Spalteneigenschaften des DataGridView-Steuerelements. Hier können Sie nämlich festlegen, wie lang eine Eingabe in einer Spalte maximal sein darf. Wenn diese maximale Länge erreicht ist, werden keine weiteren Eingaben mehr zugelassen.

Markieren Sie bitte das DataGridView-Steuerelement im Formular **Kundenliste**. Stellen Sie dann die Einfügemarke in das Feld für die Eigenschaft **Columns** und klicken Sie auf das Symbol . Sie finden die Eigenschaft **Columns** in der Gruppe **Sonstiges** im Eigenschaftenfenster.

Tipp:

Alternativ können Sie die Spalten auch über den Eintrag **Spalten bearbeiten...** bei den DataGridView-Aufgaben bearbeiten.

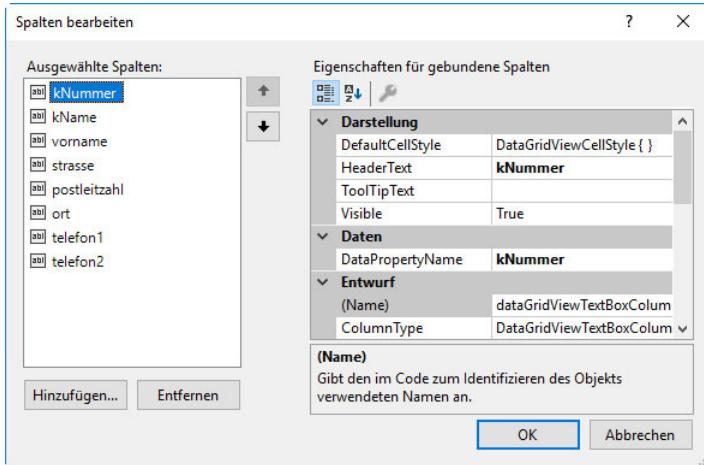


Abb. 4.1: Der Dialog Spalten bearbeiten

Im Dialog Spalten bearbeiten können Sie jetzt die Eigenschaften jeder einzelnen Spalte im Detail festlegen. Die aktuelle Spalte erkennen Sie dabei an der farbigen Hinterlegung im Feld Ausgewählte Spalten.

Die maximale Länge der Eingabe für eine Spalte wird über die Eigenschaft MaxInputLength⁵ gesteuert. Sie finden diese Eigenschaft unten in der Gruppe Verhalten.

Übernehmen Sie jetzt bitte die maximalen Längen für die Spalten aus der folgenden Tabelle. Schließen Sie dann den Dialog Spalten bearbeiten mit einem Mausklick auf OK.

Tab. 4.1: Die maximalen Eingabelängen

Spalte	maximale Eingabelänge
kNummer	beliebig, da der Wert in der Spalte nicht vom Anwender geändert werden kann
kName	50
vorname	50
strasse	50
postleitzahl	5
ort	50
telefon1	20
telefon2	20

Testen Sie dann die Änderungen. Bei der Postleitzahl sollten sich jetzt zum Beispiel nur noch maximal fünf Zeichen eingeben lassen.

Im nächsten Schritt kümmern wir uns darum, dass für die Spalten, die keinen Nullwert zulassen, auch tatsächlich Daten eingegeben werden müssen.

5. MaxInputLength lässt sich mit „maximale Eingabelänge“ übersetzen.

4.2 Abfangen von leeren Eingaben

Das Abfangen der leeren Eingaben ist nicht ganz so einfach wie das Festlegen der maximalen Länge. Denn es gibt leider keine Eigenschaft für die Spalten einer DataGridView, mit der Sie eine Mindestlänge vorgeben können. Wir müssen daher selbst prüfen, ob für eine Spalte eine Eingabe gemacht wurde.

Dazu verwenden wir das Ereignis **CellValidating** des DataGridView-Steuerelements. Es tritt ein, wenn das Bearbeiten einer Zelle beendet werden soll – also zum Beispiel beim Verlassen einer Zelle. In dem Ereignis kontrollieren wir, ob der Inhalt der Zelle leer ist, brechen dann gegebenenfalls die weitere Verarbeitung ab und lassen die Eingabe wiederholen. Da das Ereignis **CellValidating** eintritt, bevor die Eingabe in die Zelle übernommen wird, können wir so sicherstellen, dass die Zelle nicht mehr leer bleibt.

Hinweis:

Es gibt zwei sehr ähnliche Ereignisse für das Überprüfen des Zellinhals bei einem DataGridView-Steuerelement: **CellValidating** und **CellValidated**. Das Ereignis **CellValidating** tritt beim Überprüfen des Inhalts ein und das Ereignis **CellValidated** nach der Überprüfung. Den Unterschied können Sie sich sehr einfach an der Übersetzung merken. **CellValidating** bedeutet so viel wie „beim Überprüfen der Zelle“, **CellValidated** dagegen „Zelle überprüft“.

Übernehmen Sie jetzt bitte den Quelltext aus dem folgenden Code für das Ereignis **CellValidating**. Sie finden das Ereignis in der Gruppe **Fokus** des Eigenschaftenfensters. Was in dem Code genau geschieht, erklären wir Ihnen im Anschluss.



Bitte beachten Sie:

Im Quelltext für die Klasse `Kundenliste` finden Sie zwei Methoden, die Visual Studio automatisch angelegt hat. Die Methode `KundeBindingNavigatorSaveItem_Click()` wird beim Anklicken des Symbols **Daten speichern** ausgeführt und aktualisiert die Daten in der Tabelle **Kunde**. Die Methode `KundenListe_Load()` sorgt dafür, dass die Daten beim Anzeigen des Formulars in die Liste geladen werden. Achten Sie beim Erfassen Ihrer eigenen Quelltexte bitte darauf, dass Sie diese beiden Methoden nicht versehentlich verändern oder überschreiben.

```
//überprüft werden nur die Spalten 2 bis 6
//der Index beginnt bei 0!
if ((e.ColumnIndex > 0) && (e.ColumnIndex < 6))
{
    //ist der Eintrag leer?
    if (e.FormattedValue.ToString() == string.Empty)
    {
        //eine Meldung anzeigen
        //bitte jeweils in einer Zeile eingeben
        MessageBox.Show("Sie müssen einen Wert für " +
            kundeDataGridView.Columns[e.ColumnIndex].HeaderText +
            " eingeben.");
        //den Fehlertext setzen
        kundeDataGridView.Rows[e.RowIndex].Cells[e.ColumnIndex].
        ErrorText = "Die Zelle darf nicht leer sein!";
    }
}
```

```
//die Aktion abbrechen
e.Cancel = true;
}
```

Code 4.1: Das Überprüfen auf leere Einträge in einer Zelle

Schauen wir uns die einzelnen Zeilen der Reihe nach an.

Mit der Anweisung

```
if ((e.ColumnIndex > 0) && (e.ColumnIndex < 6))
```

überprüfen wir über den Ausdruck `e.ColumnIndex6`, ob die aktuelle Zelle in den Spalten zwei bis sechs des DataGridView-Steuerelements liegt. Denn sowohl für die erste Spalte – die Kundennummer – als auch für die letzten Spalten – die Telefonnummern – müssen wir ja keine Kontrolle durchführen. Die Kundennummer kann vom Anwender gar nicht verändert werden, und die Telefonnummern dürfen auch leer bleiben.

Bitte beachten Sie:

Die Nummerierung der Spalten beginnt bei 0 und nicht bei 1. Die erste Spalte hat also den Index 0.



Danach vergleichen wir den Wert, der sich aktuell in der Zelle befindet, mit einer leeren Zeichenkette. Das übernimmt die Anweisung

```
if (e.FormattedValue.ToString() == string.Empty)
```

Hinweis:

Der Ausdruck `e.FormattedValue` liefert den Wert, der in die Zelle gestellt würde, wenn die Eingabe abgeschlossen wird. Auf den eigentlichen Wert der Zelle können Sie in dem Ereignis **CellValidating** noch nicht zugreifen, da das Ereignis eintritt, bevor der Wert endgültig in die Zelle geschrieben wird.

Die Eigenschaft `FormattedValue` ist vom Typ `object`. Sie müssen den Wert daher vor dem Vergleich mit `string.Empty` in eine Zeichenkette umwandeln.

Wenn die Zelle leer ist, erzeugen wir mit der Anweisung

```
MessageBox.Show("Sie müssen einen Wert für " +
kundeDataGridView.Columns[e.ColumnIndex].HeaderText +
" eingeben.");
```

eine MessageBox mit einem Hinweis. Damit der Anwender sofort weiß, um welche Spalte es geht, lassen wir in dem Fenster zusätzlich noch über den Ausdruck `kundeDataGridView.Columns[e.ColumnIndex].HeaderText7` den Text aus dem Kopf der aktuellen Spalte ausgeben.

6. Zur Erinnerung: `e` wird automatisch als Argument an die Methode übergeben. Übersetzt bedeutet `ColumnIndex` so viel wie Spaltenindex.

7. `HeaderText` bedeutet übersetzt so viel wie „Kopftext“.

Anschließend setzen wir noch über die Anweisung

```
kundeDataGridView.Rows[e.RowIndex].Cells[e.ColumnIndex].  
ErrorText = "Die Zelle darf nicht leer sein!";
```

einen Fehlerhinweis für die Zelle. Dieser Fehlerhinweis erscheint als Ausrufezeichen in einem roten Kreis ganz rechts in der jeweiligen Zelle.

Danach brechen wir dann die Aktion über

```
e.Cancel = true;
```

ab.⁸ Diese Anweisung führt dazu, dass die Eingabe verworfen wird und noch einmal neu beginnt. Da auch die neue Eingabe wieder durch unsere Methode überprüft wird, kann der Anwender die Zelle erst dann verlassen, wenn die Eingabe nicht leer ist.

Testen Sie jetzt bitte die Erweiterungen. Speichern Sie die Änderungen und lassen Sie das Programm ausführen. Klicken Sie dann zum Beispiel mit der Maus in die Spalte **kName** in der leeren Zeile unten in der Tabelle und versuchen Sie, die Zelle wieder zu verlassen. Sie werden sehen, zuerst erscheint die Meldung und danach wird das Ausrufezeichen hinten in der Zelle angezeigt. Wenn Sie den Mauszeiger auf das Symbol für den Fehlerhinweis stellen, erscheint auch der Text „Die Zelle darf nicht leer sein!“

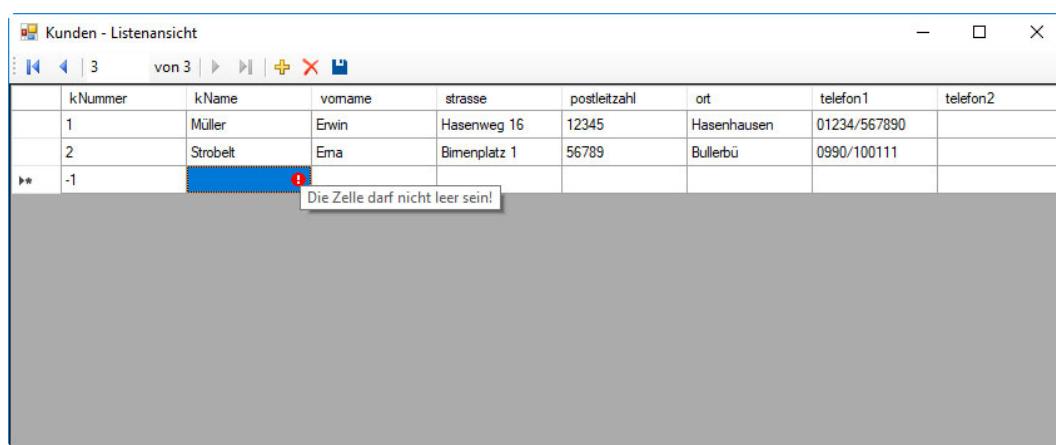


Abb. 4.2: Der Fehlerhinweis bei einer leeren Zelle

Da der Fehlerhinweis nicht automatisch wieder zurückgesetzt wird, bleibt er allerdings auch dann noch weiter in der Zelle stehen, wenn die Eingabe korrekt war. Wir setzen deshalb die Eigenschaft `ErrorText`⁹ für die aktuelle Zelle im Ereignis `CellEndEdit`¹⁰ auf eine leere Zeichenkette. Dadurch verschwindet der Fehlerhinweis wieder, wenn die Eingabe erfolgreich abgeschlossen wurde. Die vollständige Methode für das Ereignis `CellEndEdit` finden Sie im Code 4.2.

8. `Cancel` bedeutet so viel wie „abbrechen“.

9. `ErrorText` bedeutet übersetzt „Fehlertext“.

10. `CellEndEdit` lässt sich mit „Bearbeiten der Zelle beendet“ übersetzen.

```
private void KundeDataGridView_CellEndEdit(object sender,
DataGridViewCellEventArgs e)
{
    //bitte jeweils in einer Zeile eingeben
    if (kundeDataGridView.Rows[e.RowIndex].Cells[e.ColumnIndex].
        ErrorText != string.Empty)
        kundeDataGridView.Rows[e.RowIndex].Cells[e.ColumnIndex].
        ErrorText = string.Empty;
}
```

Code 4.2: Die Methode KundeDataGridView_CellEndEdit()**Hinweis:**

Ob Sie in der Methode zuerst überprüfen, ob die Eigenschaft `ErrorText` nicht leer ist, ist Geschmackssache. Sie können die Abfrage auch weglassen und die Eigenschaft `ErrorText` in jedem Fall auf eine leere Zeichenkette setzen.

Das Ereignis **CellEndEdit** finden Sie im Eigenschaftenfenster in der Gruppe **Daten**.

Bitte beachten Sie, dass wir bisher lediglich überprüfen, ob die aktuelle Zelle nicht leer ist, und nicht kontrollieren, ob alle Felder, die keinen Nullwert haben dürfen, gefüllt sind. Wenn Sie zum Beispiel für einen neuen Datensatz nur den Namen eingeben und dann auf das Symbol **Daten speichern**  klicken, wird nach wie vor eine Ausnahme ausgelöst. Um dieses Problem kümmern wir uns gleich.

Jetzt wollen wir erst einmal sicherstellen, dass die Postleitzahl im korrekten Format eingegeben wird.

4.3 Überprüfung der Postleitzahl

Bei der Prüfung der Postleitzahl müssen wir zwei Punkte berücksichtigen:

1. Es müssen exakt fünf Stellen eingegeben worden sein.
2. Es dürfen nur Ziffern benutzt werden.

Die Überprüfung auf die Länge ist einfach. Wir fragen im Ereignis **CellValidating** erst einmal ab, ob gerade die Postleitzahl eingegeben wird, und vergleichen dann die Länge der Eingabe mit 5.

Ob für die Eingabe nur Ziffern verwendet wurden, können wir mit der Methode `TryParse()`¹¹ überprüfen. Diese Methode versucht, eine Zeichenkette in eine Zahl umzuwandeln, und liefert entweder `true`, wenn die Umwandlung gelungen ist, oder `false`, wenn die Umwandlung gescheitert ist. Die Methode erwartet als ersten Parameter die Zeichenkette, die umgewandelt werden soll, und als zweiten Parameter das Ziel, in dem die umgewandelte Zeichenkette gegebenenfalls abgelegt werden soll.

Der Ausdruck

```
Int32.TryParse(e.FormattedValue.ToString(), out int
postleitzahltemp)
```

11. `TryParse` lässt sich mit „versuche und analysiere“ übersetzen.

versucht zum Beispiel, die Zeichenkette `e.FormattedValue.ToString()` in einen `Int32`-Typ umzuwandeln, und legt das Ergebnis in der Variablen `postleitzahltemp` ab, wenn die Umwandlung gelungen ist.



Bitte beachten Sie:

Sie müssen vor den Bezeichner der Variable, in der das Ergebnis abgelegt werden soll, ausdrücklich das Schlüsselwort `out` setzen. Andernfalls erkennt der Compiler nicht, wo er das Ergebnis speichern soll.

Da wir die beiden Prüfungen für die Postleitzahl nur dann durchführen müssen, wenn überhaupt eine Eingabe in der entsprechenden Zelle vorliegt, benutzen wir in der Methode für das Ereignis `CellValidating` noch eine weitere Variable vom Typ `bool`, die markiert, ob die Zelle für die Postleitzahl noch leer ist.

Die erweiterte Methode `KundeDataGridView_CellValidating()` könnte dann folgendermaßen aussehen. Die neuen Anweisungen haben wir zur besseren Übersicht fett markiert.

```
private void KundeDataGridView_CellValidating(object sender,
DataGridviewCellValidatingEventArgs e)
{
    //für die zusätzlichen Prüfungen der Postleitzahl
    bool fehlerKeinEintrag = false;
    //überprüft werden nur die Spalten 2 bis 6
    //der Index beginnt bei 0!
    if ((e.ColumnIndex > 0) && (e.ColumnIndex < 6))
    {
        //ist der Eintrag leer?
        if (e.FormattedValue.ToString() == string.Empty)
        {
            //eine Meldung anzeigen
            //bitte jeweils in einer Zeile eingeben
            MessageBox.Show("Sie müssen einen Wert für " +
                kundeDataGridView.Columns[e.ColumnIndex].HeaderText +
                " eingeben.");
            //den Fehlertext setzen
            kundeDataGridView.Rows[e.RowIndex].Cells[e.ColumnIndex].
            ErrorText = "Die Zelle darf nicht leer sein!";
            //die Aktion abbrechen
            e.Cancel = true;
            //fehlerKeinEintrag wird true, da die weiteren Prüfungen
            //für die Postleitzahl nicht mehr erforderlich sind
            fehlerKeinEintrag = true;
        }
    }
    //die Detailprüfung für die Postleitzahl
    //sind wir in der Spalte für die Postleitzahl und ist ein
    //Eintrag vorhanden?
    if ((e.ColumnIndex == 4) && (fehlerKeinEintrag == false))
    {
        //ist der Eintrag exakt fünf Zeichen lang?
        //sonst eine Meldung anzeigen und die Verarbeitung abbrechen
        if (e.FormattedValue.ToString().Length != 5)
```

```
    //bitte in einer Zeile eingeben
    MessageBox.Show("Die Postleitzahl muss 5 Ziffern lang
    sein!");
    e.Cancel = true;
}
//sind es nur Ziffern
else
    //bitte jeweils in einer Zeile eingeben
    if (Int32.TryParse (e.FormattedValue.ToString(), out int
postleitzahltemp) == false)
{
    MessageBox.Show("Das Format der Postleitzahl ist
    ungültig!");
    e.Cancel = true;
}
}
```

Code 4.3: Die erweiterte Methode `KundeDataGridView_CellValidating()`

Übernehmen Sie die Erweiterung aus dem Code in Ihr Projekt und testen Sie dann das Programm noch einmal. Jetzt sollten auch die korrekte Länge und das korrekte Format der Postleitzahl sichergestellt sein.

Hinweis:

Da wir die Variable `postleitzahltemp` nicht weiter verwenden, können Sie die Vereinbarung auch weglassen und einen Platzhalter – eine Ausschussvariable – verwenden. Solch eine Variable wird immer durch den Unterstrich `_` markiert. Die Anweisung würde mit einer Ausschussvariablen also so aussehen:

```
if (Int32.TryParse(e.FormattedValue.ToString(), out int _) ==  
    false)
```

Damit können wir uns um das letzte Problem bei der Eingabe kümmern – das eigentliche Speichern der Daten.

4.4 Das Speichern

Die Ausnahme beim Speichern von unvollständigen Daten lässt sich wieder recht einfach abfangen. Denn das Steuerelement DataGridView kennt ein Ereignis **DataError**¹², das bei solchen Fehlern ausgelöst wird. Hier können Sie dann einen eigenen Hinweis über eine MessageBox anzeigen lassen – zum Beispiel mit der folgenden Anweisung:

```
MessageBox.Show("Beim Speichern ist ein Fehler  
aufgetreten!\nBitte überprüfen Sie Ihre Eingaben.", "Fehler",  
MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Code 4.4: Das Anzeigen einer Fehlermeldung im Ereignis **DataError** für das Steuerelement `kundeDataGridView`

12. **DataError** bedeutet übersetzt „Datenfehler“:

Da das Ereignis **DataError** aber nur dann ausgelöst wird, wenn gerade keine Zelle bearbeitet wird, sollten Sie zusätzlich noch die Anweisungen in der automatisch angelegten Methode `KundeBindingNavigatorSaveItem_Click()` in eine `try-catch`-Konstruktion einbetten. Das könnte dann zum Beispiel so aussehen:

```
try
{
    this.Validate();
    this.kundeBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(this.auto2030DataSet);
}
catch
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Beim Speichern ist ein Fehler
                    aufgetreten!\nBitte überprüfen Sie Ihre Eingaben.", "Fehler",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Code 4.5: Die try-catch-Konstruktion für die Methode `KundeBindingNavigatorSaveItem_Click()` (die neuen Anweisungen sind fett markiert)

Jetzt versucht das Programm zunächst nur, das Bearbeiten zu beenden und die Daten zu aktualisieren. Wenn dabei eine Ausnahme auftritt, erscheint wieder eine entsprechende Meldung.

Erstellen Sie bitte die Methode für das Ereignis **DataError** des Steuerelements `kundeDataGridView` und lassen Sie dort eine eigene Nachricht anzeigen. Sie finden das Ereignis in der Gruppe **Verhalten** im Eigenschaftenfenster. Ergänzen Sie dann auch die neuen Anweisungen aus dem vorigen Code. Speichern Sie anschließend die Änderungen und testen Sie das Programm. Nun werden auch Fehler beim Speichern der Daten sauber abgefangen.

Damit ist die Dateneingabe erst einmal komplett. Jetzt werden wir uns noch ein wenig um den Feinschliff für das Formular kümmern.

4.5 Ein wenig Feinschliff

Beginnen wir mit der Anzeige des Textes in den Spaltenköpfen. Hier wird ja bisher der Name der entsprechenden Spalte aus der Datenbanktabelle angezeigt. Über die Eigenschaft **HeaderText** einer Spalte können Sie aber auch einen beliebigen anderen Text anzeigen lassen.

Öffnen Sie bitte den Dialog **Spalten bearbeiten** für das `DataGridView`-Steuerelement. Klicken Sie dazu auf das Symbol  im Feld der Eigenschaft **Columns** oder wählen Sie den Eintrag **Spalten bearbeiten...** bei den `DataGridView`-Aufgaben. Passen Sie anschließend die Texte in den Spaltenköpfen an. Welche Überschriften Sie dabei verwenden, ist Ihnen freigestellt.

Wenn Sie schon einmal die Eigenschaften der Spalten bearbeiten, können Sie auch gleich einen Tooltip für den Spaltenkopf hinterlegen – zum Beispiel „Klicken Sie hier, um die Anzeige nach der Kundennummer zu sortieren.“. Die entsprechende Eigenschaft finden Sie direkt unterhalb der Eigenschaft **HeaderText** in der Gruppe **Darstellung**.

Tipp:

Das Sortieren ist nicht für alle Spalten wirklich sinnvoll. So ist ja eine Sortierung nach dem Vornamen zum Beispiel eigentlich unnötig. Sie können die Sortierung in solchen Fällen auch abschalten. Setzen Sie dazu die Eigenschaft **SortMode**¹³ in der Gruppe **Verhalten** auf *NotSortable*. Denken Sie dann aber auch daran, dass Sie für solche Spalten keinen Tooltip mit einem Hinweis auf die Sortierung anzeigen lassen sollten.

Im nächsten Schritt „peppen“ wir die Darstellung der Tabelle etwas auf. So können Sie zum Beispiel alle ungeraden Zeilen in anderen Farben darstellen lassen als die geraden Zeilen. Dazu bearbeiten Sie die Eigenschaft **AlternatingRowsDefaultCellStyle**¹⁴ für das DataGridView-Steuerelement über das Symbol . Sie finden diese Eigenschaft in der Gruppe **Darstellung**.

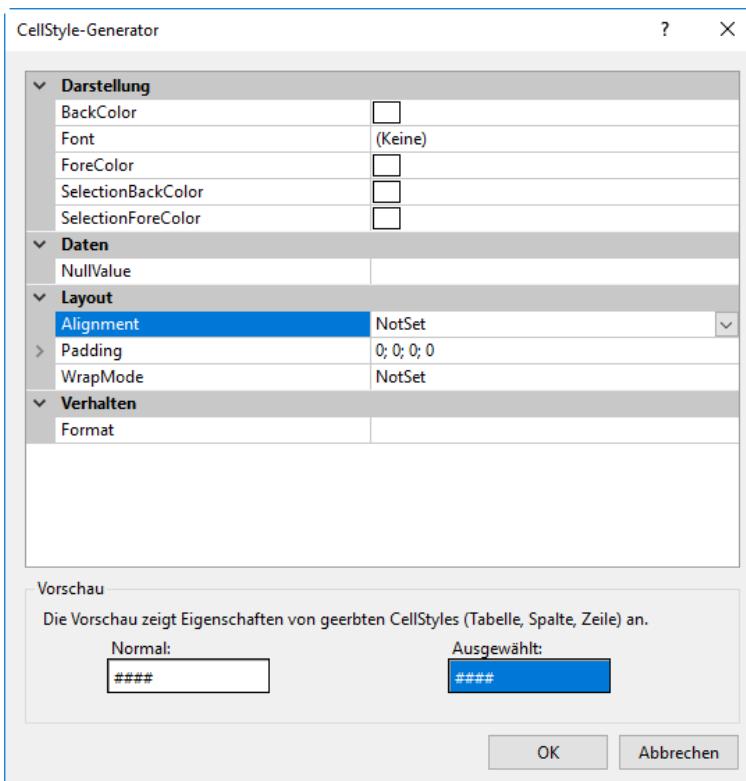


Abb. 4.3: Der CellStyle-Generator

Im Dialog **CellStyle-Generator** können Sie dann zum Beispiel die Hintergrundfarbe (**BackColor**) und die Vordergrundfarbe (**ForeColor**) für die Zeilen setzen. Probieren Sie das einfach einmal aus.

Sie können auch einzelne Spalten in der Anzeige unterschiedlich darstellen lassen. Dazu öffnen Sie den Dialog **Spalten bearbeiten** und markieren die gewünschte Spalte im Feld **Ausgewählte Spalten**. Anschließend öffnen Sie dann den Dialog **CellStyle-Generator** über die Eigenschaft **DefaultCellStyle** in der Gruppe **Darstellung**.

13. SortMode steht für „Sortiermodus“.

14. AlternatingRowsDefaultCellStyle bedeutet so viel wie „Standardzellenstil für abwechselnde Zeilen“.

Ändern Sie jetzt einmal die Darstellung der Spalte für die Kundennummer so, dass der Hintergrund sowohl bei der normalen Darstellung als auch bei der ausgewählten Darstellung immer in der Systemfarbe **InactiveCaption**¹⁵ erscheint. Die entsprechende Farbe finden Sie im Register **System** bei der Farbauswahl. Für die ausgewählte Darstellung setzen Sie die Werte der Eigenschaften **SelectionBackColor** beziehungsweise **BackColor**.

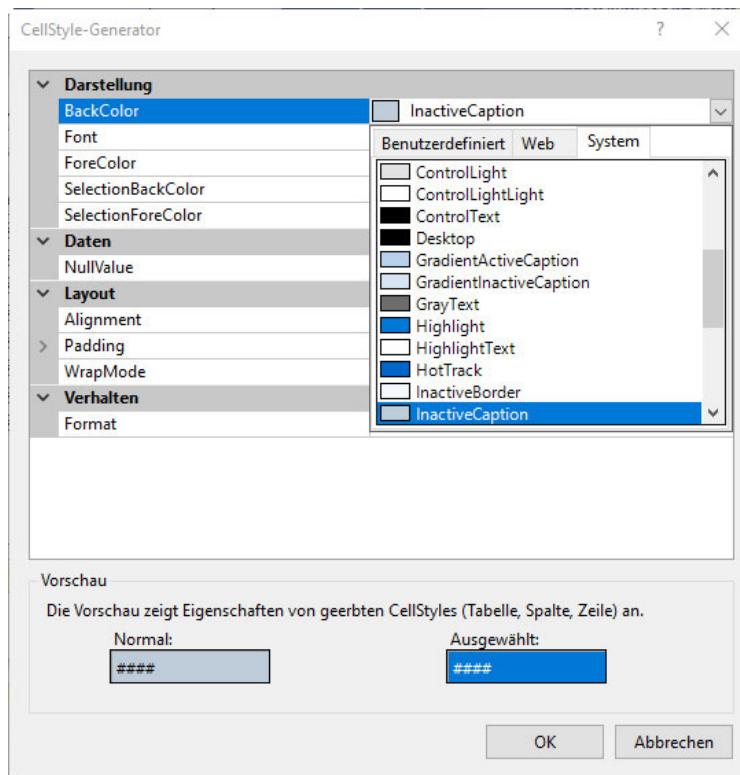


Abb. 4.4: Die Farbauswahl

Die Tabelle mit den unterschiedlich formatierten Zeilen und Spalten könnte nun zum Beispiel so aussehen:

	Kundennummer	Name	Vorname	Strasse	PLZ	Ort	Telefon 1	Telefon 2
1	Müller		Erwin	Hasenweg 16	12345	Hasenhausen	01234/567890	
2	Strobelt		Ema	Birnenplatz 1	56789	Bullerbü	0990/100111	
-1								

Abb. 4.5: Unterschiedlich formatierte Zeilen und Spalten in der Tabelle

15. InactiveCaption bedeutet übersetzt so viel wie „nicht aktiver Titel“.

Abschließend sorgen wir noch dafür, dass der Anwender sofort sehen kann, welche Zelle in der Tabelle aktiv wird, wenn er mit der Maus klickt. Dazu ändern wir im Ereignis **CellMouseEnter** für eine Zelle die Vordergrundfarbe und setzen sie im Ereignis **CellMouseLeave** wieder zurück. Sie finden diese beiden Ereignisse in der Gruppe **Maus** im Eigenschaftenfenster.

Legen Sie zunächst bitte ein Feld `alteFarbe` vom Typ `Color` in der Klasse `Kundenliste` an. Dieses Feld benötigen wir, um die alte Vordergrundfarbe zwischen- speichern zu können.

Übernehmen Sie dann die Anweisungen für die Methoden `KundeDataGridView_CellMouseEnter()` und `KundeDataGridView_CellMouseLeave()` aus dem folgenden Code. Was dort genau geschieht, erklären wir Ihnen im Anschluss.

```
private void KundeDataGridView_CellMouseEnter(object sender,
DataEventArgs e)
{
    //befindet sich die Maus über einer Zelle?
    if ((e.ColumnIndex != -1) && (e.RowIndex != -1))
    {
        //für den leichteren Zugriff
        //bitte in einer Zeile eingeben
        DataGridViewCell zelle = kundeDataGridView.Rows[e.RowIndex].
        Cells[e.ColumnIndex];
        //die alte Farbe sichern
        alteFarbe = zelle.Style.BackColor;
        //die neue Farbe setzen
        zelle.Style.BackColor = Color.LightBlue;
    }
}

private void KundeDataGridView_CellMouseLeave(object sender,
DataEventArgs e)
{
    //befindet sich die Maus über einer Zelle?
    if ((e.ColumnIndex != -1) && (e.RowIndex != -1))
    {
        //für den leichteren Zugriff
        //bitte in einer Zeile eingeben
        DataGridViewCell zelle = kundeDataGridView.Rows[e.RowIndex].
        Cells[e.ColumnIndex];
        //die alte Farbe wiederherstellen
        zelle.Style.BackColor = alteFarbe;
    }
}
```

Code 4.6: Die Methoden `KundeDataGridView_CellMouseEnter()` und `KundeDataGridView_CellMouseLeave()`

Zunächst einmal überprüfen wir mit der Anweisung

```
if ((e.ColumnIndex != -1) && (e.RowIndex != -1))
```

ob sich die Maus nicht gerade über einem Spalten- oder Zeilenkopf befindet. In diesem Fall liefern die Ausdrücke `e.ColumnIndex` beziehungsweise `e.RowIndex` den Wert -1.

Danach erzeugen wir eine Variable `zelle` vom Typ `DataGridViewCell` und weisen dieser Variablen die aktuelle Zelle zu. Das übernimmt die Zeile

```
DataGridViewCell zelle = kundeDataGridView.Rows[e.RowIndex].  
Cells[e.ColumnIndex];
```

Hinweis:

Diese Zuweisung vereinfacht lediglich den Zugriff auf die Zelle. Sie ist nicht zwingend erforderlich. Sie können stattdessen auch immer den Ausdruck `kundeDataGridView.Rows[e.RowIndex].Cells[e.ColumnIndex]` in den folgenden Anweisungen benutzen.

Danach speichern wir den Wert der Eigenschaft `Style.BackColor` der aktuellen Zelle in unserem Feld `alteFarbe` zwischen und setzen anschließend die Hintergrundfarbe auf ein helles Blau. Das erledigen die beiden Anweisungen

```
alteFarbe = zelle.Style.BackColor;  
zelle.Style.BackColor = Color.LightBlue;
```

In der Methode `KundeDataGridView_CellMouseLeave()` setzen wir dann mit einer ähnlichen Technik die Hintergrundfarbe der Zelle wieder auf den zwischengespeicherten Wert aus dem Feld `alteFarbe`.

Damit wollen wir das Feintuning für die Tabelle beenden. Wenn Sie wollen, können Sie ja selbst noch ein wenig aktiv werden und zum Beispiel die Tooltips für die Symbolleiste überarbeiten oder auch noch weitere Gestaltungsmöglichkeiten ausprobieren.

Im nächsten Kapitel werden wir noch ein Formular für die Einzelanzeige von Kundendaten erstellen.

Zusammenfassung

Das Ereignis `CellValidating` tritt ein, wenn die Eingabe in einer Zelle eines DataGridView-Steuerelements abgeschlossen werden soll.

Im Ereignis `CellValidating` liegt erst einmal nur der Wert vor, der beim Beenden der Eingabe in die Zelle gestellt würde. Diesen Wert können Sie über den Ausdruck `e.FormattedValue` verarbeiten. Da es sich um einen allgemeinen Typ `object` handelt, müssen Sie den Wert ausdrücklich konvertieren – zum Beispiel in eine Zeichenkette.

Wenn die Bearbeitung einer Zelle in einem DataGridView-Steuerelement abgeschlossen wird, tritt das Ereignis `CellEndEdit` ein.

Mit der Methode `TryParse()` können Sie überprüfen, ob die Umwandlung in einen anderen Typ möglich ist. Falls die Umwandlung scheitert, liefert die Methode `false` zurück.

Bei Fehlern beim Speichern der Daten in einem DataGridView-Steuerelement wird das Ereignis **DataError** ausgelöst.

Über die Eigenschaft **HeaderText** können Sie den Text in der Spalte eines DataGridView-Steuerelements ändern.

Über den CellStyle-Generator können Sie das Aussehen von Zeilen und Spalten einzeln bearbeiten.

Die Ereignisse **CellMouseEnter** und **CellMouseLeave** treten ein, wenn die Maus in eine Zelle eines DataGridView-Steuerelements bewegt wird und wenn die Maus eine Zelle wieder verlässt. Um nur die eigentlichen Zellen zu berücksichtigen, müssen Sie in den Ereignissen überprüfen, ob sich der Mauszeiger aktuell über einem Spalten- beziehungsweise Zeilenkopf befindet.

Aufgaben zur Selbstüberprüfung

- 4.1 Über welche Eigenschaft legen Sie die maximale Länge der Eingabe in einer Zelle eines DataGridView-Steuerelements fest?

- 4.2 Formulieren Sie bitte einen Ausdruck, mit dem Sie im Ereignis **CellValidating** eines DataGridView-Steuerelements `gridView1` auf die aktuelle Zelle zugreifen können.

- 4.3 Sie wollen im Ereignis **CellValidating** die aktuelle Eingabe eines Anwenders verwerfen und eine neue Eingabe durchführen. Welche Anweisungen verwenden Sie dazu?

- 4.4 Sie möchten überprüfen, ob eine Zeichenkette `zKette1` nur aus Zahlen besteht. Formulieren Sie bitte eine entsprechende Abfrage mit der Methode `TryParse()`.

- 4.5 Sie wollen, dass eine Spalte in einem DataGridView-Steuerelement nicht sortiert werden kann. Welche Eigenschaft müssen Sie dazu auf welchen Wert setzen?

- 4.6 Über welche Eigenschaft eines DataGridView-Steuerelements legen Sie das Aussehen der ungeraden Zeilen fest?

5 Das Formular zur Einzelanzeige

In diesem Kapitel werden wir ein Formular für die Einzelanzeige eines Kunden-datensatzes anlegen.

Legen Sie im ersten Schritt bitte im Formular, das nach dem Start der Anwendung angezeigt wird, eine weitere Schaltfläche in der GroupBox für die Kunden an. Als Text können Sie zum Beispiel **Einzelansicht** verwenden.

Erstellen Sie dann ein neues leeres Formular **Kundeneinzel** für die Einzelansicht und lassen Sie dieses Formular beim Anklicken der Schaltfläche modal anzeigen. Setzen Sie dann noch den Titel für das Formular – zum Beispiel auf **Kunden - Einzelansicht**.

Wechseln Sie anschließend in das neue Formular und öffnen Sie das Register mit den Datenquellen. Markieren Sie die Datenquelle **Kunde** und klicken Sie auf das Symbol . Wählen Sie anschließend im Menü den Eintrag **Details** aus.

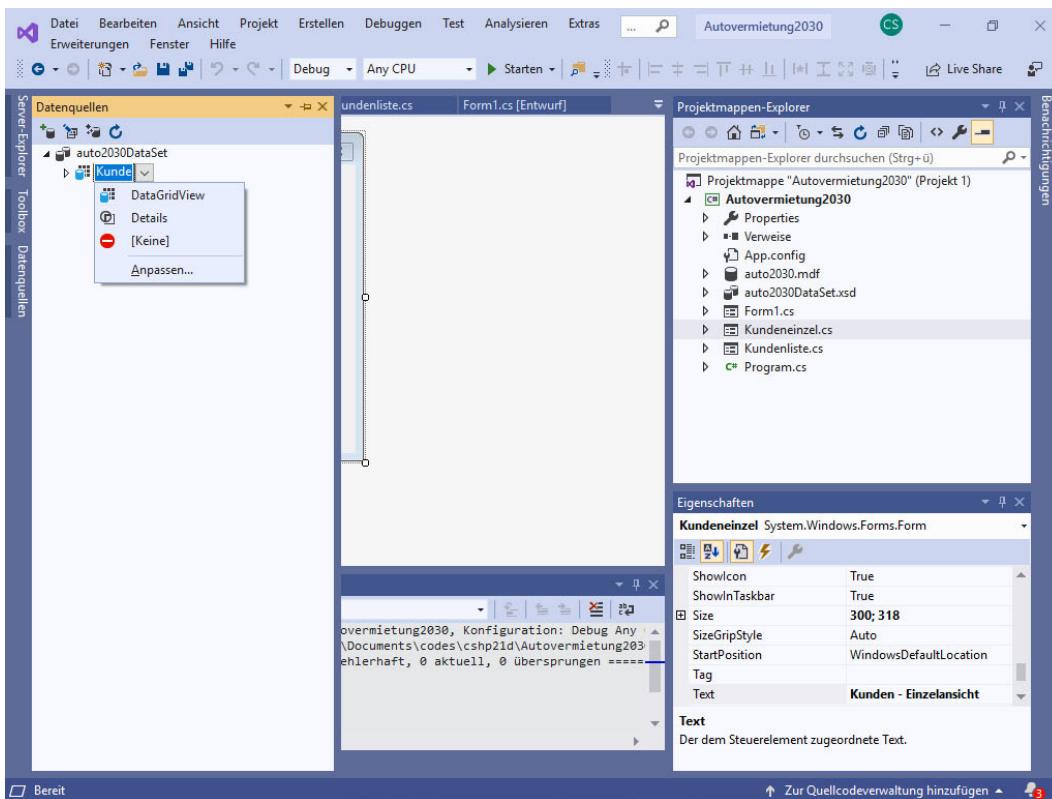


Abb. 5.1: Das Menü für die Auswahl der Darstellung (links oben in der Abbildung)

Ziehen Sie dann die Datenquelle mit gedrückter linker Maustaste in das Formular und lassen Sie die Maustaste los. Visual Studio erstellt einzelne Felder für die Spalten aus der Datenquelle und fügt auch die weiteren erforderlichen Steuerelemente automatisch ein.

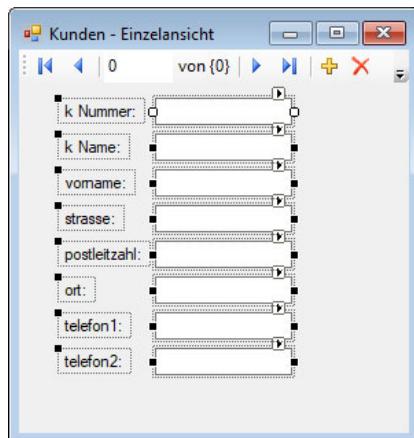


Abb. 5.2: Das Formular zur Einzelanzeige nach dem Einfügen der Datenquelle

Hinweis:

Die Bindung der einzelnen Eingabefelder an die Spalten der Datenbanktabelle erfolgt über die Untereigenschaft **Text** der Eigenschaft (**DataBindings**). Sie finden diese Eigenschaft in der Gruppe **Daten** des Eigenschaftenfensters. Dass der Text aus einer Datenbanktabelle stammt, erkennen Sie auch an dem kleinen Datenbanksymbol rechts hinter der Eigenschaft **Text** für die Eingabefelder.

Ändern Sie jetzt bitte die Labels für die Eingabefelder. Sie können dabei die gleichen Texte benutzen, die Sie auch in den Spaltenköpfen des DataGridView-Steuerelements verwendet haben. Ordnen Sie anschließend die Elemente im Formular neu an und verbreitern Sie auch die Eingabefelder noch ein wenig.

Das Formular sollte danach ungefähr so aussehen wie in der Abb. 5.3.



Abb. 5.3: Das Formular nach den Anpassungen

Speichern Sie dann alle Änderungen und testen Sie das neue Formular.

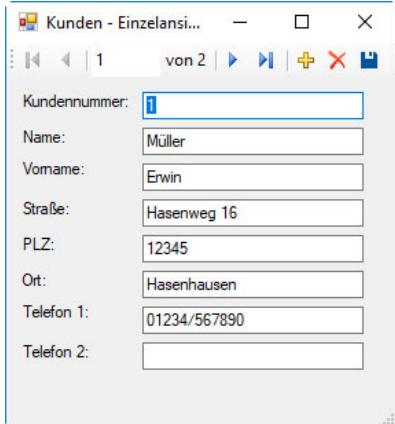


Abb. 5.4: Die Einzelansicht für die Kunden im praktischen Einsatz

Zusammenfassung

Neben einer Listenansicht können Sie aus einer Datenquelle auch automatisch eine Detailansicht erstellen lassen.

Aufgaben zur Selbstüberprüfung

- 5.1 Beschreiben Sie bitte, wie Sie für eine Datenquelle eine Detailansicht in einem Formular erstellen lassen.

- 5.2 Bei der Eigenschaft **Text** eines Eingabefelds wird eine kleine Tonne angezeigt. Wofür steht dieses Symbol?

6 Die Tabelle und die Formulare für die Fahrzeuge

In diesem Kapitel werden wir die Tabelle und die Formulare für die Fahrzeuge erstellen. Da es dabei keine Besonderheiten gibt, stellen wir Ihnen die einzelnen Schritte lediglich im Überblick vor. Details können Sie bei Bedarf in den letzten Kapiteln nachlesen.

Erstellen Sie bitte zunächst im Formular, das direkt nach dem Start angezeigt wird, in der GroupBox für die Fahrzeuge zwei Schaltflächen. Über die eine starten wir gleich die Listenansicht und über die zweite die Einzelansicht.

Erstellen Sie anschließend ein leeres Formular für die Listenansicht und ein leeres Formular für die Einzelansicht. Lassen Sie diese Formulare beim Anklicken der entsprechenden Schaltfläche modal anzeigen.

Hinweis:

Wir verwenden in unserem Projekt die Namen **Fahrzeugeinzel** und **Fahrzeugliste** für die beiden Formulare.

6.1 Das Anlegen der Tabelle

Um die Datenbanktabelle für die Fahrzeuge anzulegen, lassen Sie den Server-Explorer anzeigen und wählen im Kontext-Menü des Eintrags **Tabellen** für die Datenverbindung **auto2030.mdf** die Funktion **Neue Tabelle hinzufügen**.

Bevor Sie weiterlesen:

Überlegen Sie einmal selbst, welche Spalten Sie benötigen. Sehen Sie sich dazu die Abb. 6.1 mit der Tabelle **Fahrzeug** und der Klasse **Fahrzeug** an.

Fahrzeug	
PK	fNummer
	bezeichnung kennzeichen preisProTag typ ausgeliehen

Fahrzeug
-fNummer : int -bezeichnung : string -kennzeichen : string -preisProTag : float -typ : char -ausgeliehen : bool = false +anlegenFahrzeug(): bool +loescheFahrzeug(fNr: int): bool +aendereFahrzeug(fNr: int): bool +erzeugeListen() +sucheFahrzeug(fNr: int): bool +sendeFInfo(fNr: int): fType -erzeugeFNr(): int +setzeAusgeliehen(status: bool) : bool

Abb. 6.1: Die Tabelle für die Fahrzeugdaten (links) und die Klasse **Fahrzeug** (rechts)

Die Spalten könnten zum Beispiel so aussehen wie in der Tab. 6.1.

Tab. 6.1: Die Spalten für die Tabelle der Fahrzeuge

Spaltenname	Datentyp	NULL erlaubt	Länge
fNummer	int	nein	
bezeichnung	nvarchar	nein	50
kennzeichen	nvarchar	nein	12
preisProTag	float	nein	
typ	nvarchar	nein	1
ausgeliehen	bit	nein	

Legen Sie fest, dass die Fahrzeugnummer automatisch vergeben werden soll und bei 1 beginnt. Als Schrittweite können Sie ebenfalls 1 benutzen. Die entsprechenden Eigenschaften finden Sie unten im Formular bei den Spalteneigenschaften.

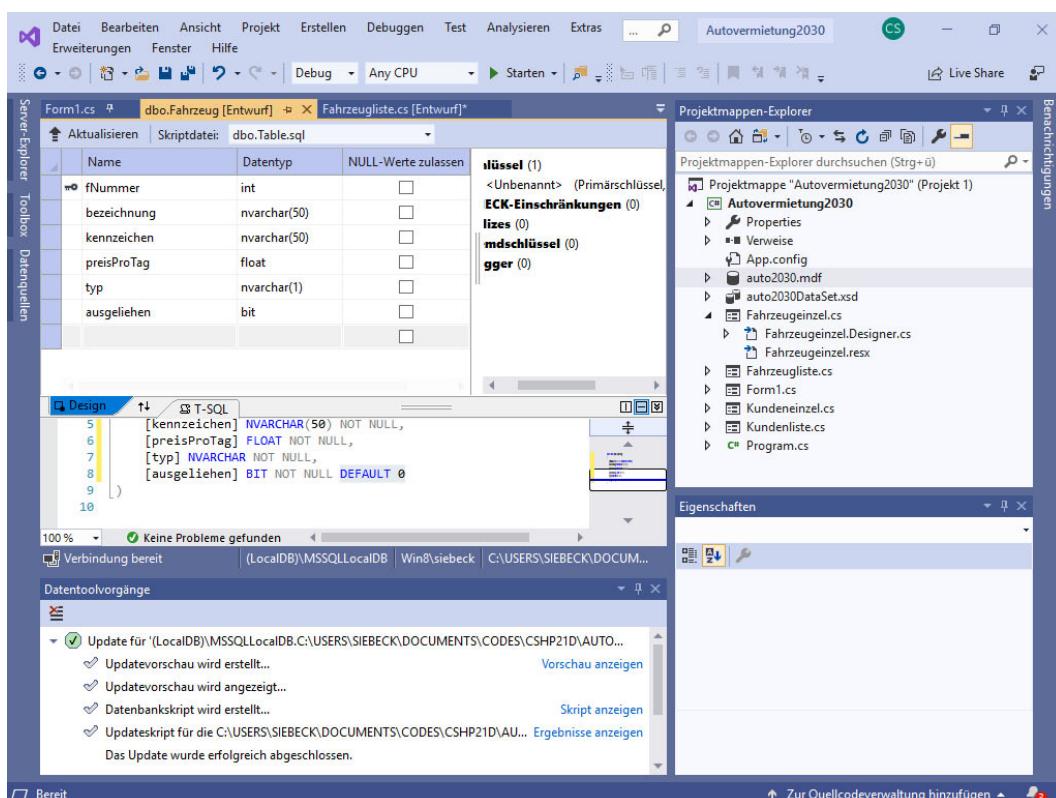
Prüfen Sie außerdem, ob die Spalte **fNummer** als Primärschlüssel vereinbart ist. Ändern Sie den Namen der Tabelle in **Fahrzeug**. Passen Sie dazu den Eintrag [Table] in der Anweisung

```
CREATE TABLE [dbo]. [Table]
```

oben im Register T-SQL entsprechend an. Die Anweisung sollte nach den Änderungen so aussehen:

```
CREATE TABLE [dbo]. [Fahrzeug]
```

Lassen Sie die Datenbank dann aktualisieren.

**Abb. 6.2:** Die Tabelle **Fahrzeug**

Schließen Sie dann den Tabellen-Designer und fügen Sie die neu erstellte Tabelle zu den Datenquellen hinzu. Wechseln Sie dazu in ein beliebiges Formular der Anwendung und lassen Sie die Datenquellen anzeigen. Klicken Sie im Register **Datenquellen** auf das Symbol **Datenquelle mit Assistenten konfigurieren** und markieren Sie im Fenster **Assistent zum Konfigurieren von Datenquellen** im Zweig **Tabellen** zusätzlich die gerade angelegte Tabelle.



Bitte beachten Sie:

Änderungen an der Datenbank führen nicht automatisch auch zu Änderungen an den Datenquellen. Die Datenquellen beziehen sich in der Regel auf einzelne Tabellen einer Datenbank und nicht auf eine vollständige Datenbank.

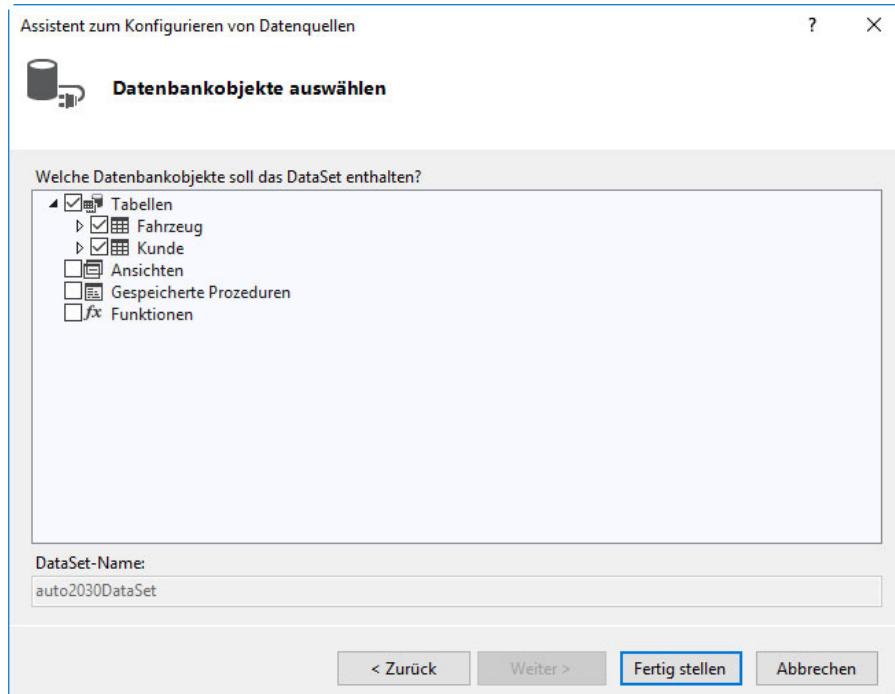


Abb. 6.3: Das Aktualisieren der Datenquelle

Übernehmen Sie abschließend die Änderungen mit einem Klick auf die Schaltfläche **Fertig stellen**. Visual Studio aktualisiert dann die Datenquellen. Das kann einen kurzen Moment dauern.

6.2 Die Formulare

Danach können Sie – wie von der Tabelle **Kunde** gewohnt – die Formulare für die Listenanzeige und die Einzelanzeige der Fahrzeuge erstellen. Da es dabei keinerlei Besonderheiten gibt, stellen wir Ihnen die einzelnen Schritte hier nicht noch einmal vor.

Die Formulare könnten zum Beispiel so aussehen wie in den beiden folgenden Abbildungen.

	Fahrzeugnummer	Bezeichnung	Kennzeichen	Preis pro Tag	Typ	Ausgeliehen
▶	2	Mercedes E 200 ...	BO-ST 123	90	C	<input type="checkbox"/>
	3	Volvo V90	E-S 987	85,8	K	<input type="checkbox"/>
*						<input type="checkbox"/>

Abb. 6.4: Die Listenanzeige für die Fahrzeuge

Fahrzeugnummer:	2
Bezeichnung:	Mercedes E 200 Cabrio
Kennzeichen:	BO-ST 123
Preis pro Tag:	90
Typ:	C
Ausgeliehen:	<input type="checkbox"/>

Abb. 6.5: Die Einzelanzeige für die Fahrzeuge

Denken Sie bitte bei der Listenanzeige für die Fahrzeuge an die Überprüfungen bei der Eingabe. Legen Sie maximale Eingabelängen für die Felder fest und kontrollieren Sie bei der Eingabe und auch beim Speichern, ob alle erforderlichen Felder gefüllt sind beziehungsweise ob Probleme aufgetreten sind. Die Anweisungen für die Prüfungen können Sie aus den Formularen der Kunden übernehmen und anpassen.

Beim Preis pro Tag müssen wir lediglich prüfen, ob die Eingabe in den Typ `Single` umgewandelt werden kann. Die entsprechenden Anweisungen sehen so aus:

```
if (Single.TryParse(preisProTagTextBox.Text, out float preistemp)
== false)
{
    MessageBox.Show("Das Format des Preises ist nicht gültig!");
    e.Cancel = true;
}
```

6.3 Eine Fahrzeugliste

Im nächsten Schritt erstellen wir jetzt noch eine erste, sehr einfache Liste für die Anzeige der Fahrzeuge. Legen Sie dazu ein neues Formular mit einem DataGridView-Steuerelement für die Datenquelle **Fahrzeug** an. Lassen Sie das Steuerelement in der maximalen Größe darstellen, und setzen Sie die Eigenschaft **ReadOnly** für alle Spalten auf `True`. Passen Sie außerdem die Titel für die Spalten an. Löschen Sie dann die drei Symbole und die Trennlinie ganz rechts in der Symbolleiste.

Entfernen Sie anschließend noch bei den DataGridView-Aufgaben die Markierungen bei den Funktionen **Hinzufügen aktivieren**, **Bearbeiten aktivieren** und **Löschen aktivieren**.

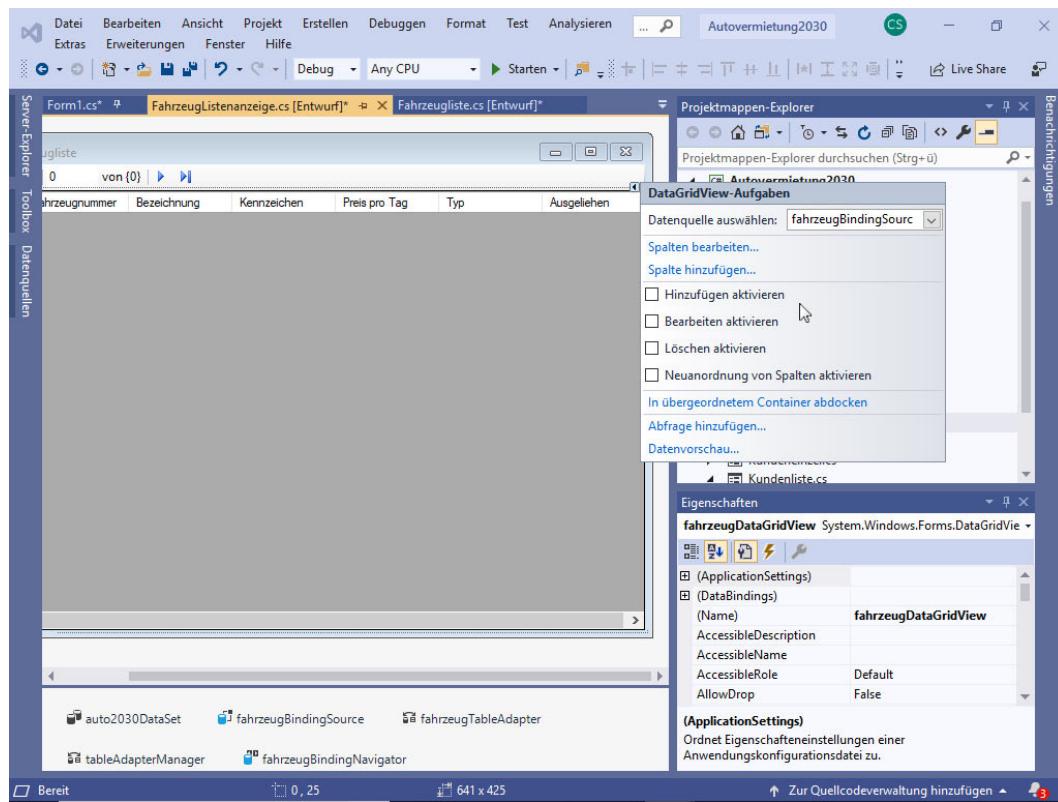


Abb. 6.6: Die deaktivierte Funktionen bei den DataGridView-Aufgaben

Erstellen Sie abschließend noch eine Schaltfläche für die Anzeige der Liste im Formular mit den GroupBoxen. Die fertige Liste könnte dann so aussehen:

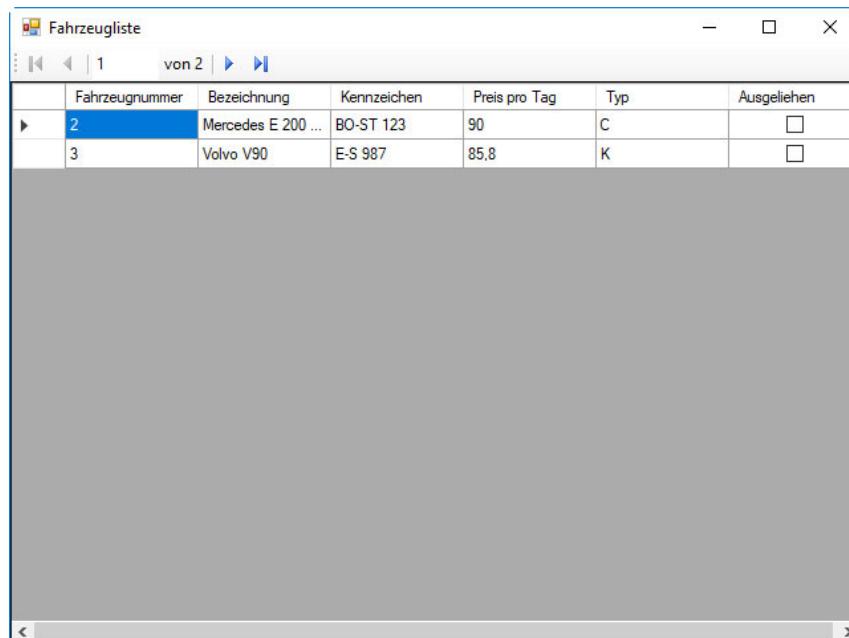


Abb. 6.7: Eine erste einfache Liste der Fahrzeuge

Abschließend sorgen wir noch dafür, dass das Kontrollkästchen für den Status **Ausgeliehen** keinen undefinierten Wert erhält. Das geht recht einfach über den DataSet-Designer. Lassen Sie noch einmal die Datenquellen anzeigen und markieren Sie die Tabelle **Fahrzeug**. Klicken Sie dann auf das Symbol **DataSet mit Designer bearbeiten**.

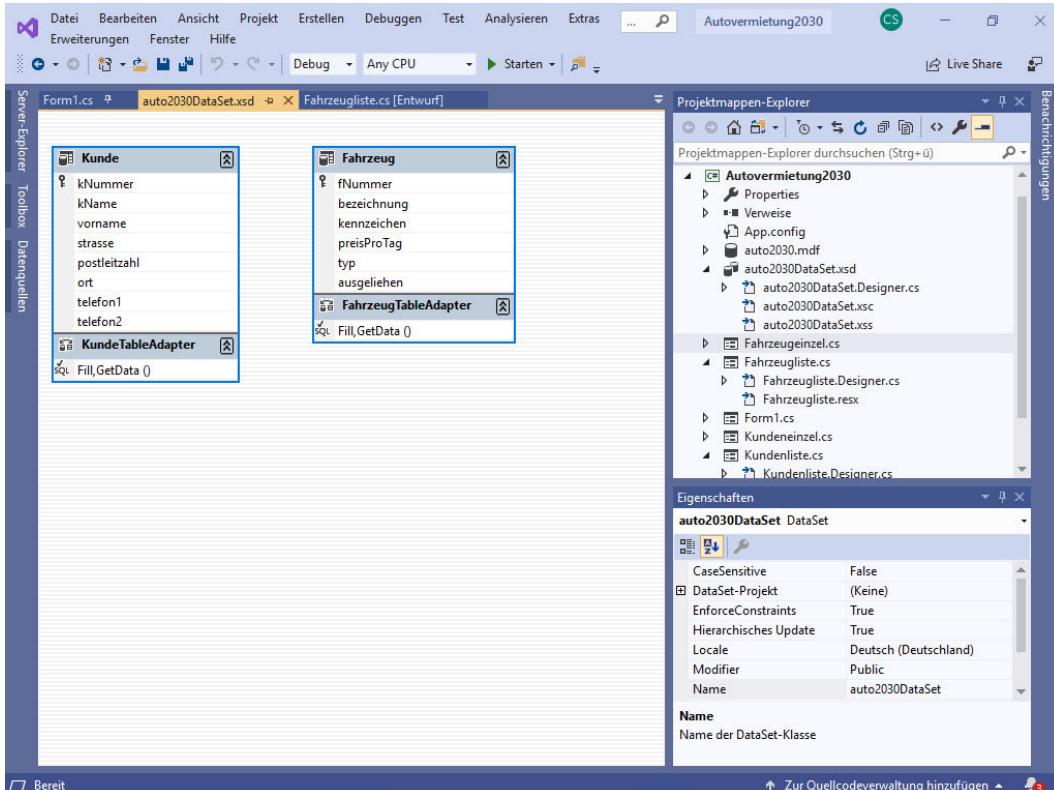


Abb. 6.8: Der DataSet-Designer

Markieren Sie danach in der Tabelle **Fahrzeug** die Spalte **ausgeliehen** durch einen Mausklick und ändern Sie den Wert der Eigenschaft **DefaultValue** in `False`. Speichern Sie anschließend die Änderungen am DataSet.

Hinweis:

Mit dem DataSet-Designer werden wir uns später noch intensiver beschäftigen.

Damit können wir jetzt Kunden und Fahrzeuge anlegen, löschen und ändern. Auch eine erste Liste für die Fahrzeuge ist vorhanden. Eine Erweiterung für die Formulare wartet noch bei den Einsendeaufgaben auf Sie.

Schlussbetrachtung

Herzlichen Glückwunsch! Sie haben einen weiteren wichtigen Schritt geschafft.

Sie wissen jetzt, wie Sie mit den Werkzeugen von Visual Studio Datenbanken erstellen und wie Sie aus Ihren Anwendungen auf diese Datenbanken zugreifen können. Diese Kenntnisse werden Sie später noch weiter vertiefen.

In der Zwischenzeit können Sie ja einmal selbst versuchen, eine kleine Datenbankanwendung zu erstellen – zum Beispiel eine einfache Adressverwaltung, die auch E-Mail-Adressen, Handy- und Faxnummern speichern kann. Die dazu nötigen Kenntnisse haben Sie jetzt. Denken Sie aber auch hier wieder daran, dass eine gute Vorbereitung entscheidend zu einem gelungenen Programm beiträgt. Überlegen Sie also vorher in aller Ruhe, was das Programm können soll und wie Sie die Datenbank aufbauen müssen. Wie Sie in diesem Studienheft selbst gesehen haben, sind Klassendiagramme und Datenbankmodelldiagramme dabei eine wertvolle Hilfe.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 2

- 2.1 Im ersten Schritt blenden Sie den Server-Explorer über die Registerung **Server-Explorer** ein. Im Server-Explorer klicken Sie auf das Symbol **Mit Datenbank verbinden** . Markieren Sie im Dialog **Datenquelle auswählen** den Eintrag **Microsoft SQL Server-Datenbankdatei** und klicken Sie dann auf **Weiter**. Überprüfen Sie im Dialog **Verbindung hinzufügen** noch einmal, ob bei der Datenquelle eine Microsoft SQL Server-Datenbankdatei angegeben ist, und passen Sie den Eintrag – wenn erforderlich – an.

Geben Sie anschließend einen Namen für die Datenbankdatei ein. Bei den Einstellungen für die Anmeldung sind keine Änderungen erforderlich. Klicken Sie abschließend auf **OK** und bestätigen Sie die Abfrage, ob die Datenbankdatei erstellt werden soll.

- 2.2 Um eine neue Tabelle für eine Datenbank zu erstellen, klicken Sie im Server-Explorer mit der rechten Maustaste auf den Eintrag **Tabellen** der Datenbank. Wählen Sie dann im Kontext-Menü den Eintrag **Neue Tabelle hinzufügen**.
- 2.3 Über das Feld **NUL-Werte zulassen** im Tabellen-Designer steuern Sie, ob das Feld auch leer bleiben darf.
- 2.4 Damit die Werte für eine Spalte automatisch vergeben werden, setzen Sie die Eigenschaft (**Ist Identity**) für die Spalte auf **True**. Für den Startwert legen Sie die Eigenschaft **Identitätsseed** auf 10 fest. Das Erhöhen um den Wert 5 steuern Sie über die Eigenschaft **ID-Inkrement**.

Kapitel 3

- 3.1 ADO.NET ist die Datenzugriffsschnittstelle für .NET Framework-Anwendungen. Sie ist Bestandteil des .NET Frameworks.
- 3.2 Um in einem Steuerelement Daten aus einer Datenbanktabelle anzeigen zu lassen, verbinden Sie das Steuerelement mit der Datenquelle, die zu der Datenbanktabelle gehört.
- 3.3 Zuerst starten Sie den Assistenten zum Erstellen einer Datenquelle über die Funktion **Projekt/Neue Datenquelle hinzufügen...** Im ersten Schritt markieren Sie gegebenenfalls den Eintrag **Datenbank** und klicken auf die Schaltfläche **Weiter >**. Im zweiten Schritt legen Sie **DataSet** als Datenbankmodell fest. Im dritten Schritt wählen Sie die Datenverbindung für die Datenbankdatei **test.mdf** aus. Lassen Sie dann gegebenenfalls eine Kopie erstellen und speichern Sie die Verbindungszeichenfolge. Danach zeigt Ihnen der Assistent die vorhandenen Objekte in der Datenbank an. Um die Tabelle **versuch** auszuwählen, öffnen Sie den Zweig **Tabellen** und markieren das Kontrollkästchen vor dem Eintrag **versuch**. Abschließend klicken Sie auf die Schaltfläche **Fertig stellen**.

- 3.4 Es werden insgesamt fünf verschiedene Steuerelemente erzeugt. Eins davon ist die eigentliche Tabelle zum Anzeigen und Bearbeiten der Daten – die DataGridView-View. Ein Steuerelement steht für die Datenbank und ein weiteres für die Datenbanktabelle. Außerdem werden noch ein Steuerelement für die Kommunikation zwischen den Steuerelementen und der Datenbank sowie ein Steuerelement für die Navigation eingefügt.

Kapitel 4

- 4.1 Die maximale Länge der Eingabe in einer Zelle eines DataGridView-Steuerelements wird über die Eigenschaft **MaxInputLength** gesteuert.

- 4.2 Um im Ereignis **CellValidating** auf die aktuelle Zelle im Steuerelement `gridView1` zuzugreifen, können Sie zum Beispiel folgenden Ausdruck verwenden:

```
gridView1.Rows[e.RowIndex].Cells[e.ColumnIndex]
```

- 4.3 Um die Eingabe zu verwerfen und erneut durchzuführen, benutzen Sie die Anweisung `e.Cancel = true;`.

- 4.4 Die Abfrage könnte zum Beispiel so aussehen:

```
if (Int32.TryParse(zKette1, out int zahl) == true)  
    beziehungsweise if (Int32.TryParse(zKette1, out int zahl) !=  
    false)
```

Der Bezeichner für das Ziel der Umwandlung ist beliebig. Wichtig ist die ausdrückliche Angabe von `out` vor dem Bezeichner für das Ziel der Umwandlung.

- 4.5 Damit eine Spalte nicht sortiert werden kann, setzen Sie die Eigenschaft **SortMode** der Spalte auf **NotSortable**.

- 4.6 Das Aussehen der ungeraden Zeilen legen Sie über die Eigenschaft **AlternatingRowsDefaultCellStyle** fest.

Kapitel 5

- 5.1 Im ersten Schritt lassen Sie die Datenquellen anzeigen. Wählen Sie dann über das Kombinationsfeld für die Tabelle den Eintrag **Details** aus. Anschließend ziehen Sie die Tabelle in das Formular.

- 5.2 Die kleine Tonne bei der Eigenschaft **Text** symbolisiert, dass der Text des Eingabefelds aus einer Datenquelle stammt.

B. Glossar

ADO.NET	ADO.NET ist eine Schnittstelle für den Zugriff auf Daten in Datenbanken. Sie gehört zum <i>.NET Framework</i> .
Ausschussvariable	Eine Ausschussvariable kann als Platzhalter verwendet werden. Sie wird durch den Unterstrich <code>_</code> markiert.
Datenbank	Eine Datenbank ist eine strukturierte Sammlung von Daten in elektronisch verarbeitbarer Form.
Datenbank-Management-System	Ein Datenbank-Management-System übernimmt die Verwaltung von Datenbanken und stellt Anwendungsprogramme für die Verwaltung zur Verfügung.
Datenfeld	Ein Datenfeld speichert Teilinformationen eines Datensatzes. Jedes Datenfeld hat bestimmte Eigenschaften – zum Beispiel die Länge oder den Datentyp.
Datenprovider	Die Datenprovider gehören zu ADO.NET. Sie ermöglichen den Zugriff auf unterschiedliche Datenbank-Management-Systeme.
Datenquelle	Eine Datenquelle wird – etwas vereinfacht – von einem Datenprovider zur Verfügung gestellt und bietet die Daten aus Datenbanktabellen an. Die Datenquelle kann mit Steuerelementen in einem Formular verbunden werden.
Datensatz	Ein Datensatz speichert logisch zusammengehörige Informationen in einer Datenbank beziehungsweise einer Datenbanktabelle.
DBMS	Abkürzung für Datenbank-Management-System.
Primärschlüssel	Über den Primärschlüssel kann ein Datensatz in einer Datenbanktabelle eindeutig identifiziert werden.
Relation	Eine Relation ist im relationalen Datenmodell eine Tabelle.
Relationales Datenmodell	Beim relationalen Datenmodell werden Daten in Tabellen abgelegt, die miteinander verbunden werden. Die Spalten der Tabelle – im relationalen Datenmodell Attribute genannt – beschreiben dabei die Eigenschaften der Daten.
Server-Explorer	Der Server-Explorer ist ein Teil der Oberfläche von Visual Studio. Über den Server-Explorer können Sie zum Beispiel Verbindungen zu einer Datenbank herstellen.
SQL	SQL steht für <i>Structured Query Language</i> . SQL ist ein Sprachstandard für die Kommunikation zwischen Anwendungsprogrammen und relationalen Datenbank-Management-Systemen.

SQL Server	SQL Server ist ein Datenbank-Management-System, das zu Visual Studio gehört.
Structured Query Language	Siehe SQL.
Tupel	Ein Tupel ist im relationalen Datenmodell eine Zeile in einer Tabelle.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmiertechniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Das Klassendiagramm von Autovermietung 2030	3
Abb. 1.2	Die Datenbanktabellen für die Autovermietung	4
Abb. 1.3	Die Rohform des ersten Formulars	6
Abb. 1.4	Der Visual Studio Installer	6
Abb. 1.5	Der Workload Datenspeicherung und -verarbeitung (links in der Abbildung am Mauszeiger)	7
Abb. 2.1	Das eingeblendete Register Server-Explorer (links in der Abbildung)...	8
Abb. 2.2	Der Dialog Datenquelle auswählen	9
Abb. 2.3	Der Dialog Verbindung hinzufügen	9
Abb. 2.4	Die neue Datenbankdatei (links oben im Server-Explorer am Mauszeiger)	10
Abb. 2.5	Die Objekte einer Datenbank (links in der Abbildung)	11
Abb. 2.6	Das Anlegen einer neuen Tabelle	12
Abb. 2.7	Die Tabelle für die Kundendaten (links) und die Klasse Kunde (rechts)	13
Abb. 2.8	Die ersten beiden Spalten in der Datenbanktabelle	14
Abb. 2.9	Die aktualisierte Datenbank	16
Abb. 3.1	Der Zugriff aus einer Anwendung auf eine Datenbanktabelle	19
Abb. 3.2	Der Assistent zum Konfigurieren von Datenquellen	20
Abb. 3.3	Die Auswahl des Datenbankmodells	21
Abb. 3.4	Die Auswahl der Datenverbindung	21
Abb. 3.5	Die Rückfrage zum Kopieren	22
Abb. 3.6	Das Speichern der Verbindungszeichenfolge	23
Abb. 3.7	Die Auswahl der Datenbankobjekte	23
Abb. 3.8	Die ausgewählte Tabelle Kunde	24
Abb. 3.9	Die Anzeige der Datenquellen (links oben in der Abbildung)	25
Abb. 3.10	Die Steuerelemente für den Zugriff auf die Datenquelle	26
Abb. 3.11	Das Formular für die Listenansicht der Kunden	28
Abb. 3.12	Der erste gespeicherte Datensatz	28
Abb. 4.1	Der Dialog Spalten bearbeiten	33
Abb. 4.2	Der Fehlerhinweis bei einer leeren Zelle	36
Abb. 4.3	Der CellStyle-Generator	41
Abb. 4.4	Die Farbauswahl	42
Abb. 4.5	Unterschiedliche formatierte Zeilen und Spalten in der Tabelle	42
Abb. 5.1	Das Menü für die Auswahl der Darstellung (links oben in der Abbildung)	47

Abb. 5.2	Das Formular zur Einzelanzeige nach dem Einfügen der Datenquelle .	48
Abb. 5.3	Das Formular nach den Anpassungen	48
Abb. 5.4	Die Einzelansicht für die Kunden im praktischen Einsatz.....	49
Abb. 6.1	Die Tabelle für die Fahrzeugdaten (links) und die Klasse Fahrzeug (rechts).....	50
Abb. 6.2	Die Tabelle Fahrzeug	51
Abb. 6.3	Das Aktualisieren der Datenquelle	52
Abb. 6.4	Die Listenanzeige für die Fahrzeuge	53
Abb. 6.5	Die Einzelanzeige für die Fahrzeuge	53
Abb. 6.6	Die deaktivierten Funktionen bei den DataGridView-Aufgaben	54
Abb. 6.7	Eine erste einfache Liste der Fahrzeuge	54
Abb. 6.8	Der DataSet-Designer	55

E. Tabellenverzeichnis

Tab. 2.1	Die weiteren Spalten für die Datenbanktabelle	15
Tab. 4.1	Die maximalen Eingabelängen	33
Tab. 6.1	Die Spalten für die Tabelle der Fahrzeuge	51

F. Codeverzeichnis

Code 4.1	Das Überprüfen auf leere Einträge in einer Zelle	35
Code 4.2	Die Methode KundeDataGridView_CellEndEdit()	37
Code 4.3	Die erweiterte Methode KundeDataGridView_CellValidating()	39
Code 4.4	Das Anzeigen einer Fehlermeldung im Ereignis DataError für das Steuerelement kundeDataGridView	39
Code 4.5	Die try-catch-Konstruktion für die Methode KundeBindingNavigatorSaveItem_Click() (die neuen Anweisungen sind fett markiert)	40
Code 4.6	Die Methoden KundeDataGridView_CellMouseEnter() und KundeDataGridView_CellMouseLeave()	43

G. Sachwortverzeichnis

A

ADO.NET	19
Ausschussvariable	39

C

CellStyle-Generator	41
---------------------------	----

D

Datenbank

Erstellen der	8
Dateneingabe	32
Datenprovider	19
Datenquelle	19
anlegen	20
auswählen	9
Datentyp	17
Datenzugriffsschnittstelle	19

E

Eingabe

Afbangen von leeren	34
Eingabelänge	
Festlegen der maximalen	32
Einzelanzeige	
Formular zur	47

S

Server-Explorer	8
SQL Server	8
Steuerelement	
DataGridView	24

T

Tabelle

anlegen	10
---------------	----

H. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSHP21D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Schicken Sie bitte das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, packen Sie bitte die Projekte vor dem E-Mail-Versand mit einem geeigneten Programm – zum Beispiel mit WinZip oder direkt über Windows. Beschreiben Sie bitte bei der Einsendeaufgabe für dieses Studienheft zusätzlich, welche grundsätzlichen Schritte für die Erweiterungen erforderlich sind – also zum Beispiel, welche Steuerelemente Sie einfügen und wie Sie die Eigenschaften dieser Elemente setzen.

In dem Programm aus diesem Studienheft werden in den Formularen für die Einzelanzeige der Kunden und Fahrzeuge bisher keinerlei Prüfungen für die Daten durchgeführt. Ergänzen Sie diese Prüfungen bitte. Achten Sie dabei vor allem auf folgende Punkte:

1. Sorgen Sie dafür, dass in die Felder nicht mehr Zeichen eingegeben werden können, als in der entsprechenden Spalte in der Datenbanktabelle abgelegt werden können.
2. Stellen Sie sicher, dass alle Felder, für die Daten erforderlich sind, auch tatsächlich gefüllt werden.
3. Überprüfen Sie bei der Eingabe eines Kunden für die Postleitzahl, ob genau fünf Ziffern eingegeben wurden.
4. Stellen Sie sicher, dass mögliche Fehler beim Sichern der Daten über das Symbol **Daten speichern** abgefangen werden.
5. Sorgen Sie dafür, dass das erste Feld in dem Formular – die Kunden- beziehungsweise Fahrzeugnummer – nicht mehr eingabebereit ist.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Fortgeschrittene Techniken
bei der Datenbankprogrammierung
Wartungs- und Konfigurationsmanagement

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1219N01

CSHP22D

Objektorientierte Software-Entwicklung mit C#

**Beispielprojekt: Fortgeschrittene Techniken
bei der Datenbankprogrammierung**

Wartungs- und Konfigurationsmanagement

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Beispielprojekt: Fortgeschrittene Techniken bei der Datenbankprogrammierung Wartungs- und Konfigurationsmanagement

Inhaltsverzeichnis

Einleitung	1
1 Die Tabellen für die Verträge und die Reservierungen	3
1.1 Ein wenig Theorie	3
1.2 Das Anlegen der Tabellen	6
1.3 Das Erstellen der Beziehungen	10
Zusammenfassung	17
2 Die Verträge	20
2.1 Das Formular für die Verträge	20
2.2 Die Detailanzeige für Kunden und Fahrzeuge	29
2.3 Die Vermietung	37
2.4 Die Rückgabe	47
3 Die Reservierungen	50
Zusammenfassung	53
4 Die Datenintegrität	55
Zusammenfassung	63
5 Wartung	64
5.1 Warum muss Software gewartet werden?	64
5.2 Wartungsformen	66
5.3 Probleme bei der Wartung	67
Zusammenfassung	69
6 Konfigurationsmanagement	70
6.1 Aufgaben des Konfigurationsmanagements	70
6.2 Die Versionsverwaltung	71
6.3 Geplante Wartung und Schnellschüsse	73
6.4 Werkzeuge für das Konfigurationsmanagement	74
Zusammenfassung	75
Schlussbetrachtung	76

Anhang

A.	Lösungen zu den Aufgaben zur Selbstüberprüfung	77
B.	Glossar	79
C.	Literaturverzeichnis	83
D.	Abbildungsverzeichnis	84
E.	Tabellenverzeichnis	86
F.	Codeverzeichnis	87
G.	Medienverzeichnis	88
H.	Sachwortverzeichnis	89
I.	Einsendeaufgabe	91

Einleitung

In diesem Studienheft werden wir die Datenbankanwendung weiter ausbauen. Wir erstellen unter anderem zwei weitere Tabellen für Verträge und Reservierungen und verknüpfen die Informationen in den Tabellen miteinander. Außerdem erstellen wir die Funktionen zum Vermieten, zum Zurückgeben und zum Reservieren von Fahrzeugen. Abschließend beschäftigen wir uns auch noch mit der Wartung von Software.

Im Einzelnen lernen Sie in diesem Studienheft,

- wie Daten im relationalen Datenbankmodell auf unterschiedliche Tabellen verteilt werden,
- wie Daten in verschiedenen Tabellen miteinander verknüpft werden können,
- welche unterschiedlichen Beziehungen es zwischen Datenbanktabellen gibt,
- worauf Sie bei Beziehungen zwischen Datenbanktabellen besonders achten müssen,
- was sich hinter der referenziellen Integrität verbirgt,
- wie Sie die Tabellen für die Verträge und die Reservierungen mit Visual Studio anlegen,
- wie Sie die Beziehungen zwischen den Tabellen mit den Datenbankwerkzeugen von Visual Studio erstellen,
- wie Sie die Regeln für die Beziehungen festlegen,
- wie Sie Formulare für verbundene Datenbanktabellen erstellen,
- wie Sie im Formular für eine Datenbanktabelle über Kombinationsfelder direkt auf Werte aus einer anderen Datenbanktabelle zugreifen,
- wie Sie über Abfragen gezielt Daten in einer Datenbanktabelle selektieren,
- wie Sie das Steuerelement **DateTimePicker** für die Auswahl von Datums- und Zeitwerten einsetzen,
- wie Sie direkt im Quelltext neue Datensätze anlegen,
- wie Sie die Daten für ein Formular über eine Methode und eine Abfrage beschaffen,
- wie Sie aus einem Formular eine Detailanzeige für einen Datensatz in einer verbundenen Datenbanktabelle durchführen,
- wie Sie sicherstellen, dass beim Löschen von Kunden und Fahrzeugen auch die dazugehörigen Verträge und Reservierungen gelöscht werden,
- wie Sie Einschränkungen für verbundene Tabellen gezielt ausschalten,
- warum Software gewartet werden muss,
- welche verschiedenen Formen der Wartung unterschieden werden,
- welche typischen Probleme bei der Wartung auftreten,
- warum ein Konfigurationsmanagement erforderlich ist,
- welche Aufgaben das Konfigurationsmanagement hat,
- wie die Versionsverwaltung arbeitet,
- was die geplante Wartung und einen Schnellschuss unterscheidet und
- mit welchen Programmen das Konfigurationsmanagement unterstützt wird.

Christoph Siebeck

1 Die Tabellen für die Verträge und die Reservierungen

In diesem Kapitel werden wir die beiden neuen Tabellen für die Verträge und die Reservierungen anlegen. Außerdem werden wir die Beziehungen zwischen den Tabellen erstellen.

1.1 Ein wenig Theorie

Bevor wir uns an die Erweiterung unserer Datenbankanwendung machen, wollen wir uns zunächst einmal ansehen, wofür Beziehungen zwischen Tabellen überhaupt eingesetzt werden und welche unterschiedlichen Formen der Beziehungen es gibt.

Im Entwurf haben wir die Datenhaltung und die einzelnen Tabellen für unsere Anwendung ja bereits festgelegt. Das Datenbankmodelldiagramm sah so aus:

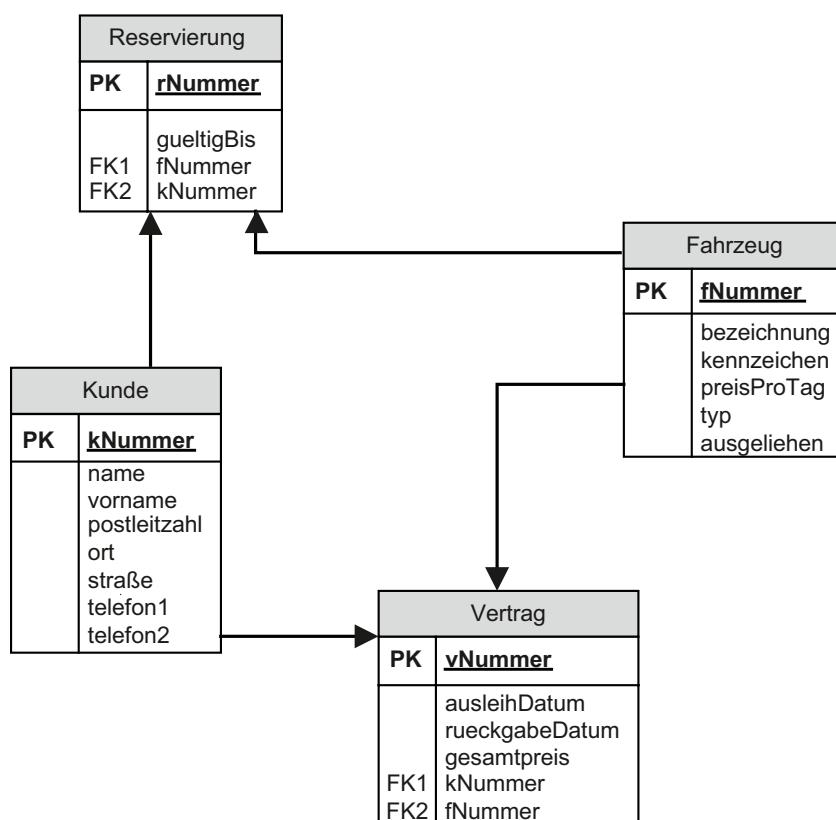


Abb. 1.1: Die Tabellen für die Autovermietung

Alle diese Daten könnten Sie grundsätzlich auch zusammen in einer einzigen Tabelle ablegen. Wenn Sie nur Daten zu einem Kunden eingeben möchten, lassen Sie die Felder für das Fahrzeug, den Vertrag und die Reservierung eben leer. Genauso könnten Sie vorgehen, wenn Sie Daten zu einem Fahrzeug erfassen möchten. In diesem Fall lassen Sie einfach die Felder für den Kunden, den Vertrag und die Reservierung leer.

Diese Form der Speicherung ist zwar nicht besonders effektiv, funktioniert aber zumindest bei den Kunden- und den Fahrzeugdaten noch einigermaßen. Problematisch wird es allerdings dann, wenn Sie Daten zu einem Vertrag oder einer Reservierung erfassen möchten. Sie müssten dann für jeden Vertrag oder jede Reservierung neben den eigent-

lichen Daten immer wieder die vollständigen Kundendaten und auch die vollständigen Fahrzeugdaten erfassen. Wenn Sie nur die eigentlichen Daten eingeben und die Kunden- und Fahrzeugdaten leer lassen, haben Sie keine Möglichkeit, den Vertrag oder die Reservierung einem Kunden oder einem Fahrzeug zuzuordnen.

Bei diesem Verfahren machen Sie sich aber durch das wiederholte Erfassen identischer Daten sehr viel Arbeit, die ja eigentlich völlig überflüssig ist. Wenn Sie einmal die Daten eines Kunden eingegeben haben, müssen Sie sie ja nicht immer wieder neu erfassen.

Auch das Ändern von Informationen ist bei der gemeinsamen Speicherung der Daten in einer einzigen Tabelle ausgesprochen schwierig. Stellen Sie sich nur einmal vor, Sie haben für einen Kunden 25 Verträge erfasst und der Kunde zieht um. Sie müssten dann in 25 Datensätzen die Anschrift des Kunden einzeln ändern.

Einen Ausweg aus diesem Dilemma bietet die **Normalisierung** von Tabellen.



Bei der Normalisierung werden die Daten auf mehrere einzelne Tabellen verteilt. Diese einzelnen Tabellen werden dann miteinander verbunden.

Die Normalisierung von Tabellen ist eine anspruchsvolle Aufgabe, die vor allem bei komplexen Datenbanken von Spezialisten ausgeführt wird. Wir beschränken uns nur auf die wichtigsten Zwischenschritte.

In unserem Beispiel haben wir große Teile der Normalisierung bereits beim Entwurf anhand des Klassendiagramms durchgeführt. Die Daten werden dabei auf vier Tabellen verteilt. Die erste Tabelle speichert nur die Daten zu den Kunden, die zweite Tabelle speichert die Daten zu den Fahrzeugen, die dritte Tabelle speichert die Daten eines Vertrags und die vierte Tabelle schließlich die Daten einer Reservierung. Die Beziehungen zwischen den Tabellen haben wir dabei über Primär- und Fremdschlüssel hergestellt.



Zur Auffrischung:

Ein **Primärschlüssel** identifiziert einen Datensatz eindeutig.

Primärschlüssel einer Tabelle, die in einer anderen Tabelle verwendet werden, um eine Beziehung herzustellen, werden **Fremdschlüssel** genannt.

In der Tabelle mit den Verträgen haben wir zum Beispiel zwei zusätzliche Felder mit den Namen **kNummer** und **fNummer** angelegt. In diese beiden Felder können die Werte der entsprechenden Primärschlüssel aus der Tabelle mit den Kunden und der Tabelle mit den Fahrzeugen eingegeben werden. Damit können wir problemlos den Kunden und auch das Fahrzeug identifizieren, ohne die Daten noch einmal in der Tabelle mit den Verträgen erfassen zu müssen.

Die Kundennummer kann in der Tabelle mit den Verträgen auch mehrfach vorkommen. Das ist zum Beispiel dann der Fall, wenn ein Kunde mehrere Fahrzeuge gemietet hat. In der Tabelle mit den Kundendaten kann die Kundennummer dagegen nur einmal vorkommen, da sie ja dort als Primärschlüssel definiert ist.

Diese Art der Beziehung wird **1:n-Beziehung** genannt. Für genau einen Datensatz in einer Tabelle können mehrere Datensätze in einer anderen Tabelle existieren.



Weitere mögliche Beziehungen sind die **1:1-Beziehung** und die **n:m-Beziehung**. Bei der 1:1-Beziehung gibt es für genau einen Datensatz in einer Tabelle auch genau einen Datensatz in einer anderen Tabelle. Bei der n:m-Beziehung dagegen kann es für beliebig viele Datensätze in einer Tabelle auch beliebig viele Datensätze in einer anderen Tabelle geben.

Eine 1:1-Beziehung liegt zum Beispiel vor, wenn Sie die Daten von Mitarbeitern in einer Tabelle speichern und in einer weiteren Tabelle eine Art Personalakte für jeden Mitarbeiter pflegen. In der Tabelle mit den Personalakten gibt es dann genau einen Datensatz für jeden Datensatz in der Tabelle mit den Mitarbeitern.

Eine n:m-Beziehung besteht zum Beispiel zwischen Sportlern und Sportarten. Ein Sportler kann beliebig viele Sportarten betreiben, gleichzeitig kann eine Sportart auch von mehreren Sportlern betrieben werden.

Damit Beziehungen zwischen mehreren Tabellen nicht zu Problemen führen, müssen Sie einige Regeln beachten:

- Sie dürfen in dem Feld, über das die Beziehung hergestellt wird, nur gültige Werte aus der anderen Tabelle eingeben.

Wenn Sie zum Beispiel bei den Verträgen eine Kundennummer erfassen, die nicht vorhanden ist, kann natürlich auch kein Datensatz in der Tabelle mit den Kunden gefunden werden.

- Sie dürfen Datensätze, zu denen eine Beziehung besteht, nicht löschen.

Wenn zum Beispiel in der Tabelle mit den Verträgen die Fahrzeugnummer 1001 erscheint, darf der Datensatz mit der Fahrzeugnummer 1001 in der Tabelle mit den Fahrzeugen nicht gelöscht werden. Andernfalls geht die Beziehung verloren und Sie haben eine Art „Karteileiche“ in der Tabelle mit den Verträgen.

In der Tabelle mit den Verträgen dagegen können Sie ohne Weiteres Datensätze löschen, da hier in den anderen Tabellen keine Daten verloren gehen.

- Sie dürfen im Entwurf einer Tabelle keine Felder löschen, über die eine Beziehung zu einer anderen Tabelle hergestellt wird.

Wenn Sie zum Beispiel in der Tabelle mit den Verträgen das Feld für die Kundennummer löschen, geht der Bezug zu den Kunden verloren. Ähnliches würde passieren, wenn Sie das Feld für die Kundennummer in der Tabelle **Kunde** selbst löschen. Dann könnte ebenfalls kein Bezug mehr aus der Tabelle mit den Verträgen hergestellt werden.

Für die Einhaltung dieser Regeln verwendet der SQL Server die **referenzielle Integrität**.



Die referenzielle Integrität verhindert zum Beispiel, dass Sie ungültige Werte eingeben oder dass Sie Datensätze löschen, die für einen Bezug zwingend erforderlich sind.

Nach so viel Theorie wollen wir uns nun die Beziehungen zwischen Tabellen in der Praxis ansehen. Beginnen wir jetzt mit dem Anlegen der beiden neuen Tabellen.

1.2 Das Anlegen der Tabellen

Hinweis:

Da sich das Anlegen der neuen Tabellen kaum vom Anlegen der Tabelle für die Kunden unterscheidet, stellen wir Ihnen hier nur die wichtigsten Schritte in kompakter Form vor. Wenn Sie nicht mehr genau wissen, was sich hinter den einzelnen Schritten verbirgt, lesen Sie bitte noch einmal im letzten Studienheft nach.

Starten Sie Visual Studio und laden Sie das Projekt für unsere Datenbankanwendung. Lassen Sie dann den Server-Explorer anzeigen und klicken Sie im Zweig **Datenverbindungen** auf das Symbol vor dem Eintrag **auto2030.mdf**. Öffnen Sie anschließend in der Baumstruktur das Kontextmenü für den Eintrag **Tabellen** und wählen Sie die Funktion **Neue Tabelle hinzufügen**.

Hinweis:

Falls der Eintrag für die Datenbank im Server-Explorer nicht angezeigt wird, klicken Sie einmal im Projektmappen-Explorer auf den Eintrag. Danach erscheint er auch im Server-Explorer im Zweig **Datenverbindungen**.

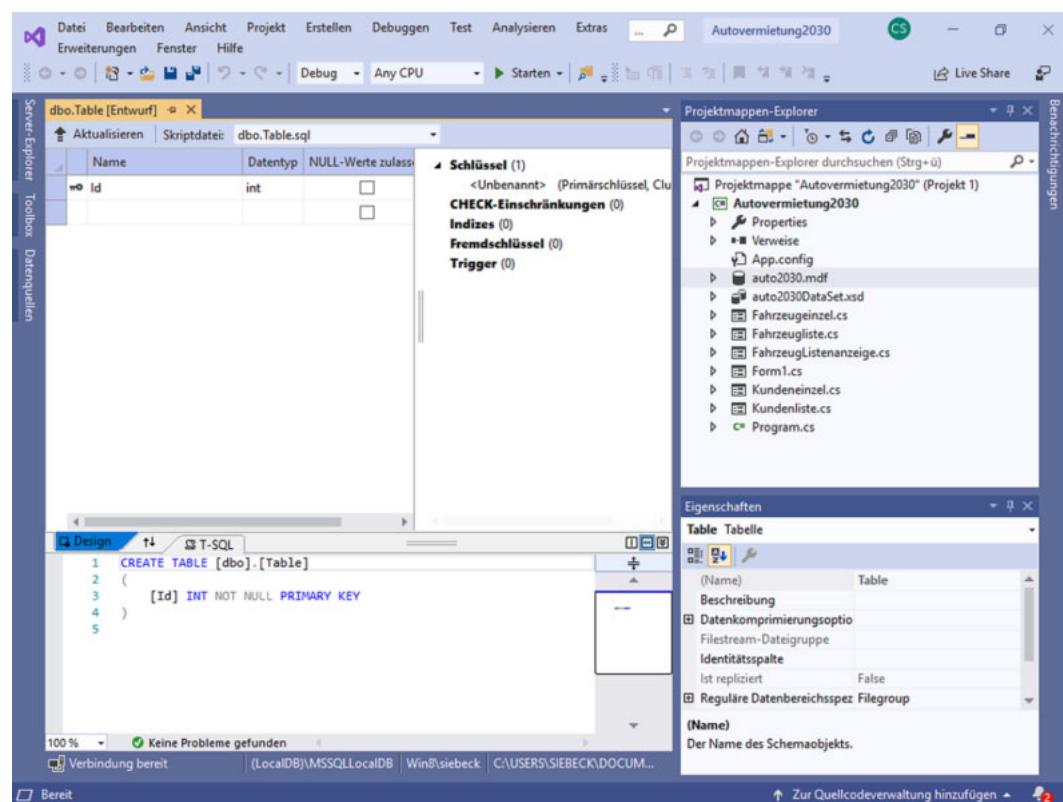


Abb. 1.2: Die neue Tabelle im Tabellen-Designer

Legen Sie dann die Spalten für die Verwaltung der Vertragsdaten an.

Bevor Sie weiterlesen ...

Versuchen Sie erst einmal, selbst die Spalten aus dem Datenbankmodelldiagramm in der Abb. 1.1 und dem Klassendiagramm abzuleiten.

Die erforderlichen Daten finden Sie in der folgenden Tab. 1.1.

Tab. 1.1: Die Spalten für die Tabelle mit den Vertragsdaten

Spaltenname	Datentyp	NULL-Werte zulassen
vNummer	int	nein
ausleihDatum	datetime	nein
rueckgabeDatum	datetime	nein
gesamtpreis	float	nein
kNummer	int	nein
fNummer	int	nein

Für die Spalten **ausleihDatum** und **rueckgabeDatum** verwenden wir den Datentyp `datetime`¹. Dieser Typ kann sowohl ein Datum als auch eine Uhrzeit speichern.

Die Vertragsnummer soll genau wie die Kundennummer automatisch vom System vergeben werden. Dazu benutzen wir wieder die Eigenschaft **Ist Identity**.

Klicken Sie oben im Tabellen-Designer in eine beliebige Spalte der Zeile für die Vertragsnummer. Ändern Sie dann unten im Fenster den Wert der Eigenschaft (**Ist Identity**) in der Gruppe **Identitätsspezifikation** von `False` in `True` und setzen Sie die Eigenschaft **Identitätsseed** auf den Wert `1000`. Die Schrittweite bei der Eigenschaft **ID-Inkrement** können Sie unverändert auf `1` stehen lassen. Damit werden unsere Vertragsnummern automatisch ab der Nummer `1000` vergeben.

Hinweis:

Sie können die Vertragsnummer auch bei `1` beginnen lassen. Eine eindeutige Unterscheidung zur Kundennummer ist nicht zwingend erforderlich, da die beiden Spalten ja in verschiedenen Tabellen liegen.

Prüfen Sie dann noch, ob die Spalte **vNummer** als Primärschlüssel festgelegt ist.

Ändern Sie abschließend den Namen der Tabelle in **Vertrag**. Die neue Tabelle sollte nun ungefähr so aussehen:

1. Übersetzt bedeutet `datetime` so viel wie „Datum/Zeit“.

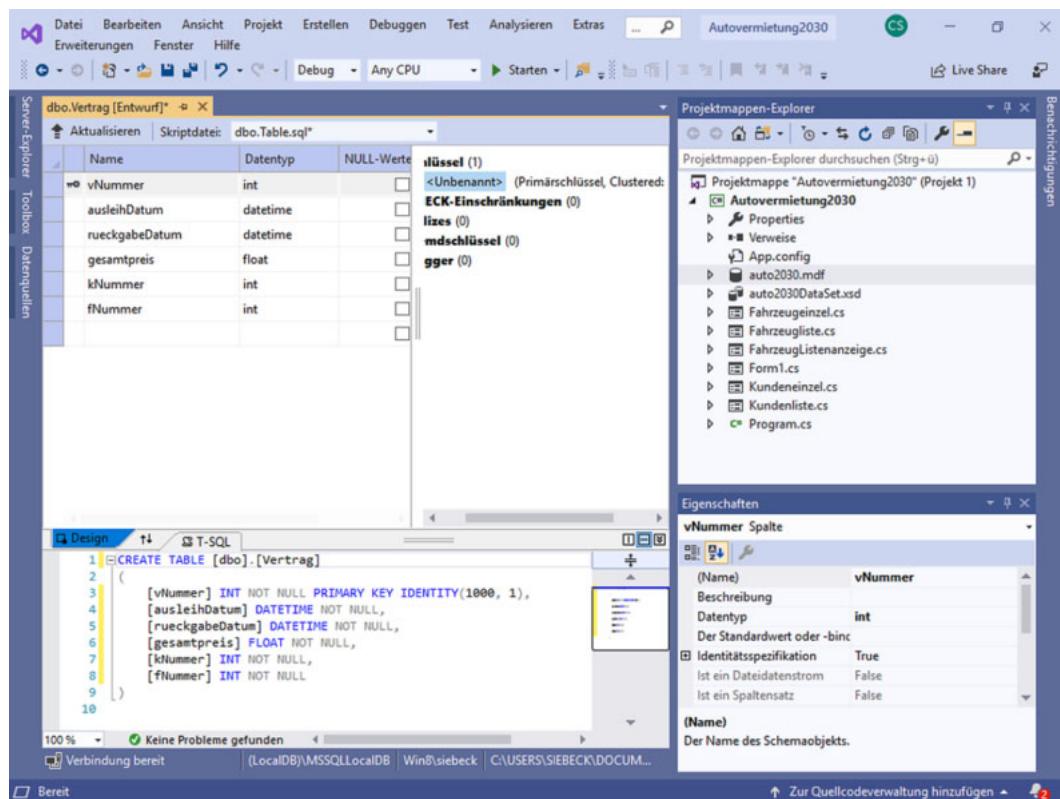


Abb. 1.3: Die Tabelle **Vertrag**

Lassen Sie jetzt noch die Datenbank aktualisieren.



Bitte beachten Sie:

Über die beiden Spalten **kNummer** und **fNummer** stellen wir gleich die Beziehung zu den beiden Tabellen **Kunde** und **Fahrzeug** her. Das ist aber nur dann möglich, wenn die Spalten denselben Datentyp verwenden wie die entsprechenden Spalten in den Tabellen, zu denen die Beziehung hergestellt werden soll. Daher müssen in unserem Beispiel sowohl die Spalte **kNummer** als auch die Spalte **fNummer** zwingend den Datentyp **int** haben.

Die Bezeichner der Spalten, über die eine Beziehung hergestellt wird, können dagegen auch unterschiedlich sein. Sie könnten also für die Spalte **kNummer** in der Tabelle **Vertrag** auch den Bezeichner **vkNummer** benutzen und trotzdem eine Beziehung zur Spalte **kNummer** in der Tabelle **Kunde** herstellen. Wichtig ist vor allem, dass die Datentypen zueinander passen. Der Bezeichner spielt keine Rolle.

Im nächsten Schritt können wir jetzt die Tabelle für die Reservierungen anlegen.

Lassen Sie gegebenenfalls noch einmal den Server-Explorer anzeigen und markieren Sie in der Baumstruktur den Eintrag für die Datenbank. Erstellen Sie anschließend in der Baumstruktur über das Kontextmenü für den Eintrag **Tabellen** eine neue Tabelle.

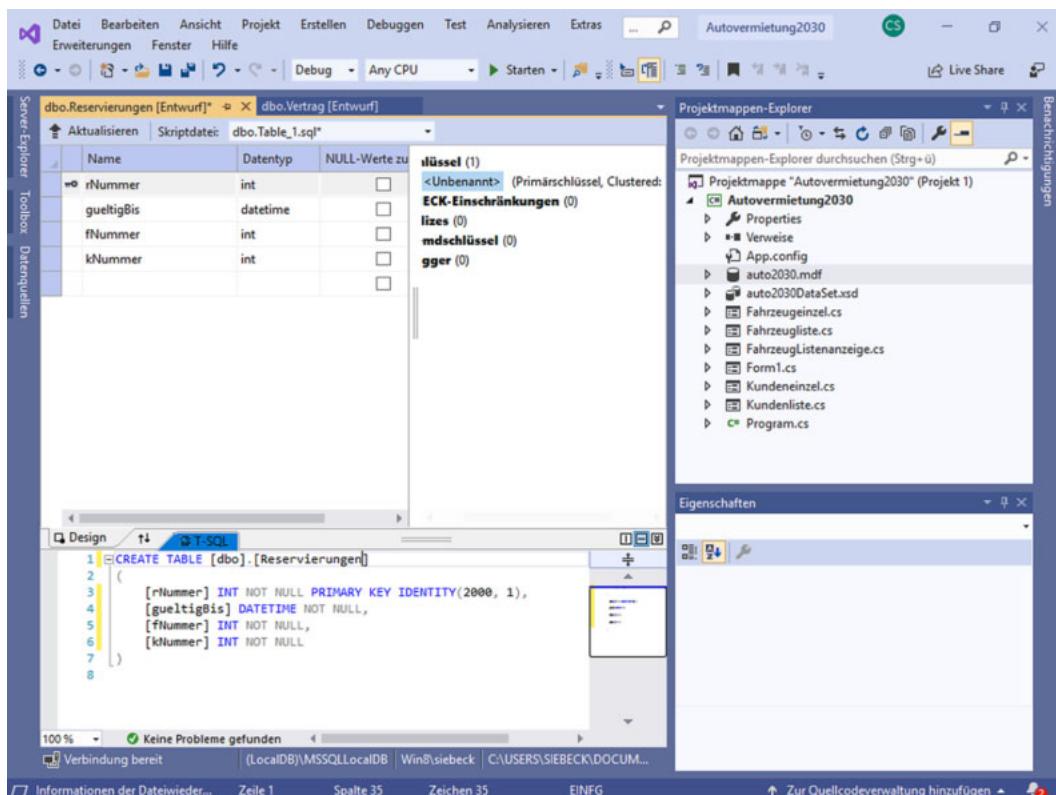
Legen Sie danach vier Spalten für die neue Tabelle an. Überlegen Sie auch hier zunächst einmal selbst anhand des Datenbankmodelldiagramms und des Klassendiagramms, welche Daten Sie benötigen. Vergleichen Sie dann Ihre Überlegungen mit der Tab. 1.2.

Tab. 1.2: Die Spalten für die Tabelle mit den Reservierungen

Spaltenname	Datentyp	NULL erlaubt
rNummer	int	nein
gueltigBis	datetime	nein
fNummer	int	nein
kNummer	int	nein

Sorgen Sie über die Eigenschaft (**Ist Identity**) dafür, dass die Werte für die Spalte **rNummer** automatisch vom System vergeben werden. Sie können hier zum Beispiel mit der Nummer 2000 beginnen.

Prüfen Sie abschließend, ob die Spalte **rNummer** als Primärschlüssel festgelegt ist, und ändern Sie den Namen der Tabelle in **Reservierungen**. Die neue Tabelle sollte nun so aussehen:

**Abb. 1.4:** Die Tabelle **Reservierungen**

Lassen Sie dann noch einmal die Datenbank aktualisieren.

Damit sind die beiden Tabellen vollständig angelegt und wir können jetzt die Beziehungen herstellen.

1.3 Das Erstellen der Beziehungen

Dazu müssen Sie im ersten Schritt zunächst einmal die neu erstellten Tabellen zu den Datenquellen für das Projekt hinzufügen.



Bitte denken Sie daran:

Änderungen an der Datenbank führen nicht automatisch auch zu Änderungen an den Datenquellen. Die Datenquellen beziehen sich in der Regel auf einzelne Tabellen einer Datenbank und nicht auf eine vollständige Datenbank.

Wechseln Sie in ein beliebiges Formular der Anwendung und klicken Sie im Register **Datenquellen** auf das Symbol **Datenquelle mit Assistenten konfigurieren** . Markieren Sie im Fenster **Assistent zum Konfigurieren von Datenquellen** im Zweig **Tabellen** zusätzlich die beiden gerade angelegten Tabellen.

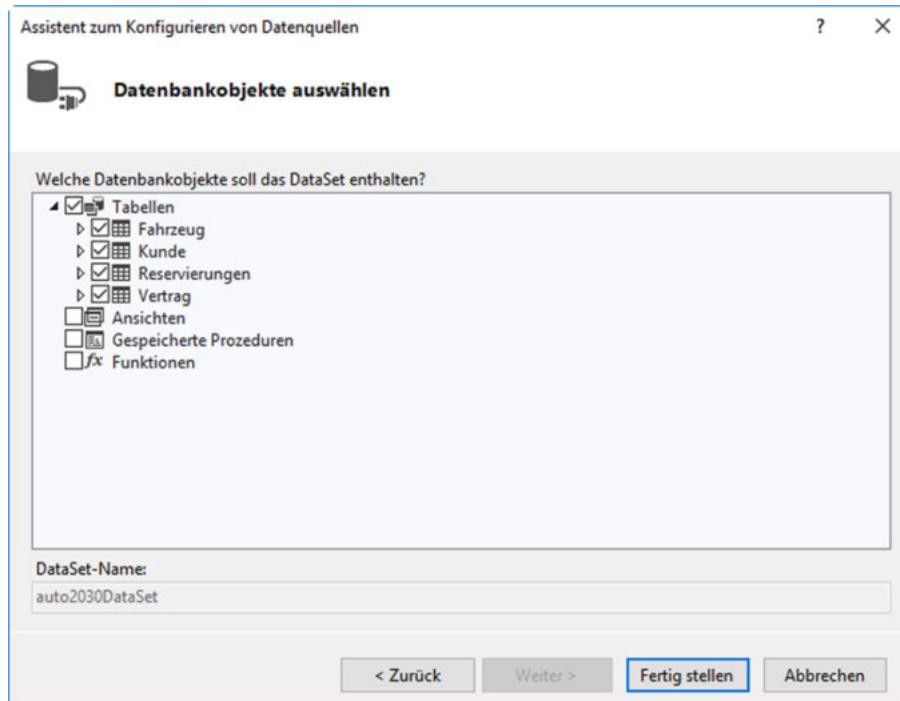


Abb. 1.5: Die Auswahl der neu erstellten Tabellen im Assistenten

Übernehmen Sie anschließend die Änderungen mit einem Klick auf die Schaltfläche **Fertig stellen**. Visual Studio aktualisiert dann die Datenquellen. Das kann einen kurzen Moment dauern.

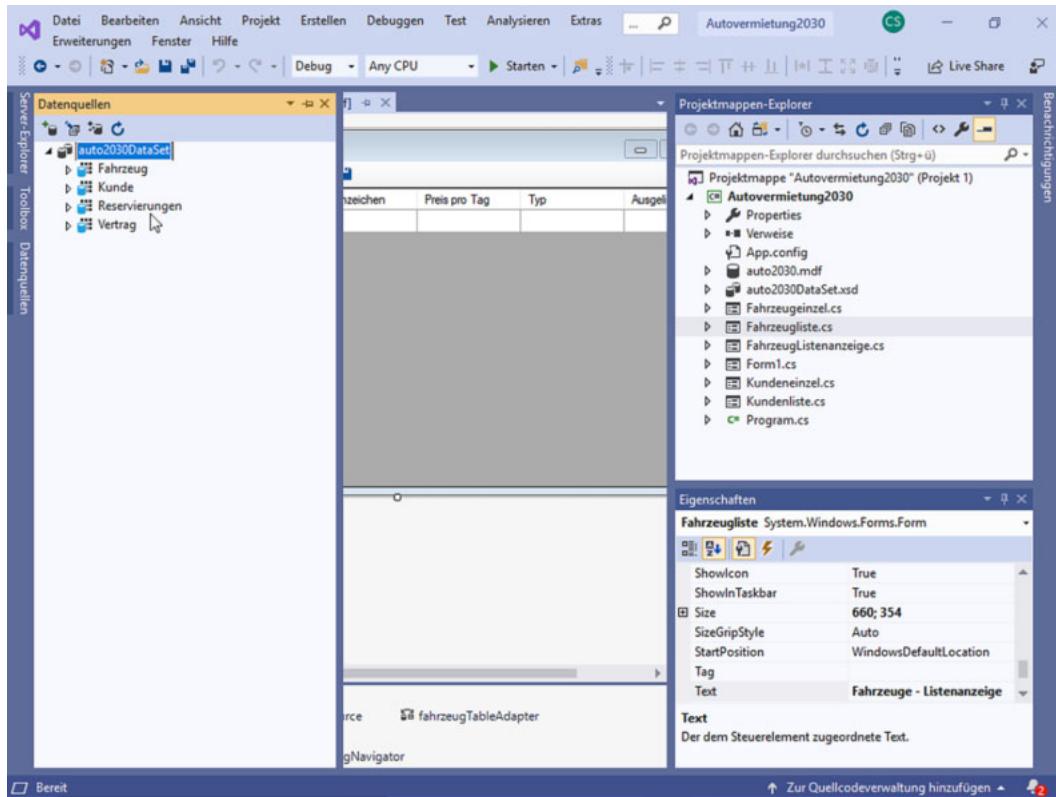


Abb. 1.6: Die aktualisierten Datenquellen (oben links in der Abbildung)

Im nächsten Schritt können Sie jetzt die Beziehungen zwischen den Tabellen festlegen. Das erfolgt am einfachsten auf grafischem Wege über den DataSet-Designer. Starten Sie bitte den Designer über das Symbol **DataSet mit Designer bearbeiten** in der Symbolleiste der Datenquellen.

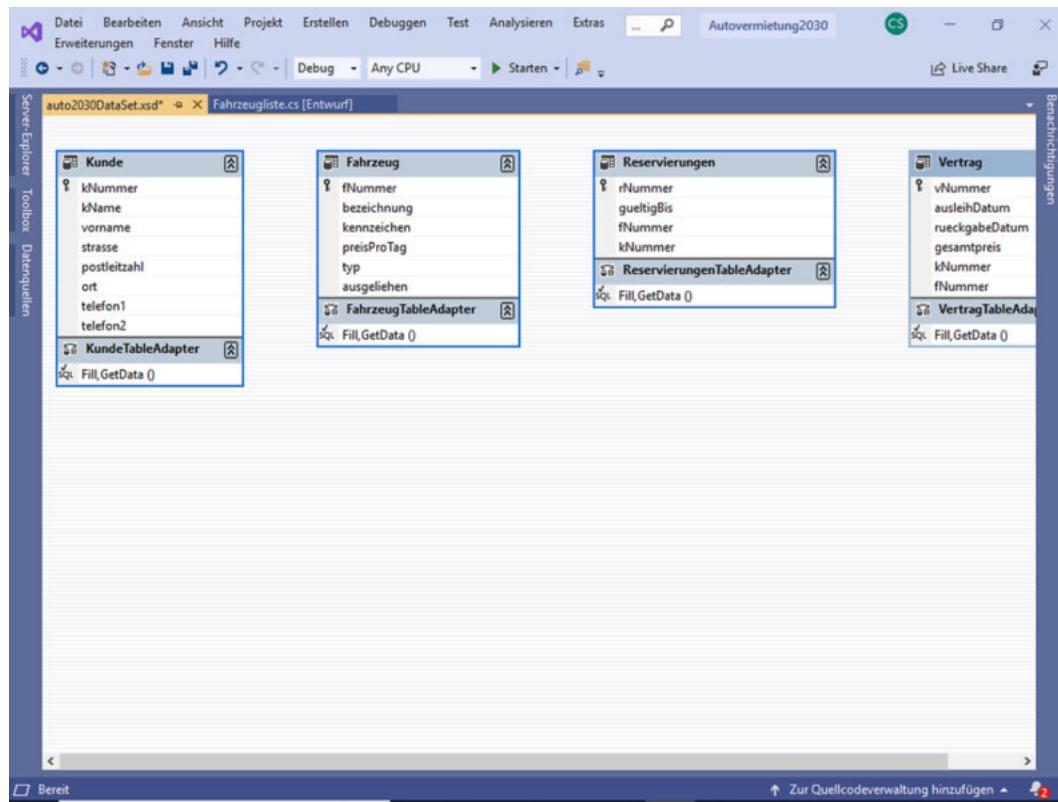


Abb. 1.7: Der DataSet-Designer (die weiteren Bereiche sind zur besseren Übersicht ausgeblendet)

Jetzt werden die vier Tabellen unserer Datenbank als einzelne Kästen angezeigt. Oben in jedem der Kästen finden Sie jeweils den Namen der Tabelle und darunter die verschiedenen Spalten sowie einige Funktionen.

Tipp:

Bei niedrigen Auflösungen sind unter Umständen nur Teile der Tabellen zu sehen. Sie können die Tabellen aber beliebig auf dem Bildschirm positionieren. Ziehen Sie dazu den Kasten mit gedrückter linker Maustaste an die neue Position und lassen Sie die Maustaste wieder los.

Um eine Beziehung zwischen zwei Tabellen herzustellen, ziehen Sie mit der Maus die Spalte aus der Tabelle, die die vollständigen Informationen enthält, auf die Spalte der Tabelle, in der der Schlüssel aus der anderen Tabelle gespeichert werden soll.



Die Tabelle mit den vollständigen Informationen wird auch **übergeordnete Tabelle** genannt. Die Tabelle, aus der die Daten über den Schlüssel beschafft werden, heißt auch **untergeordnete Tabelle**.

Probieren Sie das jetzt einmal aus. Klicken Sie mit der Maus in den Zeilenkopf der Spalte **kNummer** in der Tabelle **Kunde**. Ziehen Sie dann die Spalte über den Zeilenkopf zunächst mit gedrückter linker Maustaste nach links oder rechts aus der Tabelle heraus und legen Sie sie auf der Spalte **kNummer** in der Tabelle **Vertrag** ab.

Bitte beachten Sie:

Sie müssen die Zeile durch einen Mausklick in den Zeilenkopf markieren und auch den Zeilenkopf zum Ziehen benutzen. Wenn Sie mit der Maus in die Zeile klicken, wird lediglich der Name der Spalte markiert und die Zeile kann nicht verschoben werden. Auch das Ziehen der komplett markierten Zeile über den Namen der Zeile funktioniert nicht.



Achten Sie außerdem bitte darauf, dass Sie die Zeile zunächst nach links oder rechts aus der Tabelle herausziehen müssen. Wenn Sie die Zeile nach unten ziehen, werden lediglich weitere Zeilen in der Tabelle markiert.

Tipp:

Dass Sie an der richtigen Stelle in die richtige Richtung ziehen, erkennen Sie an einer gestrichelten Linie, die erscheint, sobald Sie die Maus aus der Tabelle herausbewegen. Wenn diese Linie nicht angezeigt wird, müssen Sie gegebenenfalls die Markierung ändern, am Zeilenkopf der Zeile ziehen oder aber die Maus nach rechts oder links aus der Tabelle herausbewegen.

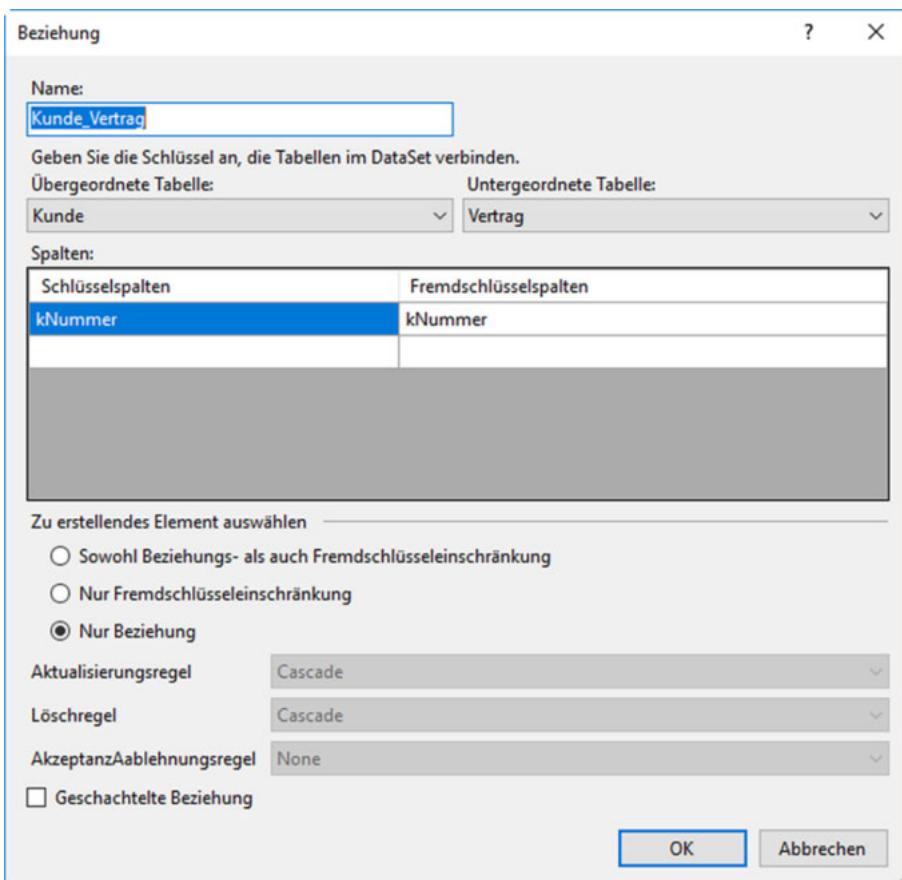


Abb. 1.8: Das Fenster **Beziehung**

Im Fenster **Beziehung** schlägt Ihnen der DataSet-Designer anschließend oben einen Namen für die Beziehung vor. Darunter finden Sie links die übergeordnete Tabelle und rechts die untergeordnete Tabelle. Im Bereich **Spalten** werden links die Spalte mit dem Primärschlüssel der übergeordneten Tabelle angezeigt und rechts die Spalte mit dem Fremdschlüssel in der untergeordneten Tabelle.

Tipp:

Sowohl die Tabellen als auch die Spalten können Sie nachträglich ändern – zum Beispiel, wenn Sie die Maus versehentlich auf einer falschen Spalte losgelassen haben. Klicken Sie dazu einfach mit der Maus in das entsprechende Feld und wählen Sie dann den korrekten Eintrag über die Liste des Kombinationsfelds aus.

*Bitte beachten Sie dabei aber, dass Visual Studio nicht direkt überprüft, ob die Beziehung überhaupt möglich ist. Sie können hier also auch zwei Spalten benutzen, die zum Beispiel einen unterschiedlichen Datentyp haben. Die eigentliche Prüfung erfolgt erst dann, wenn Sie auf die Schaltfläche **OK** klicken. Und auch erst dann erscheinen Fehlermeldungen.*

Im Bereich **Zu erstellendes Element auswählen** schließlich legen Sie fest, wie die beiden Tabellen miteinander verbunden werden sollen. In unserem Fall sollen sowohl eine Beziehung erstellt werden als auch Fremdschlüsseleinschränkungen gesetzt werden. Durch diese Fremdschlüsseleinschränkungen wird später sichergestellt, dass zum Beispiel nicht ohne Weiteres Daten in der übergeordneten Tabelle gelöscht werden können, die in der untergeordneten Tabelle verwendet werden.

Markieren Sie bitte die Option **Sowohl Beziehungs- als auch Fremdschlüsseleinschränkung** oben im Bereich **Zu erstellendes Element auswählen**.

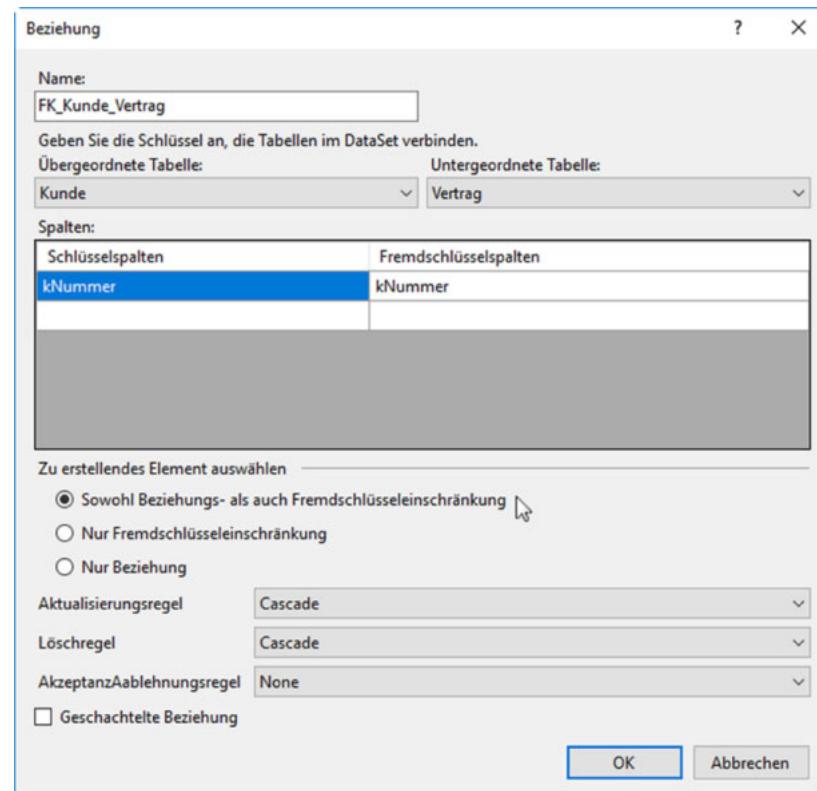


Abb. 1.9: Die geänderte Option für die Beziehung

Jetzt könnten Sie über die drei Kombinationsfelder unten im Fenster auch noch im Detail festlegen, was geschehen soll, wenn Daten in der übergeordneten Tabelle aktualisiert beziehungsweise gelöscht werden. Die Standardeinstellung **Cascade**² für die beiden oberen Felder sorgt zum Beispiel dafür, dass beim Aktualisieren beziehungsweise Löschen von Daten in der übergeordneten Tabelle auch die Datensätze in der untergeordneten Tabelle geändert beziehungsweise gelöscht werden. Da in unserem Fall die Kundennummer automatisch vom System vergeben wird und gar nicht vom Anwender geändert werden kann, können wir beim Aktualisieren auf die automatische Anpassung in der untergeordneten Tabelle verzichten. Ändern Sie daher bitte den Eintrag für das Feld **Aktualisierungsregel** auf **None**³.

Klicken Sie anschließend auf die Schaltfläche **OK**, um die Einstellungen für die Beziehung zwischen den Tabellen **Kunde** und **Vertrag** zu übernehmen.

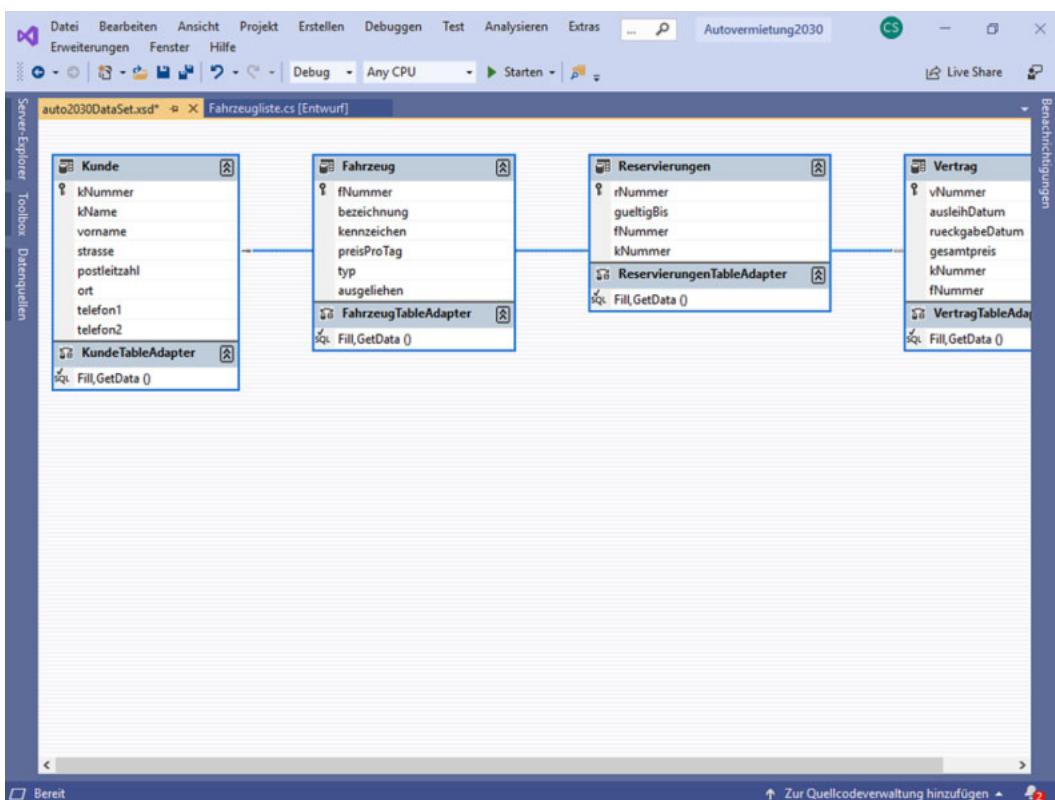
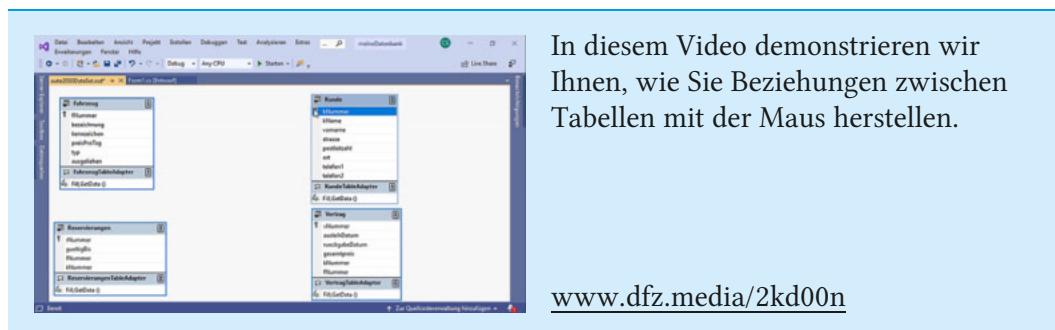


Abb. 1.10: Die Beziehung zwischen den Tabellen **Kunde** und **Vertrag** (die Linie für die Beziehung verläuft hinter der Darstellung der anderen beiden Tabellen)

Die Linie für die Beziehung wird jetzt blau und schraffiert dargestellt. Am einen Ende der Linie an der Tabelle **Kunde** finden Sie einen kleinen Schlüssel und am anderen Ende eine liegende Acht. Diese beiden Symbole markieren die Art der Beziehung – in unserem Fall eine 1 : n-Beziehung. Der Schlüssel steht dabei für die 1 und die liegende Acht für das n.

2. *Cascade* bedeutet übersetzt so viel wie „herunterfallen“ oder „Wasserfall“.

3. *None* bedeutet übersetzt „keine“.



Video 1.1: Beziehungen zwischen Tabellen über den DataSet-Designer herstellen

Erstellen Sie jetzt bitte die Beziehung zwischen der Spalte **fNummer** in der Tabelle **Fahrzeug** und der Spalte **mNummer** in der Tabelle **Vertrag**. Markieren Sie dazu die Spalte **fNummer** in der Tabelle **Fahrzeug** durch einen Mausklick in den Zeilenkopf und ziehen Sie dann den Zeilenkopf auf die Zeile der Spalte **fNummer** in der Tabelle **Vertrag**.

Prüfen Sie anschließend im Fenster **Beziehung** die Einträge für die über- und untergeordnete Tabelle sowie die Spalten. Markieren Sie dann die Option **Sowohl Beziehungs- als auch Fremdschlüsseleinschränkung** im Bereich **Zu erstellendes Element auswählen**. Da auch die Fahrzeugnummer automatisch vergeben wird, setzen Sie den Eintrag für das Feld **Aktualisierungsregel** bitte wieder auf **None**. Klicken Sie abschließend auf die Schaltfläche **OK**.

Die Anzeige im DataSet-Designer sollte nun so aussehen:

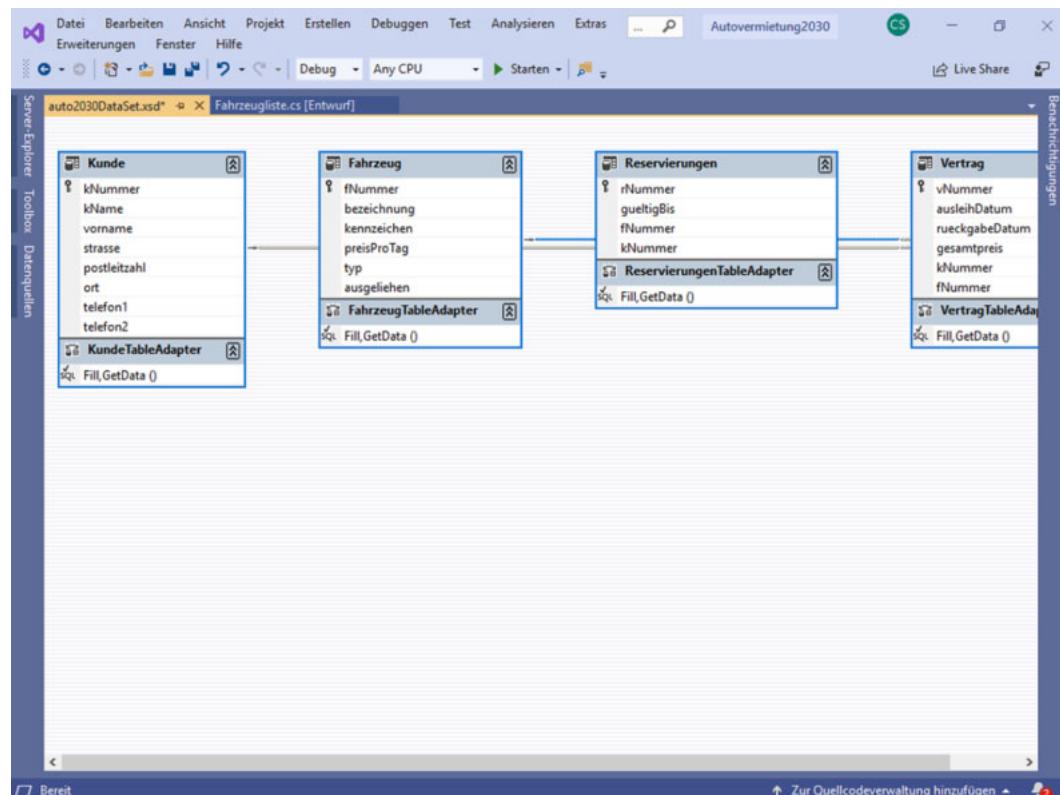


Abb. 1.11: Die neue Beziehung (die Linie oben zwischen den Tabellen **Fahrzeug** und **Vertrag**)

Legen Sie dann noch die Beziehungen zwischen den Tabellen **Kunde** und **Reservierungen** sowie **Fahrzeug** und **Reservierungen** an. Dabei können Sie im Wesentlichen genauso vorgehen wie beim Herstellen der Beziehungen für die Verträge. Sie müssen lediglich für die untergeordnete Tabelle jeweils die Tabelle **Reservierungen** wählen.

Speichern Sie anschließend sämtliche Änderungen.

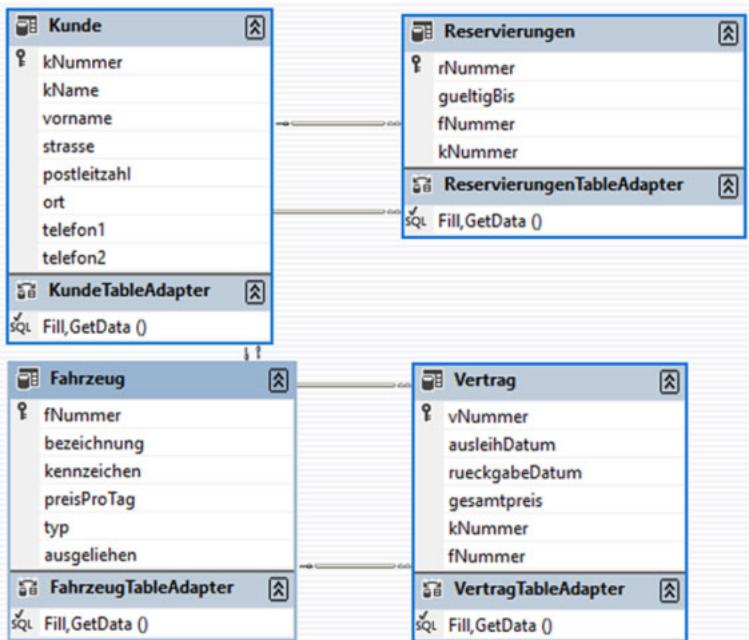


Abb. 1.12: Die Beziehungen zwischen den Tabellen (die Tabellen sind zur besseren Übersicht verschoben)

Tipp:

*Sie können eine Beziehung auch nachträglich bearbeiten. Klicken Sie dazu mit der rechten Maustaste auf die Linie der Beziehung und wählen Sie dann im Kontextmenü die Funktion **Beziehung bearbeiten...***

Um eine Beziehung wieder zu löschen, markieren Sie die Linie der Beziehung und drücken die Taste **Entf**. Alternativ können Sie im Kontext-Menü der Linie auch die Funktion **Löschen** benutzen.

Damit sind die nötigen Änderungen an der Datenbank vollständig. Im nächsten Kapitel werden wir die Formulare zur Datenerfassung für die Verträge und die Reservierungen anlegen.

Zusammenfassung

Über die Normalisierung werden Daten auf mehrere Tabellen verteilt.

Die Verbindung der Tabellen erfolgt über Beziehungen. Ein Primärschlüssel in einer Tabelle wird dabei zum Fremdschlüssel in einer anderen Tabelle.

Beziehungen zwischen Tabellen lassen sich sehr komfortabel mit dem DataSet-Designer erstellen. Dazu ziehen Sie die gewünschte Spalte aus der einen Tabelle auf die entsprechende Spalte der anderen Tabelle. Im folgenden Fenster können Sie dann noch Details für die Beziehung festlegen.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Nennen Sie drei mögliche Beziehungen zwischen Datenbanktabellen.
Beschreiben Sie diese Beziehungen kurz.

- 1.2 Was ist die referentielle Integrität?

- 1.3 Sie haben in einer Datenbank eine neue Tabelle angelegt, die aber nicht bei den Datenquellen erscheint. Was haben Sie wahrscheinlich vergessen?

- 1.4 Beschreiben Sie kurz, wie Sie mit dem DataSet-Designer eine Beziehung zwischen zwei Tabellen herstellen.

- 1.5 Sie haben in zwei Tabellen Spalten mit demselben Namen und unterschiedlichem Datentyp angelegt. Können Sie diese Spalten benutzen, um eine Beziehung zwischen den Tabellen herzustellen? Begründen Sie bitte Ihre Antwort.

2 Die Verträge

In diesem Kapitel werden wir die Formulare für die Verarbeitung der Verträge erstellen. Dabei sollen Fahrzeuge vermietet und zurückgegeben werden können.

2.1 Das Formular für die Verträge

Anders als bei den bisherigen Formularen gibt es beim Formular für die Verträge eine Besonderheit. Da die Tabelle über die Kundennummer und die Fahrzeugnummer mit anderen Tabellen verbunden ist, dürfen hier in die entsprechenden Felder nur Daten eingegeben werden, die bereits in einer der Tabellen vorhanden sind. Sie könnten nun wie gewohnt eine DataGridView oder eine Detailansicht erstellen und dem Anwender die korrekte Dateneingabe komplett allein überlassen. Er muss sich dann die gewünschte Kunden- und Fahrzeugnummer vor der Eingabe des Vertrags merken und gegebenenfalls eben notieren.

Besonders komfortabel ist das aber nicht. Außerdem werden sich Fehler nicht immer verhindern lassen – zum Beispiel dann, wenn eben doch versehentlich eine nicht vorhandene Kunden- oder Fahrzeugnummer eingegeben wird. Wir gehen deshalb einen anderen Weg und erstellen im Formular für die Verträge neben den Feldern für die Vertragsnummer und das Ausleih- und Rückgabedatum zwei Kombinationsfelder, die direkt die vorhandenen Kunden- und Fahrzeugnummern zur Auswahl anbieten.

Hinweis:

Der Einfachheit halber erfassen wir jedes vermietete Fahrzeug in einem eigenen Vertrag.

Legen Sie bitte zunächst wie gewohnt mit der Funktion **Projekt/Windows Forms hinzufügen...** ein neues Formular an. Als Namen können Sie zum Beispiel **Vertrageinzel** verwenden. Öffnen Sie dann im Register mit den Datenquellen den Zweig für die Tabelle **Kunde**.

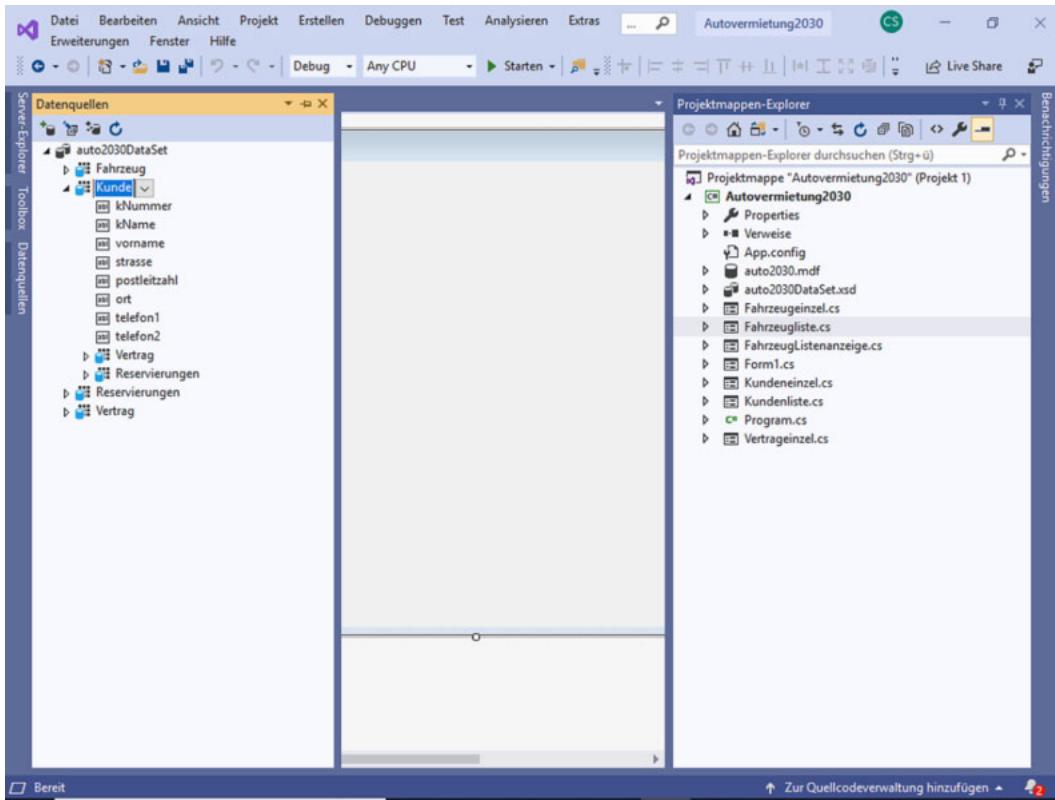


Abb. 2.1: Die Tabelle **Kunde** mit den untergeordneten Tabellen

Unten in dem Zweig sollten jetzt auch die untergeordneten Tabellen **Vertrag** und **Reservierungen** angezeigt werden, zu denen wir eine Beziehung hergestellt haben.

Bitte beachten Sie unbedingt:

Das Erstellen der Auswahllisten funktioniert nur dann korrekt, wenn Sie die Daten über die untergeordneten Tabellen im Zweig der Tabelle **Kunde** verarbeiten. Wenn Sie die Tabellen aus der obersten Ebene verwenden, können die Listen nicht erstellt werden.



Anders als bisher werden wir jetzt für die Tabelle **Vertrag** die Steuerelemente selbst festlegen. Öffnen Sie bitte den Zweig für die untergeordnete Tabelle **Vertrag** im Zweig der Tabelle **Kunde**. Klicken Sie dann auf den Eintrag für das Feld **kNummer** und öffnen Sie die Liste durch einen Mausklick auf das Symbol hinter dem Eintrag.

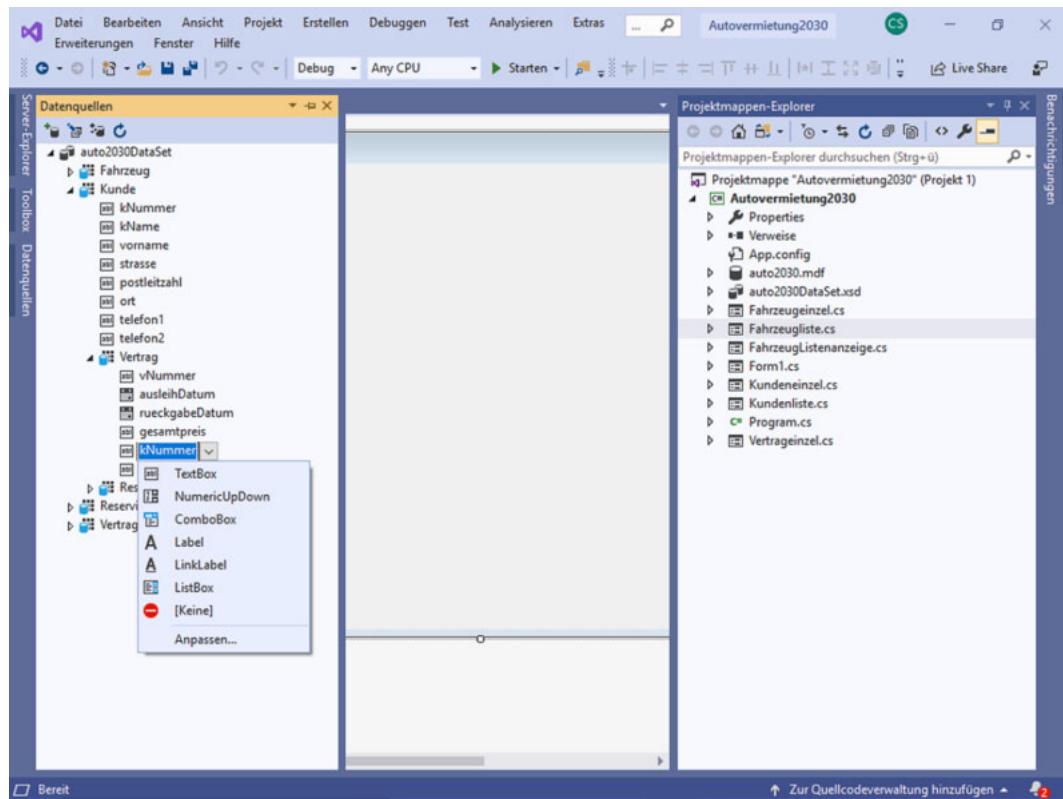


Abb. 2.2: Die Auswahlmöglichkeiten für das Feld **kNummer** der untergeordneten Tabelle **Vertrag**

Wählen Sie anschließend in der Liste den Eintrag **ComboBox** aus. Wiederholen Sie diesen Schritt dann auch für das Feld **fNummer**.

Markieren Sie danach den Eintrag der untergeordneten Tabelle **Vertrag** im Zweig der Tabelle **Kunde** und wählen Sie über das Symbol die Ansicht **Details** aus. Ziehen Sie dann die untergeordnete Tabelle **Vertrag** mit gedrückter linker Maustaste in das Formular und legen Sie sie an der gewünschten Position ab.

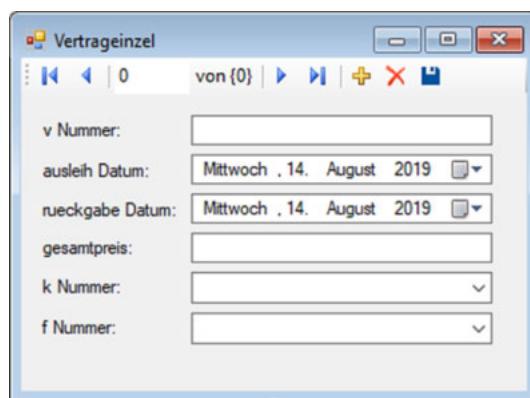


Abb. 2.3: Die Detailansicht mit den ComboBoxen

Visual Studio erstellt nun wie gewohnt die verschiedenen Steuerelemente.

Bitte beachten Sie:

Die Änderungen an der Darstellung der Steuerelemente für die Ansicht **Details** werden von Visual Studio gespeichert. Wenn Sie also in einem anderen Formular die Steuerelemente für den Kunden und die Fahrzeuge wieder als normales Eingabefeld anzeigen lassen wollen, müssen Sie zuerst wieder die Darstellung ändern.



Im nächsten Schritt müssen Sie jetzt dafür sorgen, dass die Daten im Steuerelement **kNummerComboBox** im Formular für die Verträge aus der Tabelle mit den Kunden übernommen werden. Dazu ziehen Sie den Eintrag für die Tabelle **Kunde** mit der Maus aus der Baumstruktur mit den Datenquellen auf das gewünschte Steuerelement.

Bitte beachten Sie:

Sie müssen den Eintrag der kompletten Tabelle **Kunde** aus der Ansicht der Datenquellen auf das Steuerelement ziehen. Wenn Sie nur das Feld **kNummer** aus der Tabelle **Kunde** auf das Steuerelement ziehen, funktioniert die Anzeige gleich nicht korrekt.

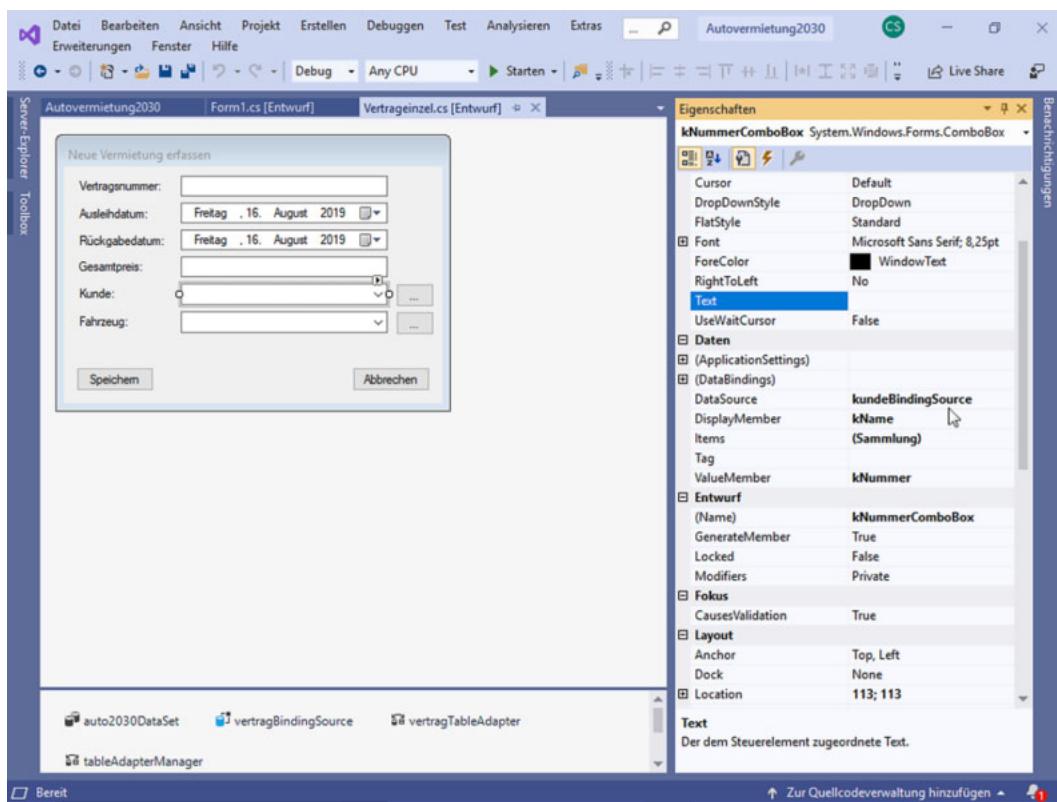


Abb. 2.4: Die Verbindung zur Tabelle **Kunde** (rechts im Eigenschaftenfenster)

Überprüfen Sie nach dem Ablegen bitte auch noch einmal im Eigenschaftenfenster, ob die folgenden Eigenschaften für das Steuerelement **kNummerComboBox** richtig gesetzt wurden. Sie finden sämtliche Eigenschaften aus der folgenden Tabelle in der Gruppe **Daten** des Eigenschaftenfensters.

Tab. 2.1: Die geänderten Eigenschaften für das Steuerelement **kNummerComboBox**

Eigenschaft	Wert
DataSource	kundeBindingSource
DisplayMember	kName
ValueMember	kNummer
(DataBindings)/SelectedValue	vertragBindingSource – kNummer

Die Eigenschaft **DataSource** legt dabei die Datenquelle fest – in unserem Beispiel also die Schnittstelle für die Kommunikation mit der Tabelle **Kunde**.

Die Eigenschaften **DisplayMember** und **ValueMember** legen fest, welcher Wert in dem Steuerelement angezeigt wird (**DisplayMember**) und welcher Wert tatsächlich in das Steuerelement eingetragen wird (**ValueMember**).⁴ In unserem Fall erscheint in der Liste der Name des Kunden, eingetragen wird aber die Kundennummer – also genau der Wert, den wir auch in der Tabelle mit den Verträgen benötigen.

Die Eigenschaft **(DataBindings)/SelectedValue** schließlich steuert, in welcher Spalte der Wert beim Speichern abgelegt wird. Das ist in unserem Fall die Spalte **kNummer** in der Tabelle, die über **vertragBindingSource** angesprochen wird – in unserem Beispiel also die Tabelle **Vertrag**.

Stellen Sie jetzt auch die Verbindung zwischen dem Steuerelement **fNummerComboBox** und der Tabelle **Fahrzeug** her. Ziehen Sie dazu den Eintrag der kompletten Tabelle mit den Fahrzeugdaten aus der Baumstruktur mit den Datenquellen auf das dazugehörige Steuerelement im Formular.

Da sich die Tabelle mit den Fahrzeugen nicht im selben Zweig wie die Tabelle **Kunde** und die untergeordneten Tabellen befindet, werden jetzt möglicherweise nur die Eigenschaften **DataSource**, **DisplayMember** und **ValueMember** korrekt gesetzt. Die Verbindung zur Tabelle mit den Verträgen müssen Sie dagegen unter Umständen selbst herstellen.

Wenn bei der Eigenschaft **(DataBindings)/SelectedValue** nicht der Wert **vertragBindingSource – fNummer** angezeigt wird, öffnen Sie bitte die Auswahlliste für das Feld. Klicken Sie anschließend in der Liste auf das Symbol vor dem Eintrag **vertragBindingSource** und wählen Sie die Spalte **fNummer** aus.

4. Übersetzt bedeuten die beiden Eigenschaften so viel wie „Anzeigemitglied“ und „Wertmitglied“.

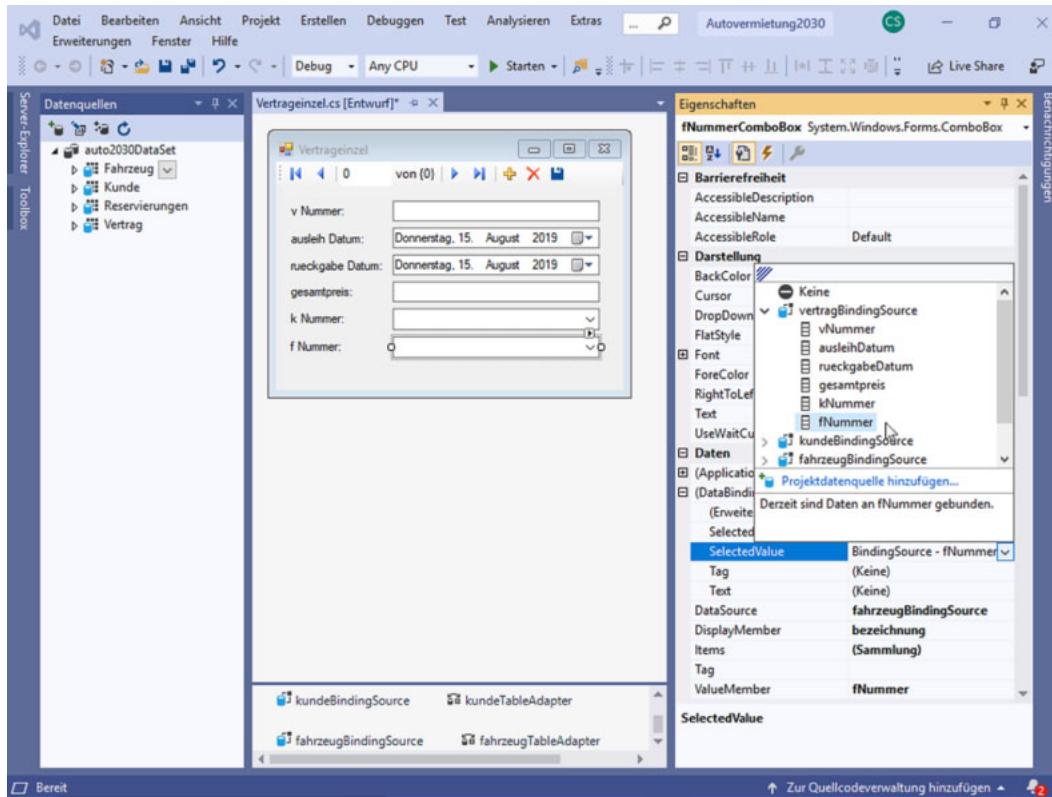


Abb. 2.5: Die Liste der Spalten für vertragBindingSource

Speichern Sie dann sämtliche Änderungen und führen Sie einen ersten Test durch.
Fügen Sie im Formular **Form1** eine Schaltfläche zum Anzeigen des neuen Formulars ein und lassen Sie das neue Formular beim Anklicken dieser Schaltfläche wie gewohnt modal anzeigen.

Hinweis:

Um den Feinschliff an dem Formular werden wir uns gleich kümmern.

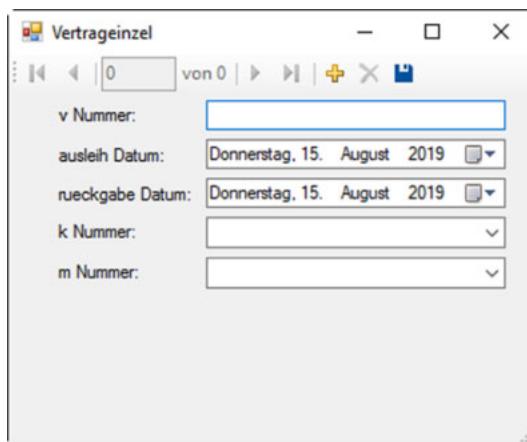


Abb. 2.6: Das Formular für die Verträge im praktischen Einsatz⁵

5. Die Werte in den beiden Datumsfeldern hängen vom aktuellen Datum ab.



Bitte beachten Sie:

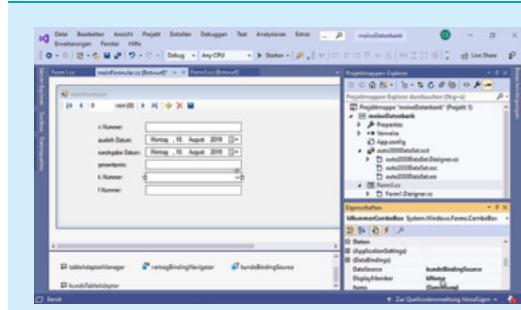
In den Datumsfeldern wird das aktuelle Datum zwar angezeigt, aber nicht automatisch übernommen. Wenn Sie beim Speichern eine Fehlermeldung erhalten, dass die Spalte **ausleihDatum** oder die Spalte **rueckgabeDatum** keine Nullen zulässt, wählen Sie das Datum bitte über das Kombinationsfeld noch einmal aus. Möglicherweise müssen Sie dabei auch zunächst ein anderes Datum auswählen, bevor Sie wieder das aktuelle Datum benutzen können. Dieses seltsame Verhalten werden wir gleich beim Feinschliff abstellen.

Denken Sie bitte daran, dass Visual Studio abhängig von Ihren Einstellungen eine Kopie der Datenbank in dem Ordner mit dem ausführbaren Programm erstellt. In dieser Kopie werden dann auch die Datensätze bearbeitet. Bei Änderungen am Aufbau der Datenbankdateien werden die Kopien im Ordner mit dem ausführbaren Programm allerdings ohne Rückfrage überschrieben und bereits erfasste Daten gehen verloren.

Wie Sie dieses Verhalten gezielt beeinflussen können, haben wir Ihnen ja bereits beim Anlegen des ersten Formulars für die Kunden beschrieben. Lesen Sie dort gegebenenfalls noch einmal nach.

Wenn Sie alles richtig gemacht haben, sollten Sie jetzt über die Kombinationsfelder für den Kunden und das Fahrzeug auf die entsprechenden Einträge in den beiden anderen Tabellen zugreifen können. Probieren Sie das einfach einmal mit mehreren Verträgen aus. Legen Sie dazu jeweils einen neuen Datensatz über das Symbol **Neu hinzufügen** an.

Denken Sie aber bitte daran, dass die Daten nicht automatisch gesichert werden. Um die Änderungen in die Tabelle zu übernehmen, müssen Sie vor dem Schließen des Formulars einmal auf das Symbol **Daten speichern** klicken.



In diesem Video zeigen wir Ihnen, wie Sie Werte aus einer verbundenen Tabelle über ein Kombinationsfeld abrufen.

www.dfz.media/368glv

Video 2.1: Daten aus einer verbundenen Tabelle über ein Kombinationsfeld anzeigen

Damit das Formular für die Vermietung etwas netter aussieht, nehmen wir jetzt noch ein wenig Feinschliff vor. Schließen Sie die Anwendung bitte wieder und wechseln Sie in den Entwurf für das Formular **Vertrageinzel**.

Ändern Sie zuerst den Text in der Titelleiste des Formulars in **Neue Vermietung erfassen**. Überarbeiten Sie dann auch die Texte vor den fünf Steuerelementen für die Dateneingabe. Es handelt sich jeweils um Labels, die Sie einzeln bearbeiten können.

Setzen Sie anschließend die Eigenschaft **Enabled** in der Gruppe **Verhalten** für das Steuerelement **vNummerTextBox** auf **False**. Damit kann der Anwender selbst keine Eingaben mehr in dem Feld machen beziehungsweise die Einfügemarkierung nicht mehr in das Feld stellen.

Tipp:

*Für die Steuerelemente mit den Datumsanzeigen können Sie sowohl die Anzeige als auch die Auswahl verändern. Wenn Sie die Eigenschaft **Format** in der Gruppe **Darstellung** zum Beispiel auf **Short** setzen, wird nur das eigentliche Datum ohne den Wochentag angezeigt. Mit der Eigenschaft **ShowUpDown** in der Gruppe **Darstellung** können Sie festlegen, dass die Veränderung über ein Drehfeld und nicht mehr über ein Kombinationsfeld erfolgt. Welche Einstellungen Sie hier benutzen, ist im Wesentlichen Geschmackssache. Auswirkungen auf die Funktionalität haben die Änderungen nicht.*

Verschieben Sie dann die Anzeige der Steuerelemente gegebenenfalls so, dass alles korrekt dargestellt wird. Verkleinern Sie außerdem das Formular. Lassen Sie dabei aber unten bitte Platz für eine Schaltfläche.

Nach diesen Änderungen könnte das Formular zum Beispiel so aussehen:

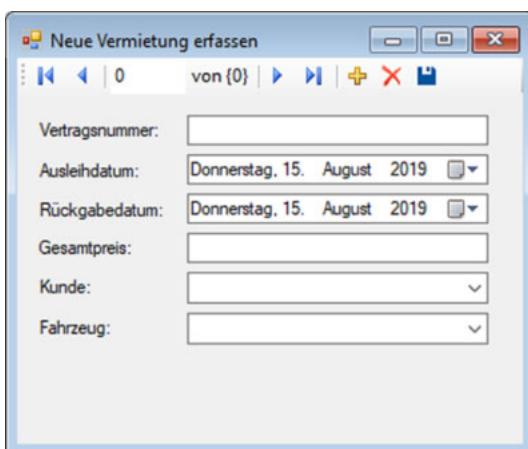


Abb. 2.7: Das überarbeitete Formular für die Vermietung

Im nächsten Schritt sorgen wir jetzt dafür, dass beim Öffnen des Formulars automatisch ein neuer Datensatz angelegt wird und auch Vorschläge für den Kunden und das Fahrzeug gemacht werden. Dazu rufen wir im Ereignis **Load** für das Formular die Methode **AddNew()**⁶ für die **BindingSource** der Tabelle **Vertrag** auf. Außerdem setzen wir die Anzeige in den Steuerelementen **kNummerComboBox** und **fNummerComboBox** auf den ersten Wert in der Liste. Das erfolgt durch die Zuweisung von 0 an die Eigenschaft **SelectedIndex** der beiden Steuerelemente.

Damit auch das Datum korrekt ohne erneute Auswahl übernommen wird, setzen wir die Eigenschaft **Value** der beiden Steuerelemente für die Datumsauswahl. Das Steuerelement **ausleihDatumDateTimePicker** erhält die aktuelle Uhrzeit und das aktuelle Datum. Diese Werte erhalten wir über **DateTime.Now**⁷.

6. AddNew bedeutet übersetzt so viel wie „füge Neuen hinzu“.

7. Now bedeutet übersetzt „jetzt“.

Für das Rückgabedatum addieren wir einen Tag auf das Ausleihdatum und lassen dann diesen Wert als Vorschlagswert im Steuerelement **rueckgabeDatumDateTimePicker** anzeigen. Die Addition ist allerdings nicht direkt möglich, sondern muss über Methode `AddDays()`⁸ der Klasse `DateTime` erfolgen.

Erweitern Sie jetzt bitte die Methode `VertragEinzel_Load()` des Formulars um die Anweisungen aus dem folgenden Code. Achten Sie dabei darauf, dass Sie die bereits vorhandenen Anweisungen zum Laden der Daten in die verschiedenen Tabellen des Data-Sets nicht überschreiben.

```

...
//einen neuen Datensatz erzeugen
vertragBindingSource.AddNew();
//die Einträge für den Kunden und das Fahrzeug setzen
kNummerComboBox.SelectedIndex = 0;
fNummerComboBox.SelectedIndex = 0;
//den Wert für das Ausleihdatum setzen
ausleihDatumDateTimePicker.Value = DateTime.Now;
//den Wert für das Rückgabedatum setzen
//hier 30 Tage auf das aktuelle Datum addieren
rueckgabeDatumDateTimePicker.Value = DateTime.Now.AddDays(30);
```

Code 2.1: Das Erzeugen eines neuen Datensatzes und das Setzen der Standardwerte

Da jetzt automatisch ein neuer Datensatz erzeugt wird, lassen wir die Navigationsleiste oben im Formular verschwinden. Das Speichern soll dann über eine eigene Schaltfläche erfolgen, die das Formular außerdem automatisch schließt.

Löschen Sie zuerst die Navigationsleiste. Das geht am einfachsten, wenn Sie das Steuer-element **vertragBindingNavigator** unten im Formular markieren und anschließend die Taste **Entf** drücken. Fügen Sie dann unten im Formular eine Schaltfläche mit der Be-schriftung **Speichern** ein. Beim Anklicken dieser Schaltfläche lassen Sie die Änderun-ge in der Datenbank speichern und schließen das Formular. Die entsprechen-de Metho-de sieht so aus:

```

private void ButtonSpeichern_Click(object sender, EventArgs e)
{
    this.Validate();
    this.vertragBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(this.auto2030DataSet);

    Close();
}
```

Code 2.2: Das Speichern der Änderungen und das Schließen des Formulars

Tipp:

Die ersten drei Zeilen können Sie aus der Methode

VertragBindingNavigatorSaveItem_Click() *kopieren. Diese Methode ist nach wie vor vorhanden, obwohl die Navigationsleiste gelöscht ist. Sie können sie nach dem Kopieren der Anweisungen aber gefahrlos löschen.*

8. AddDays lässt sich mit „füge Tage hinzu“ übersetzen.

Damit beim Schließen des Formulars über die Schließen-Schaltfläche in der Titelleiste keine Daten verloren gehen können, blenden Sie die Symbole einfach aus. Dazu setzen Sie die Eigenschaft **ControlBox** im Bereich **Fensterstil** des Formulars auf `False`.

Damit der Anwender das Formular aber auch wieder ohne Speichern verlassen kann, wenn er es versehentlich geöffnet hat, sollten Sie jetzt im letzten Schritt noch eine Schaltfläche **Abbrechen** einfügen. Diese Schaltfläche macht nichts weiter, als das Formular wieder zu schließen.

Speichern Sie dann alle Änderungen und testen Sie das Formular noch einmal.

The screenshot shows a Windows-style dialog box titled "Neue Vermietung erfassen". It contains several input fields: "Vertragsnummer:" with value "-1", "Ausleihdatum:" showing "Donnerstag, 15. August 2019" with a calendar icon, "Rückgabedatum:" showing "Freitag, 16. August 2019" with a calendar icon, "Gesamtpreis:" (total price) with an empty field, "Kunde:" dropdown menu showing "Meier", and "Fahrzeug:" dropdown menu showing "Volvo V70". At the bottom left is a "Speichern" button and at the bottom right is an "Abbrechen" button.

Abb. 2.8: Das überarbeitete Formular im praktischen Einsatz

Hinweis:

Auf die Prüfung der Felder können wir beim Formular der Verträge verzichten. Die meisten Steuerelemente haben ja in jedem Fall Werte und können daher nicht leer sein. Das automatische Berechnen des Gesamtpreises werden wir gleich noch programmieren.

2.2 Die Detailanzeige für Kunden und Fahrzeuge

Durch die Verbindung der beiden Kombinationsfelder für die Kunden und die Fahrzeuge mit den jeweiligen Tabellen haben wir jetzt zwar sichergestellt, dass nur gültige Daten eingegeben werden können, allerdings ist die Auswahl nicht sonderlich komfortabel. Da wir zum Beispiel bei den Kunden nur den Namen anzeigen lassen, lassen sich zwei Kunden mit identischen Namen nicht mehr eindeutig unterscheiden. Eine Möglichkeit wäre es nun, statt des Namens die Kundennummer anzuzeigen. Dann müsste der Anwender allerdings genau wissen, welche Nummer zu welchem Kunden gehört. Auch das ist nicht sonderlich benutzerfreundlich.

Wir gehen deshalb einen anderen Weg und stellen dem Anwender im Formular für die Verträge zusätzliche Schaltflächen zur Verfügung, über die er die vollständigen Informationen zum ausgewählten Kunden und zum ausgewählten Fahrzeug anzeigen lassen kann. Dazu erstellen wir jeweils ein neues Formular, das sich die Daten über eine Abfrage aus der jeweiligen Tabelle beschafft.

Beginnen wir mit der Detailanzeige für die Kunden. Erstellen Sie bitte ein neues leeres Formular mit dem Namen **Kundendetail**. Blenden Sie dann gegebenenfalls die Datenquellen ein und legen Sie für die Tabelle **Kunde** die Ansicht **Details** fest. Ziehen Sie anschließend die gesamte Tabelle in das Formular. Visual Studio erstellt dann wie gewohnt die verschiedenen Steuerelemente.

Löschen Sie im nächsten Schritt bitte die Navigationsleiste aus dem Formular und setzen Sie die Eigenschaft **ReadOnly** in der Gruppe **Verhalten** für alle Felder auf `True`. Damit ist sichergestellt, dass der Anwender keine Eingaben machen kann. Fügen Sie anschließend eine Schaltfläche **Schließen** unten im Formular ein und blenden Sie die Schaltflächen in der Titelleiste des Formulars aus. Sorgen Sie dann dafür, dass beim Anklicken der Schaltfläche **Schließen** das Formular wieder geschlossen wird, und ändern Sie die Texte vor den Eingabefeldern. Im letzten Schritt setzen Sie den Text in der Titelleiste des Formulars noch auf **Detailanzeige Kunden** und passen die Größe an.

Das Formular sollte danach ungefähr so aussehen:

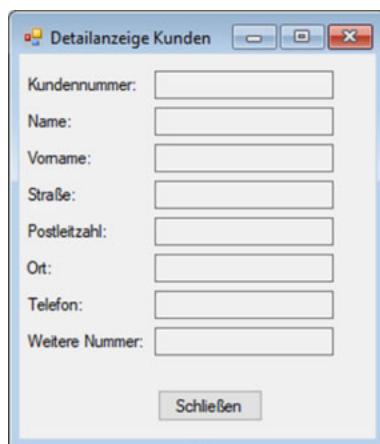


Abb. 2.9: Das Formular für die Detailanzeige der Kunden

Da wir in diesem Formular nicht die Daten aller Kunden anzeigen lassen wollen, müssen wir nun noch eine **Abfrage** für die Tabelle formulieren.



Abfragen selektieren anhand bestimmter Kriterien Daten aus einer Datenbank.

Das erfolgt sehr komfortabel über einen Assistenten des Tabellen-Designers. Klicken Sie bitte im Register der Datenquellen auf das Symbol **DataSet mit Designer bearbeiten** . Alternativ können Sie auch im Projektmappen-Explorer auf die Datei **auto2030DataSet.xsd** doppelklicken.

Klicken Sie anschließend mit der rechten Maustaste auf den Bereich **KundeTableAdapter** in der Tabelle **Kunde** und wählen Sie im Kontextmenü die Funktion **Hinzufügen/Abfrage...** beziehungsweise **Abfrage hinzufügen...** Visual Studio startet dann den Konfigurations-Assistenten für TableAdapter-Abfragen.

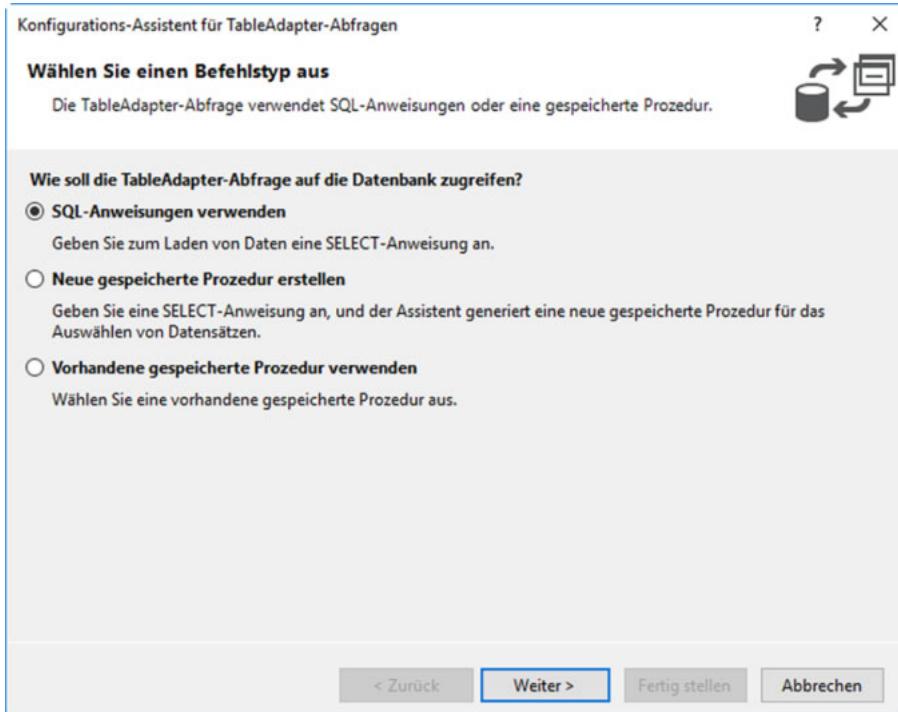


Abb. 2.10: Der Konfigurations-Assistent für TableAdapter-Abfragen

Da wir in unserem Beispiel eine SQL-Anweisung zum Erstellen der Abfrage verwenden werden, können Sie im ersten Schritt direkt auf die Schaltfläche **Weiter >** klicken.

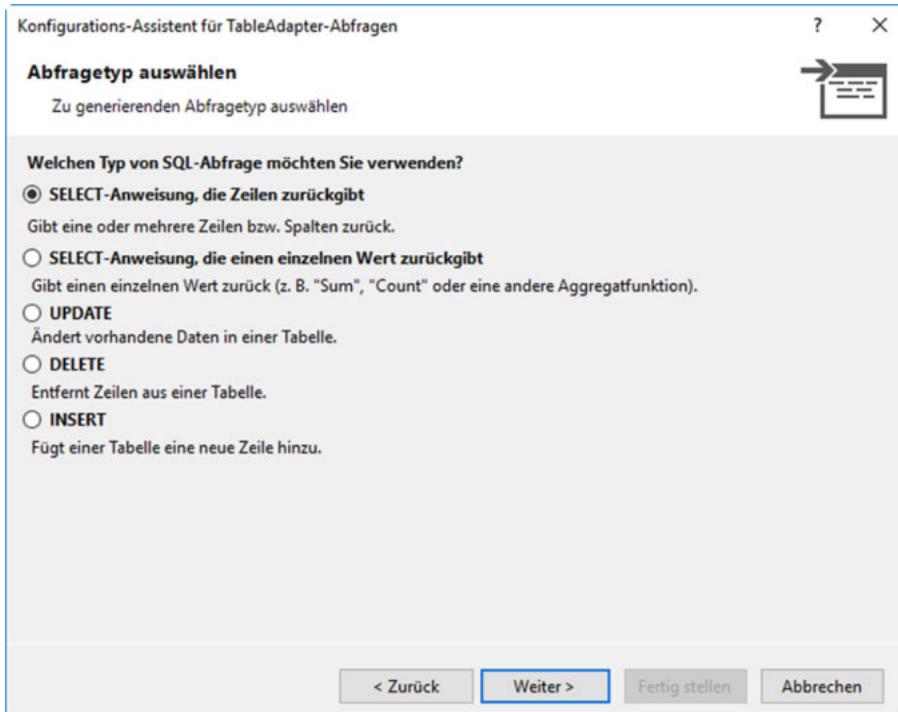


Abb. 2.11: Der zweite Schritt des Assistenten

Auch im zweiten Schritt sind keine Änderungen erforderlich. Wir wollen eine komplette Zeile aus der Tabelle zurückliefern lassen. Die passende Option **SQL-Anweisung, die Zeilen zurückgibt** ist bereits markiert. Klicken Sie noch einmal auf **Weiter >**.

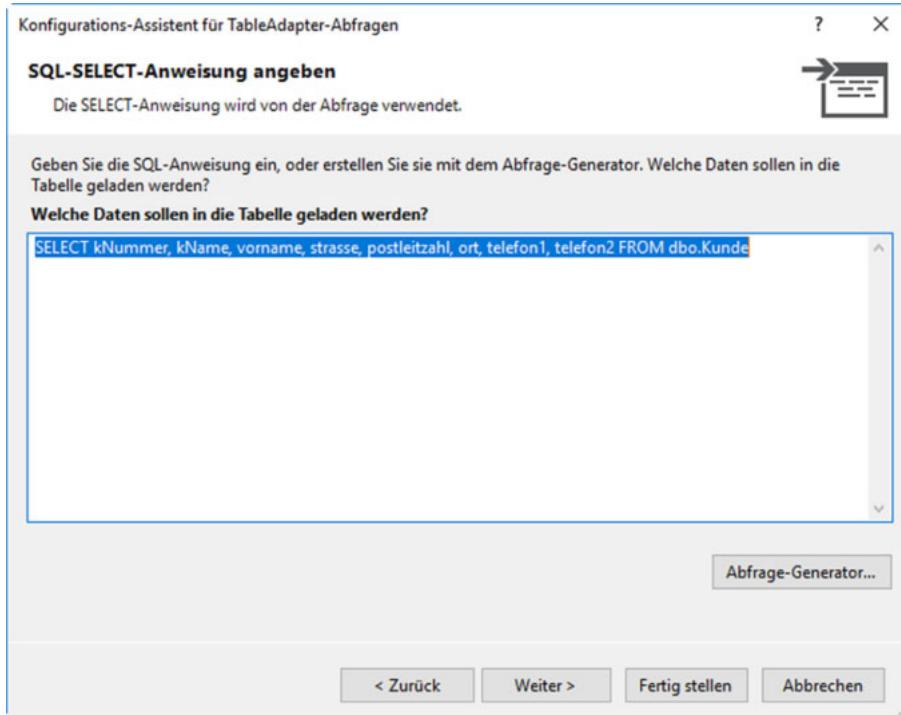


Abb. 2.12: Der dritte Schritt des Assistenten

Im dritten Schritt formulieren Sie jetzt die SQL-Anweisung, die die Daten beschaffen soll. Unten in dem Feld schlägt Ihnen der Assistent bereits eine SELECT-Anweisung vor, die sämtliche Daten aus allen Feldern der Tabelle **Kunde** liefert. Diese Anweisung müssen wir nur noch ein wenig ergänzen – und zwar um eine Bedingung, die nur die Zeilen berücksichtigt, bei denen die Kundennummer mit einem variablen Wert übereinstimmt. Diesen variablen Wert übergeben wir später beim Aufruf der Abfrage.

Ergänzen Sie jetzt bitte den Ausdruck `WHERE kNummer = @kNummer` am Ende der SELECT-Anweisung. Das Zeichen `@` vor dem zweiten Namen `kNummer` kennzeichnet dabei eine Variable. Achten Sie bitte unbedingt darauf, dass Sie dieses Zeichen mit eingeben. Die vollständige Anweisung sollte so aussehen wie in der Abb. 2.13.

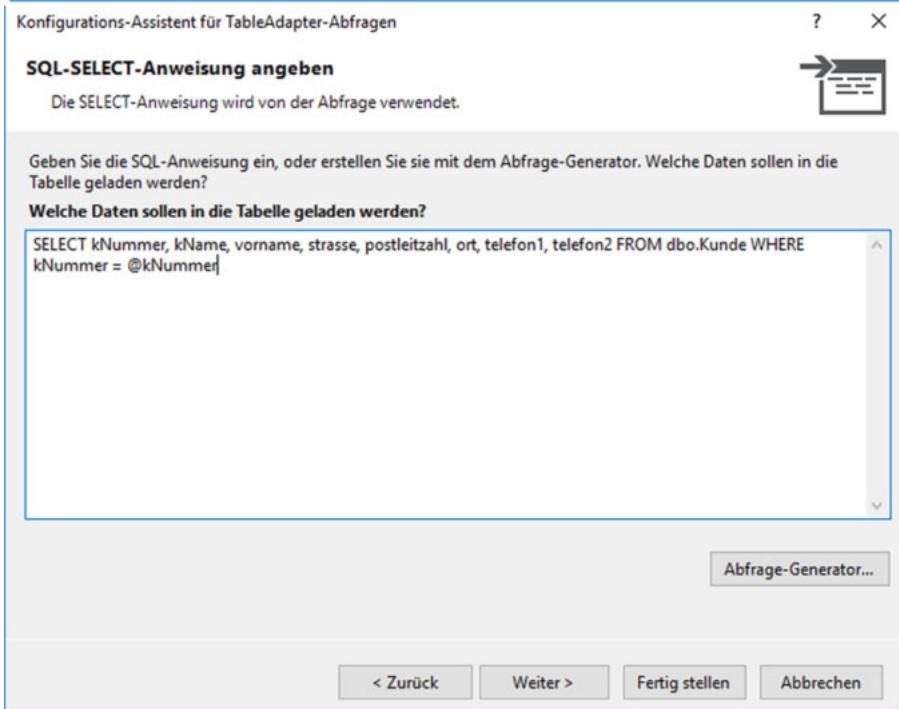


Abb. 2.13: Die SELECT-Anweisung für das Beschaffen der Kundendaten

Klicken Sie dann auf **Weiter >**.

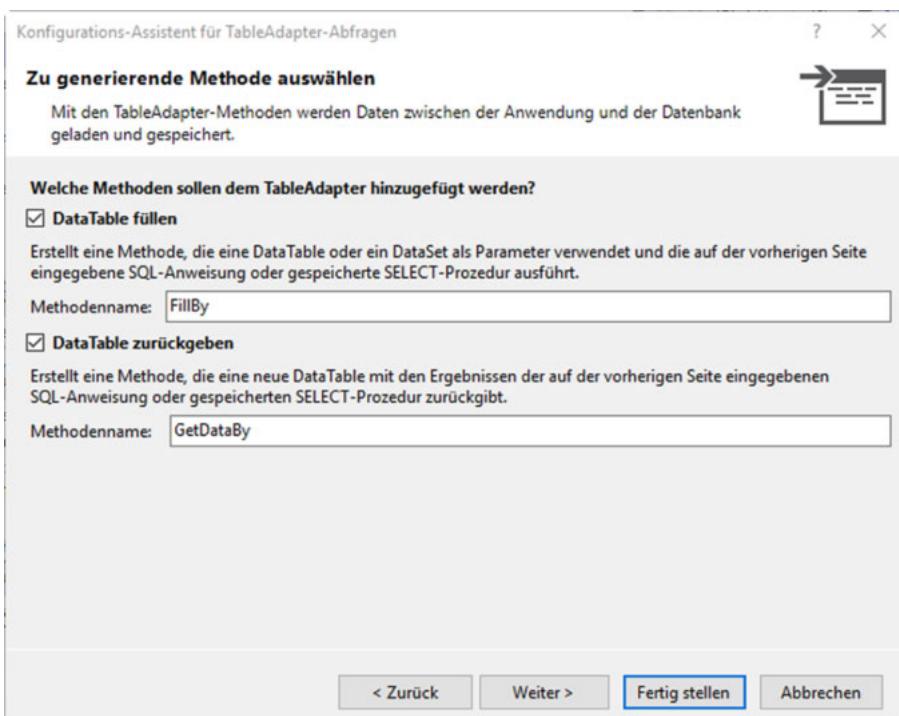


Abb. 2.14: Der vierte Schritt des Assistenten

Im vierten Schritt des Assistenten legen Sie fest, wie die Daten zurückgegeben werden sollen. In unserem Fall lassen wir eine Methode erstellen, die eine DataTable füllt. Diese Methode können wir dann später aus dem Quelltext des Formulars aufrufen.

Geben Sie bitte im Feld **Methodenname:** in der Mitte des Formulars den Namen **FillBykundeDetail** an. Schalten Sie anschließend die Markierung im Feld **DataTable zurückgeben** aus und klicken Sie auf die Schaltfläche **Fertig stellen**.

Der Tabellen-Designer erstellt dann die Methode zum Aufruf und zeigt sie unten im Kasten für die Tabelle **Kunde** an.

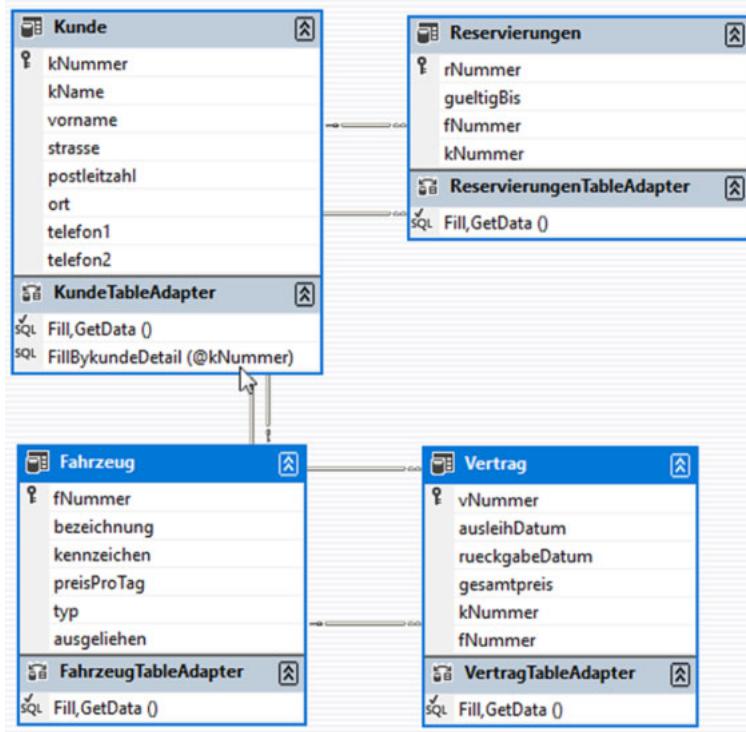


Abb. 2.15: Die Methode `FillBykundeDetail()` im Tabellen-Designer
(unten im Bereich **KundeTableAdapter**)

Jetzt müssen wir noch dafür sorgen, dass das Formular für die Detailanzeige der Kunden seine Daten nicht mehr beim Laden beschafft, sondern über eine eigene Methode. Diese Methode können wir dann aus dem Formular zum Erfassen der Verträge vor dem Anzeigen des eigentlichen Formulars für die Detailanzeige aufrufen.



Zur Erinnerung:

Sie können direkt nach dem Erzeugen eines Formulars bereits Methoden des Formulars aufrufen – ohne das Formular selbst anzuzeigen. Diese Technik haben wir ja bereits an anderer Stelle eingesetzt.

Wechseln Sie bitte in den Quelltext des Formulars **Kundendetail**. Erstellen Sie dort folgende eigene Methode – zum Beispiel direkt oberhalb des Konstruktors der Klasse:

```
public void DetailsLaden(int nummer)
{
    //bitte in einer Zeile eingeben
    this.kundeTableAdapter.FillBykundeDetail(this.
        auto2030DataSet.Kunde, nummer);
}
```

Code 2.3: Die Methode zum Laden der Daten

Mit der Anweisung wird die gerade erstellte Methode `FillBykundeDetail()` aufgerufen. Als Argumente übergeben wir dabei den Namen der Tabelle und die Nummer, nach der gesucht werden soll. Diese Nummer erhält die Methode `DetailsLaden()` selbst als Parameter.

Hinweis:

Sie können die Anweisung aus der Methode für das Ereignis **Load** des Formulars in die eigene Methode kopieren und dann entsprechend anpassen. Da die Daten nicht mehr beim Laden des Formulars beschafft werden sollen, müssen Sie auch auf das Ereignis **Load** des Formulars nicht mehr reagieren. Bitte löschen Sie deshalb im Eigenschaftenfenster den Eintrag beim Ereignis **Load** für das Formular und kommentieren Sie zur Sicherheit auch die Anweisung für das Ereignis aus. Andernfalls funktioniert die Anzeige gleich nicht.

Jetzt fehlt nur noch das Anzeigen des Detailformulars für die Kunden aus dem Formular für die Verträge. Speichern Sie bitte alle Änderungen und wechseln Sie in das Formular für die Verträge. Verbreitern Sie das Formular gegebenenfalls ein wenig und fügen Sie hinter dem Kombinationsfeld für den Kunden eine Schaltfläche mit drei Punkten ein. Lassen Sie dann beim Anklicken der Schaltfläche die folgenden Anweisungen ausführen:

```
Kundendetail detailAnzeigeKunden = new Kundendetail();
//bitte in einer Zeile eingeben
detailAnzeigeKunden.DetailsLaden(Convert.ToInt32
(kNummerComboBox.SelectedValue));
detailAnzeigeKunden.ShowDialog();
```

Code 2.4: Die Anweisungen für die Anzeige des Detailformulars

Zunächst einmal erzeugen wir eine neue Instanz für das Formular **Kundendetail**. Das kennen Sie ja bereits zur Genüge. Danach rufen wir dann die Methode `DetailsLaden()` der Instanz `detailAnzeigeKunden` auf. Dabei übergeben wir den aktuell ausgewählten Wert aus dem Kombinationsfeld `kNummerComboBox` als Argument. Da es sich dabei um einen sehr allgemeinen Typ `object` handelt, wandeln wir ihn in einen `Int32`-Typ um.



Zur Erinnerung:

In dem Kombinationsfeld wird zwar der Name des Kunden angezeigt, tatsächlich befindet sich in dem Feld aber die Kundennummer. Das wird durch die Eigenschaft **ValueMember** sichergestellt, die beim Einfügen des Steuerelements auf **kNummer** gesetzt wurde.

Speichern Sie jetzt alle Änderungen und testen Sie das Programm.

Wenn Sie alles richtig gemacht haben, sollten nach dem Anklicken der Schaltfläche im Formular für die Verträge die Details für den aktuell ausgewählten Kunden angezeigt werden.

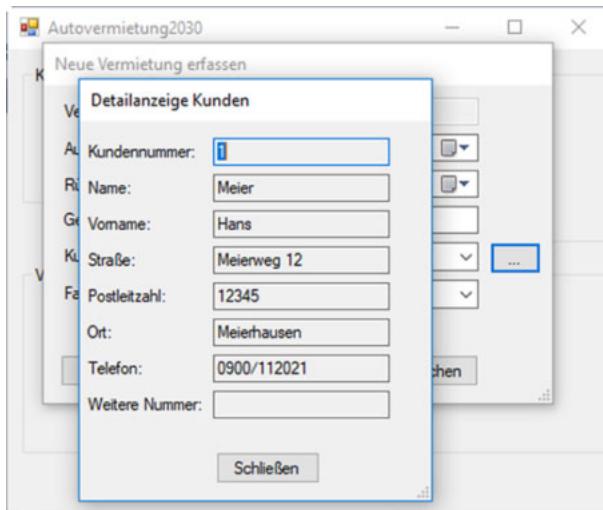


Abb. 2.16: Die Detailanzeige des Kunden im praktischen Einsatz

Die Detailanzeige für die Fahrzeuge erfolgt sehr ähnlich. Legen Sie zunächst ein entsprechendes Formular an und erstellen Sie dann mit dem Assistenten eine Abfrage für die Tabelle **Fahrzeuge**. Die SQL-Anweisung für das Fahrzeug muss dabei so aussehen:

```
SELECT fNummer, bezeichnung, kennzeichen, preisProTag,
ausgeliehen, FROM dbo.Fahrzeug WHERE fNummer = @fNummer
```

Im letzten Schritt des Assistenten vergeben Sie dann wieder einen Namen für die Methode und schalten die Markierung im Feld **DataTable erzeugen** ab.

Danach erstellen Sie ein Detailformular für die Fahrzeuge mit einer Methode zum Beschaffen der Daten und rufen diese Methode beim Klicken auf die Schaltfläche im Formular mit den Verträgen vor dem Anzeigen des Detailformulars auf. Als Argument übergeben Sie den aktuellen Wert aus dem Kombinationsfeld mit dem Fahrzeug. Achten Sie außerdem darauf, dass beim Ereignis **Load** für das Formular kein Eintrag mehr stehen darf. Denn auch hier sollen die Daten ja über die selbst erstellte Methode beschafft werden.

Da es keine weiteren Besonderheiten gibt, stellen wir Ihnen diese Schritte nicht noch einmal im Detail vor. Komplett umgesetzt finden Sie die Detailanzeige für die Fahrzeuge im Projekt für dieses Studienheft.

2.3 Die Vermietung

Das Formular für eine neue Vermietung sieht jetzt zwar ganz ordentlich aus und auch der Zugriff auf die verschiedenen Tabellen funktioniert gut, allerdings gibt es ein großes Manko: Der Anwender muss selbst überprüfen, ob ein Fahrzeug überhaupt vermietet werden kann. Denn das Programm führt in der Tabelle mit den Verträgen bisher ja nur Daten aus unterschiedlichen Quellen zusammen. Ob ein Fahrzeug aktuell vermietet ist oder nicht, lässt sich aber nicht erkennen. Außerdem muss der Anwender den Gesamtpreis selbst berechnen.

Wir müssen also im ersten Schritt beim Speichern eines Eintrags in der Tabelle **Vertrag** auch dafür sorgen, dass der Eintrag des vermieteten Fahrzeugs in der Tabelle **Fahrzeug** geändert wird. Dort müssen wir die Markierung für die Spalte **ausgeliehen** auf `true` setzen. Das hört sich komplizierter an, als es tatsächlich ist. Denn Sie können auch sehr komfortabel Abfragen erstellen, die Daten verändern. Schauen wir uns an, wie Sie dazu vorgehen.

Lassen Sie das Dataset im Designer anzeigen und erstellen Sie über die Funktion **Hinzufügen/Abfrage...** beziehungsweise **Abfrage hinzufügen...** im Kontextmenü des Bereichs **FahrzeugTableAdapter** eine neue Abfrage. Im ersten Schritt des Assistenten können Sie die Einstellungen wieder unverändert übernehmen und direkt auf die Schaltfläche **Weiter >** klicken.

Im zweiten Schritt markieren Sie dann die Option **UPDATE**.

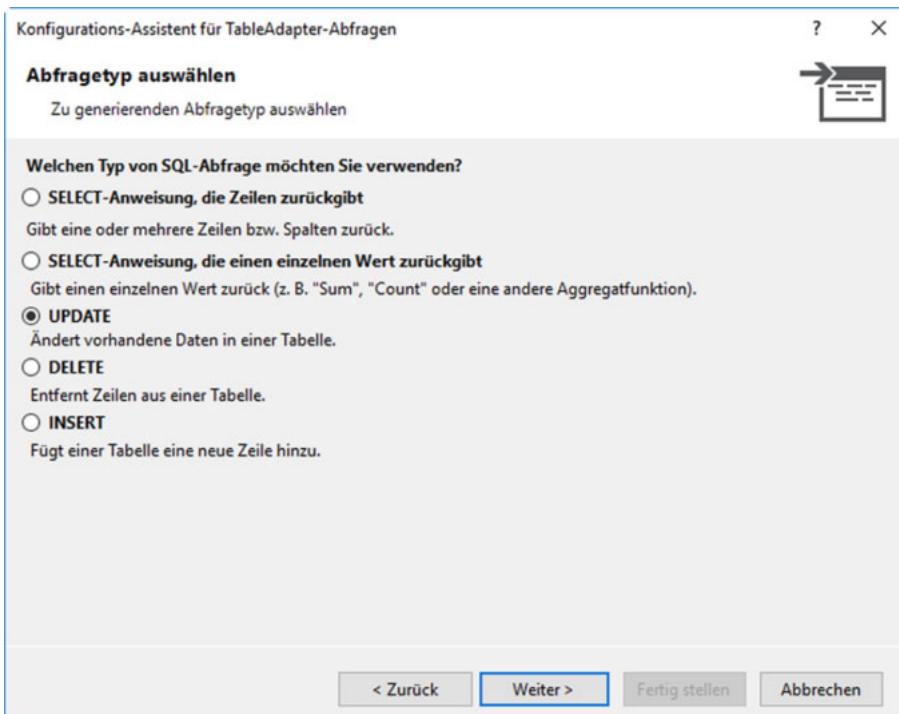


Abb. 2.17: Der zweite Schritt des Assistenten beim Anlegen einer UPDATE-Abfrage

Klicken Sie anschließend auf **Weiter >**.

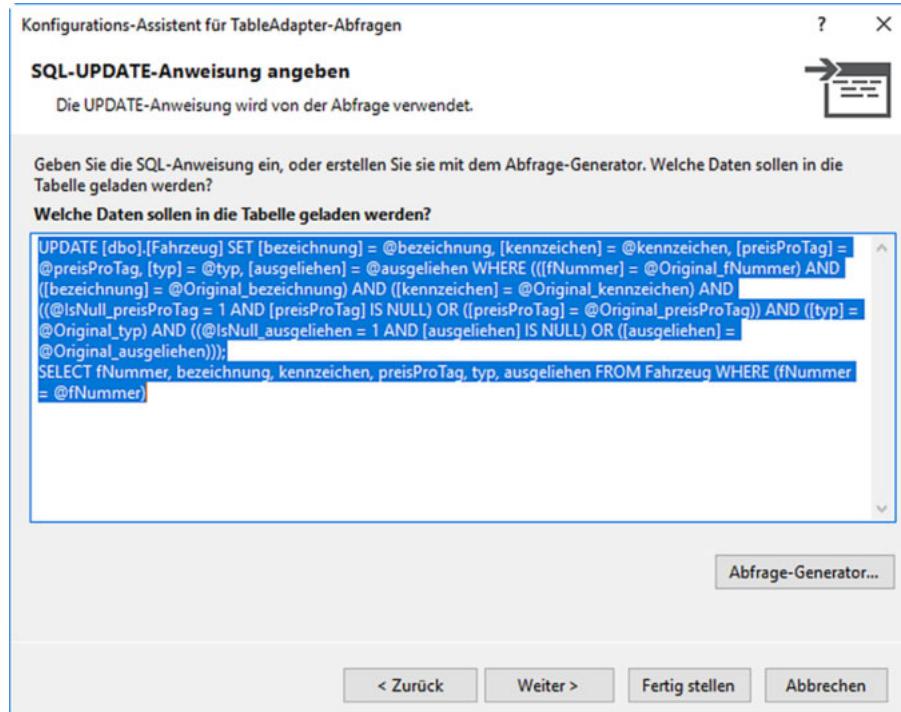


Abb. 2.18: Der dritte Schritt

Im dritten Schritt formulieren Sie die Abfrage über einen SQL-Befehl. Das geht recht einfach über den Abfrage-Generator. Löschen Sie bitte zuerst die Abfrage im Feld in der Mitte des Fensters. Klicken Sie dann auf die Schaltfläche **Abfrage-Generator...** unten rechts.

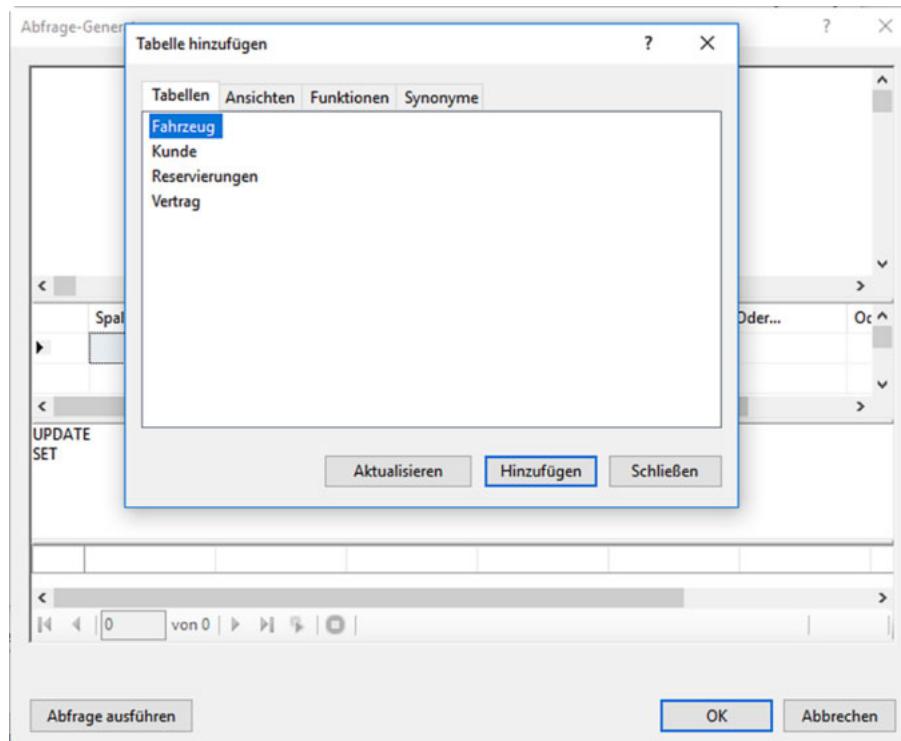


Abb. 2.19: Der Abfrage-Generator

Im Abfrage-Generator wählen Sie zunächst einmal die gewünschte Tabelle aus. Die Tabelle **Fahrzeug** ist in unserem Beispiel bereits markiert. Klicken Sie also auf die Schaltfläche **Hinzufügen**. Schließen Sie danach das Fenster **Tabelle hinzufügen**.

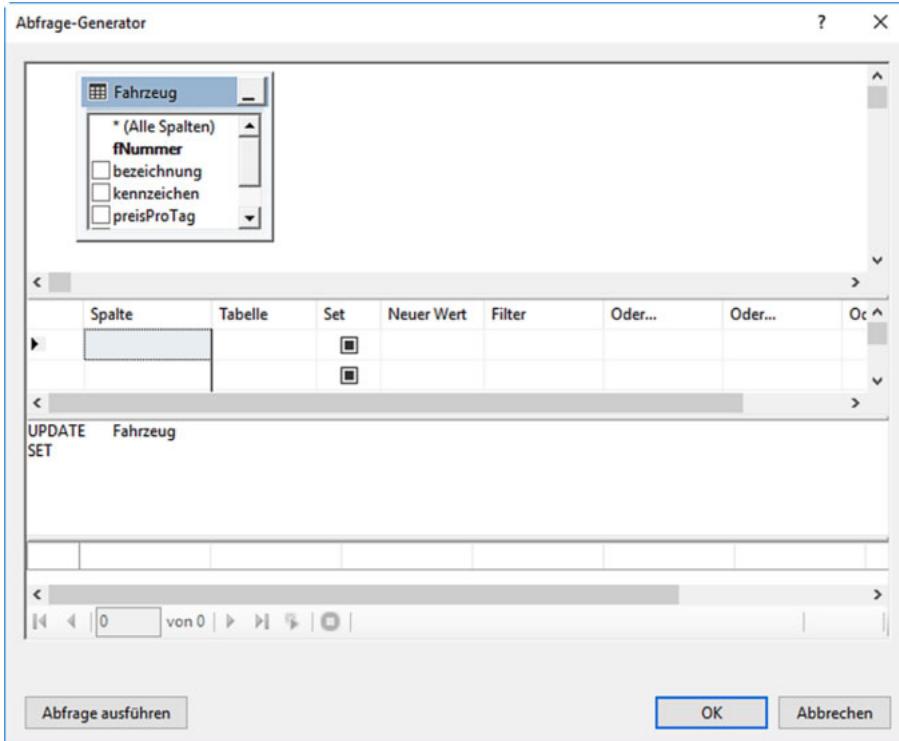


Abb. 2.20: Die übernommene Tabelle

Im nächsten Schritt markieren Sie in der Liste mit den Spalten der Tabelle den Eintrag **ausgeliehen** durch einen Mausklick in das Kästchen vor dem Feld. Der Generator übernimmt den Eintrag dann in den mittleren Bereich des Fensters.

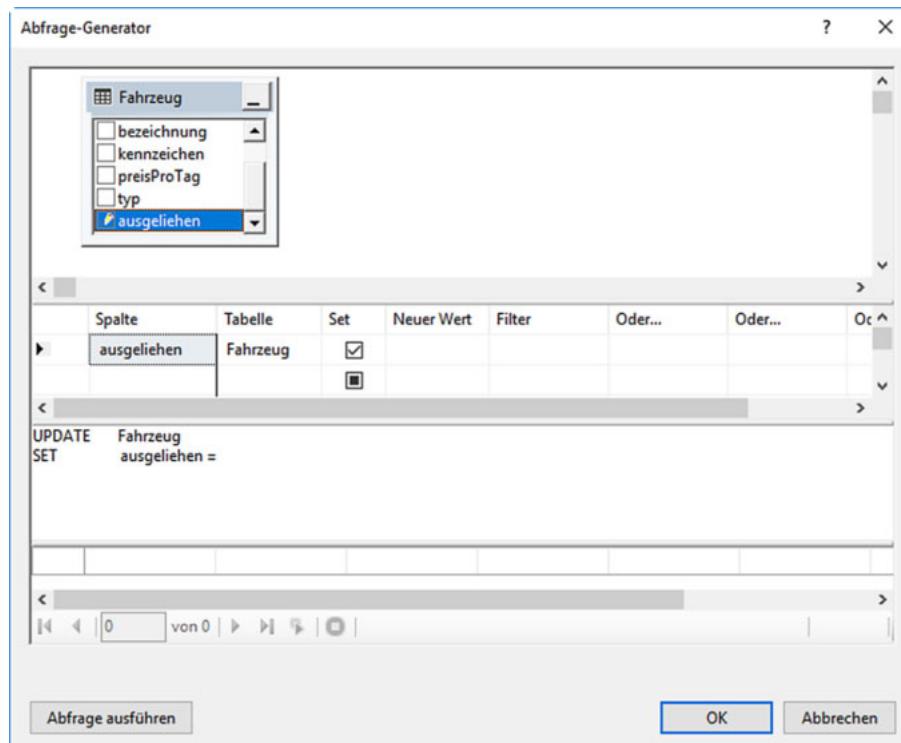


Abb. 2.21: Die übernommene Spalte

In der Spalte **Neuer Wert** im mittleren Bereich des Fensters legen Sie nun fest, welchen Wert die Spalte **ausgeliehen** in der Tabelle durch die Änderung erhalten soll. Wir setzen hier den Wert auf 1 für `true`.



Bitte beachten Sie:

Der Datentyp `bit` unterscheidet zwischen 0 für `false` und 1 für `true`.

Im letzten Schritt müssen Sie jetzt noch sicherstellen, dass auch der richtige Datensatz geändert wird. Dazu ergänzen Sie die folgende `WHERE`-Klausel am Ende der SQL-Anweisung:

```
WHERE (fNummer = @Original_fNummer)
```

Damit wird nur der Datensatz ausgewählt, bei dem der Wert der Spalte `fNummer` mit einem Wert übereinstimmt, den wir als Parameter übergeben.

Die vollständige SQL-Anweisung sollte dann so aussehen wie in der Abb. 2.22 im mittleren Bereich:

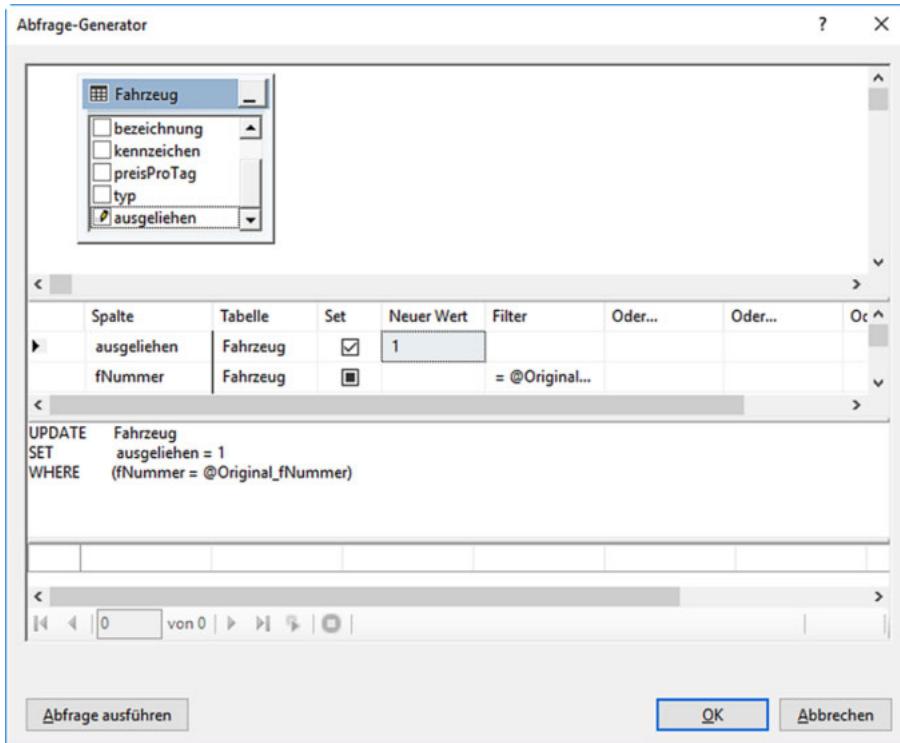


Abb. 2.22: Die vollständige SQL-Anweisung (im Feld in der Mitte)

Bitte beachten Sie:

Die Anzeige in der Tabelle in der Mitte erfolgt erst dann vollständig, wenn Sie nach der Eingabe der WHERE-Klausel einmal mit der Maus in die Tabelle klicken. Danach werden auch die Zeilen im mittleren Bereich neu umgebrochen.



Damit ist die Abfrage vollständig und Sie können den Generator mit einem Klick auf die Schaltfläche OK verlassen.

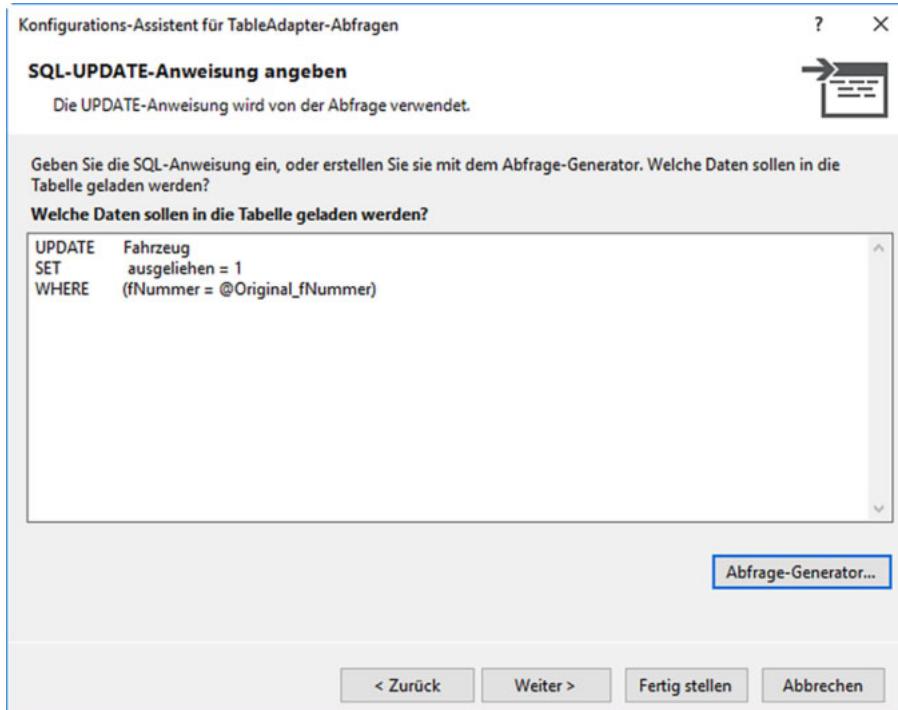


Abb. 2.23: Die SQL-Anweisung im Assistenten für TableAdapter-Abfragen

Im Assistenten für TableAdapter-Abfragen klicken Sie dann wieder auf **Weiter >**. Im vierten Schritt geben Sie einen Namen für die Abfrage ein. Wir verwenden in unserem Beispiel den Namen **AusgeliehenSetzen**.

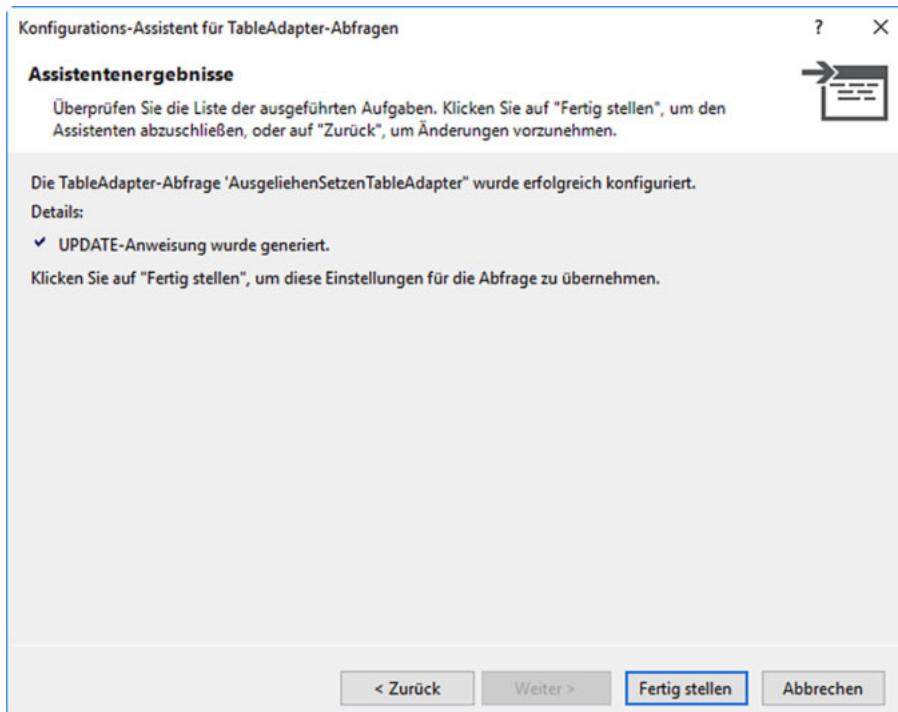


Abb. 2.24: Der letzte Schritt

Im letzten Schritt teilt Ihnen der Assistent dann mit, dass die Abfrage erfolgreich konfiguriert wurde. Klicken Sie in diesem Fenster bitte auf **Fertig stellen**.

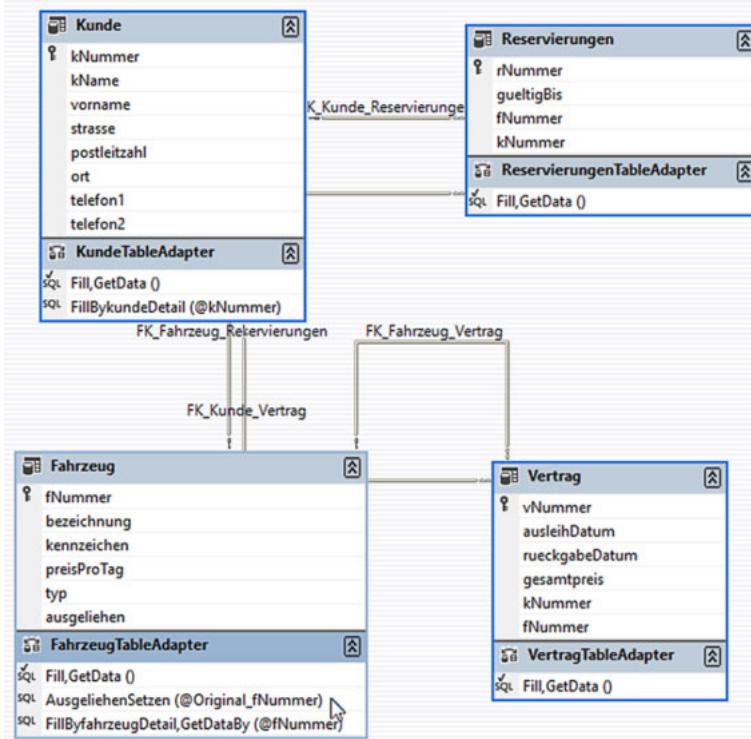


Abb. 2.25: Die neue UPDATE-Abfrage (unten im Bereich **FahrzeugTableAdapter**)

Beim Anlegen eines neuen Vertrags lassen wir jetzt auch die gerade erstellte Abfrage ausführen. Die entsprechende Anweisung ist im folgenden Code fett markiert.

```
private void ButtonSpeichern_Click(object sender, EventArgs e)
{
    this.Validate();
    this.vertragBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(this.auto2030DataSet);
    //die Update-Abfrage ausführen
    //bitte in einer Zeile eingeben
    this.fahrzeugTableAdapter.AusgeliehenSetzen
        (Convert.ToInt32(fNummerComboBox.SelectedValue));
    Close();
}
```

Code 2.5: Das Ausführen der UPDATE-Abfrage

Übernehmen Sie die Anweisung in Ihr Projekt und testen Sie die Erweiterung. Ob der Wert korrekt gesetzt wird, können Sie ganz einfach prüfen, indem Sie ein Fahrzeug „vermieten“ und dann über die Detailanzeige nachsehen.

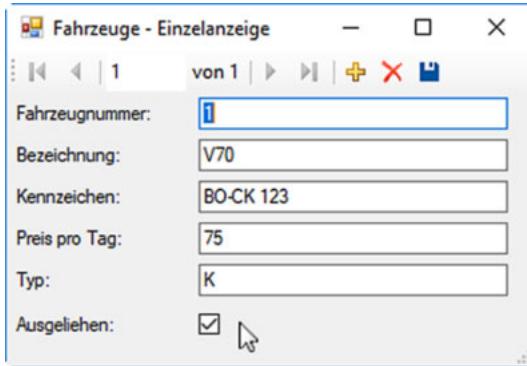


Abb. 2.26: Das gesetzte Kennzeichen **ausgeliehen**

Kümmern wir uns jetzt um die automatische Berechnung des Gesamtpreises. Hier müssen wir ein wenig mehr Aufwand betreiben. Wir berechnen aus der Differenz zwischen dem Ausleih- und dem Rückgabedatum die Mietdauer und multiplizieren sie mit dem Preis pro Tag, der in der Tabelle für die Fahrzeuge bei dem ausgewählten Fahrzeug hinterlegt ist. Da wir in jedem Fall einen Tag Mietdauer berechnen, müssen wir auf die Differenz zwischen dem Ausleih- und dem Rückgabedatum 1 addieren. Andernfalls wäre der Betrag ja 0, wenn der Kunde das Auto morgens abholt und abends wieder zurückgibt.

Den Preis pro Tag beschaffen wir uns über eine Abfrage, die uns zum ausgewählten Fahrzeug den entsprechenden Wert aus der Datenbanktabelle liefert. Anders als bisher lassen wir die Abfrage aber eine DataTable liefern.

Legen Sie bitte eine neue Abfrage für die Tabelle Fahrzeuge an, die Ihnen den Preis pro Tag für ein bestimmtes Fahrzeug über eine SELECT-Anweisung liefert. Die Auswahl soll dabei wieder über die Fahrzeugnummer erfolgen, die wir aus dem Kombinationsfeld beschaffen und an die Methode übergeben.

Die entsprechende Anweisung könnte so aussehen:

```
SELECT fNummer, bezeichnung, kennzeichen, preisProTag, typ,
ausgeliehen FROM dbo.Fahrzeug WHERE fNummer = @fNummer
```

Hinweis:

Bei der Auswahl des Abfragetyps im Konfigurations-Assistenten gibt es eine Option **SELECT-Anweisung, die einen einzelnen Wert zurückgibt**. Diese Option liefert Ihnen keinen einzelnen Wert aus einer Spalte einer Datenbanktabelle, sondern führt sogenannte Aggregatfunktionen aus. Mit diesen Aggregatfunktionen können Sie zum Beispiel die Daten in einer Spalte summieren oder die Anzahl von Datensätzen zählen.

Bei der Auswahl der Methoden markieren Sie dieses Mal bitte nur den Eintrag **DataTable zurückgeben**. Vergeben Sie außerdem einen sprechenden Namen an die Methode. Wir benutzen in unserem Beispiel **GetPreisProTag**.

Erstellen Sie dann eine Methode, die Ihnen die Daten aus der Abfrage beschafft, die Differenz zwischen den Tagen berechnet und den Gesamtpreis berechnet. Diese Methode könnte so aussehen:

```

public void PreisBerechnen()
{
    //zum Speichern des Preises pro Tag
    float preisProTag;
    //für die Differenz der Tage
    int anzahlTage;

    //den Preis aus der Tabelle beschaffen
    //bitte jeweils in einer Zeile eingeben
    auto2030DataSet.FahrzeugDataTable tabelle =
        this.fahrzeugTableAdapter.GetPreisProTag
        (Convert.ToInt32(fNummerComboBox.SelectedValue));
    //und den Wert aus der ersten Zeile und der Spalte
    //"preisProTag"
    preisProTag = Convert.ToSingle
    (tabelle.Rows[0]["preisProTag"]);
    //die Differenz der Tage berechnen
    anzahlTage = (rueckgabeDatumDateTimePicker.Value -
    ausleihDatumDateTimePicker.Value).Days;
    //und noch 1 addieren
    anzahlTage = anzahlTage + 1;
    //den berechneten Wert in das Feld setzen
    //bitte in einer Zeile eingeben
    gesamtpreisTextBox.Text = (anzahlTage *
    preisProTag).ToString();
}

```

Code 2.6: Die Methode PreisBerechnen()

Schauen wir uns die Anweisungen der Reihe nach an.

Mit der Anweisung

```

auto2030DataSet.FahrzeugDataTable tabelle =
this.fahrzeugTableAdapter.GetPreisProTag(Convert.ToInt32
(fNummerComboBox.SelectedValue));

```

erstellen wir eine DataTable vom Typ FahrzeugDataTable. Diese Tabelle kann also Daten aus der Tabelle mit den Fahrzeugdaten speichern. Erzeugt wird die Tabelle über unsere Methode GetPreisProTag(). Sie beschafft uns die Zeile mit dem Fahrzeug, das zu der ausgewählten Fahrzeugnummer gehört.

Die Anweisung

```
preisProTag = Convert.ToSingle(tabelle.Rows[0]["preisProTag"]);
```

liefert uns den Wert aus der Spalte **preisProTag** aus der ersten Zeile der Tabelle. Der Zugriff auf die Zeile erfolgt dabei über einen Index, der Zugriff auf die Spalte durch den Namen der Spalte. Den Wert bauen wir in den Typ **Single** beziehungsweise **float** um.

Die Anweisung

```
anzahlTage = (rueckgabeDatumDateTimePicker.Value -
ausleihDatumDateTimePicker.Value).Days;
```

berechnet die Anzahl der Tage, die sich aus der Differenz der beiden Datumsfelder ergibt. Dazu verwenden wir die Eigenschaft `Days`.

Anschließend addieren wir noch 1 auf den Wert, berechnen den Preis und setzen ihn in das Eingabefeld. Dabei gibt es keine Besonderheiten.

Übernehmen Sie den Code jetzt bitte und rufen Sie die Methode bei Änderungen im Feld für das Fahrzeug sowie bei Änderungen in den Datumsfeldern auf.

Hinweis:

Wir rechnen der Einfachheit halber immer komplette Tage ab. Wenn ein Kunde also ein Auto an einem Tag abholt und am nächsten Tag zurückgibt, muss er immer zwei Tage bezahlen – auch dann, wenn die Mietdauer geringer ist als 24 Stunden.

Abschließend müssen wir nun noch dafür sorgen, dass beim Vermieten im Kombinationsfeld für das Fahrzeug nur Einträge angezeigt werden, die nicht vermietet sind. Denn solche Fahrzeuge können ja nicht ein weiteres Mal vermietet werden.

Diese Einschränkungen setzen wir über einen Filter für `fahrzeugBindingSource`. Hier legen wir fest, dass nur Einträge angezeigt werden sollen, bei denen **ausgeliehen** gleich 0 ist.

Markieren Sie bitte das Steuerelement `fahrzeugBindingSource` unten im Formular für die Verträge. Geben Sie dann bei der Eigenschaft **Filter** in der Gruppe **Daten** im Eigenschaftenfenster den Ausdruck `ausgeliehen = 0` ein.

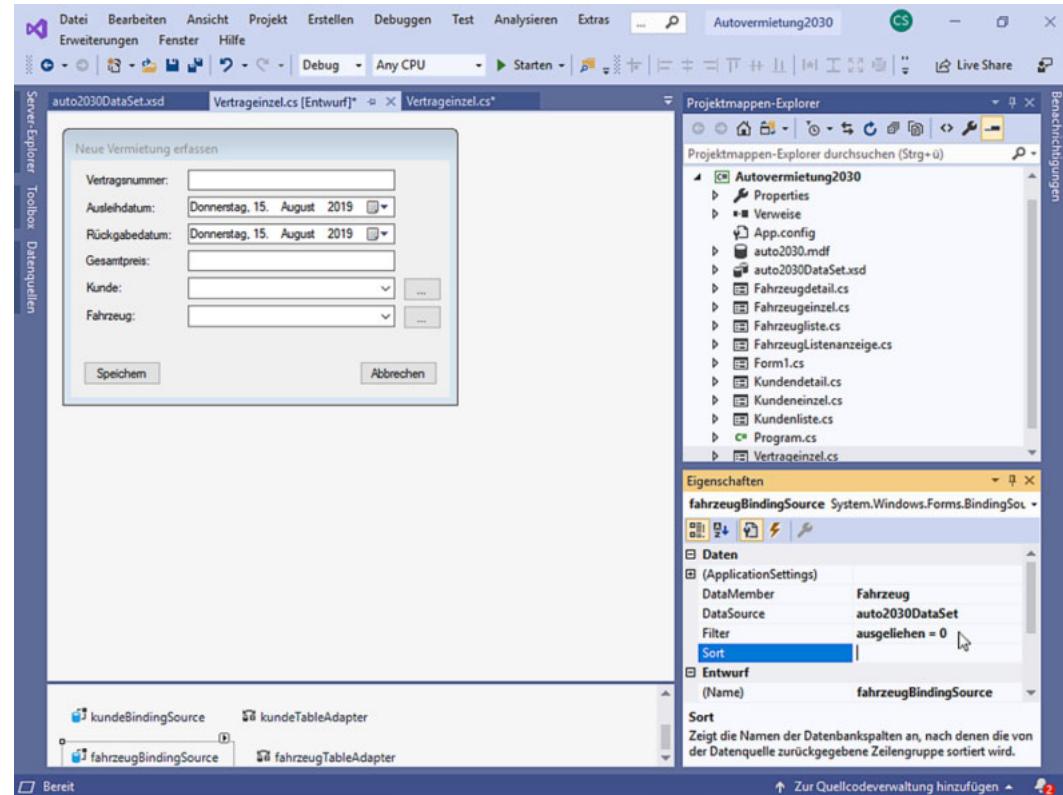


Abb. 2.27: Der gesetzte Filter (unten rechts im Eigenschaftenfenster)

Probieren Sie anschließend die Erweiterungen aus.

Bitte beachten Sie:

Die Liste für das Kombinationsfeld zur Auswahl des Fahrzeugs kann jetzt auch leer sein – nämlich dann, wenn es keine Fahrzeuge mehr zum Vermieten gibt. Dann führt das Setzen auf den ersten Eintrag beim Laden des Formulars zu einer Ausnahme. Entsprechendes gilt für die Kunden. Auch hier kann es ja sein, dass noch gar kein Eintrag erfasst wurde. Um diese Probleme abzufangen, können Sie vor dem Setzen eine Abfrage ergänzen, ob es überhaupt Einträge in den Listen gibt. Da es dabei keine Besonderheiten gibt, stellen wir Ihnen die einzelnen Schritte nicht im Detail vor. Praktisch umgesetzt finden Sie die Abfrage im Projekt im Download-Bereich.



Wir überprüfen nicht, ob die Auswahl in den beiden Datumsfeldern plausibel ist. Ein Kunde kann also auch ein Auto von heute bis gestern mieten. Dann beträgt der berechnete Preis 0. Wenn Sie das stört, können Sie entsprechende Prüfungen ja selbst ergänzen.

2.4 Die Rückgabe

Kommen wir nun zur Rückgabe von Fahrzeugen.

Hier erstellen wir ein Kombinationsfeld mit einer Liste aller vermieteten Fahrzeuge und setzen beim Anklicken einer Schaltfläche das Feld **ausgeliehen** in der Tabelle **Fahrzeuge** wieder auf 0. Den Vertrag an sich lassen wir unverändert. Auf diese Weise kann jederzeit nachvollzogen werden, wer wann welche Fahrzeuge gemietet hat.

Legen Sie ein neues Formular an und fügen Sie die Spalte **fNummer** aus der Tabelle **Fahrzeug** über die Datenquellen hinzu. Lassen Sie das Steuerelement dabei als **ComboBox** anzeigen. Öffnen Sie anschließend die ComboBox-Aufgaben über das Symbol rechts oben an dem Steuerelement. Markieren Sie im Menü den Eintrag **An Daten gebundene Elemente verwenden** und setzen Sie die Datenquelle über das entsprechende Kombinationsfeld auf **fahrzeugBinding Source**. In den Kombinationsfeldern **Member anzeigen**, **Wertemember** und **Ausgewählter Wert** wählen Sie jeweils die Spalte **fNummer** aus.

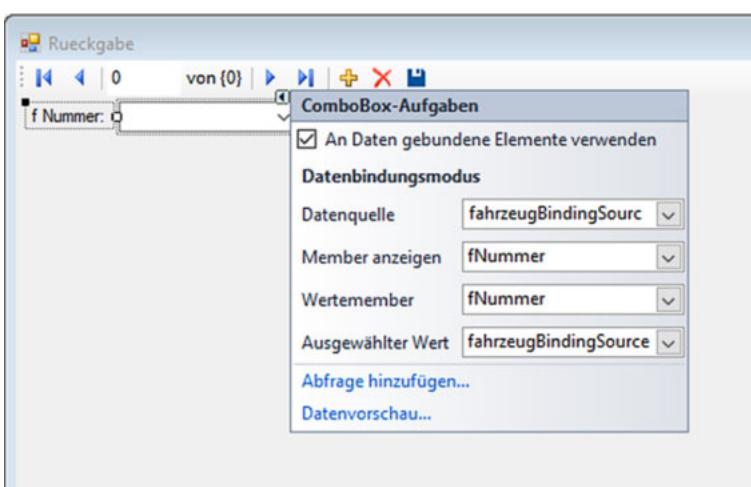


Abb. 2.28: Die Einstellungen für die ComboBox-Aufgaben

Setzen Sie dann für das Steuerelement **fahrzeugBindingSource** die Eigenschaft **Filter** auf den Ausdruck `ausgeliehen = 1`. Damit werden nur noch die Fahrzeuge angezeigt, die aktuell vermietet sind.

Fügen Sie anschließend zwei Schaltflächen **Übernehmen** und **Abbrechen** in das Formular ein. Beim Anklicken der Schaltfläche **Übernehmen** lassen wir eine UPDATE-Abfrage ausführen, die wir gleich noch erstellen werden. Beim Anklicken der Schaltfläche **Abbrechen** dagegen schließen wir das Formular.

Löschen Sie anschließend die Navigationsleiste und gestalten Sie das Formular noch ein wenig ansprechender. Es könnte zum Beispiel so aussehen:

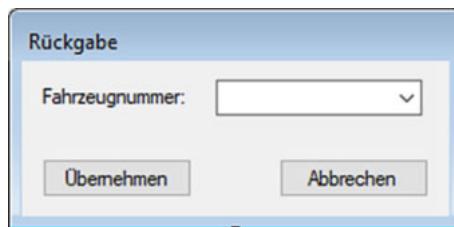


Abb. 2.29: Das Formular für die Rückgabe

Erstellen Sie dann eine UPDATE-Abfrage für die Tabelle **Fahrzeug**, die den Wert in der Spalte **ausgeliehen** für das aktuell ausgewählte Fahrzeug auf 0 setzt.

Bevor Sie weiterlesen ...

Versuchen Sie erst einmal selbst, die Abfrage zu erstellen.

Da sich das Vorgehen nicht wesentlich vom Erstellen der UPDATE-Abfrage unterscheidet, mit der wir den Wert der Spalte **ausgeliehen** auf 1 gesetzt haben, stellen wir Ihnen hier nur noch einmal die wichtigsten Schritte im Überblick vor.

Erstellen Sie zuerst im DataSet-Designer über die Funktion **Hinzufügen/Abfrage...** beziehungsweise **Abfrage hinzufügen...** im Kontextmenü des Bereichs **FahrzeugTableAdapter** eine neue Abfrage. Markieren Sie im zweiten Schritt die Option **UPDATE**. Formulieren Sie im dritten Schritt die Abfrage. Dazu ersetzen Sie den Eintrag, den der Assistent vorschlägt, durch den Ausdruck

```
UPDATE Fahrzeug
SET ausgeliehen = 0
WHERE (fNummer = @Original_fNummer)
```

Tipp:

*Den Ausdruck können Sie aus der bereits vorhandenen Abfrage **AusgeliehenSetzen** übernehmen. Sie müssen nur den Wert 1 bei der SET-Anweisung in 0 ändern.*

Vergeben Sie im nächsten Schritt einen Namen für die Abfrage und lassen Sie die Abfrage fertigstellen. Wir benutzen in unserem Beispiel den Namen **RueckgabeSetzen**.

Beim Anklicken der Schaltfläche **Übernehmen** im Formular für die Rückgabe lassen Sie dann die Abfrage ausführen und schließen das Formular. Die Methode könnte zum Beispiel so aussehen:

```
private void ButtonUebernehmen_Click(object sender,
EventArgs e)
{
    //die UPDATE-Abfrage ausführen
    //bitte in einer Zeile eingeben
    this.fahrzeugTableAdapter.RueckgabeSetzen
    (Convert.ToInt32(fNummerComboBox.SelectedValue));
    Close();
}
```

Code 2.7: Das Ausführen der UPDATE-Abfrage für die Rückgabe

Zusätzlich bieten wir dem Anwender jetzt noch weitere Details zu einem Fahrzeug über eine Schaltfläche an. Da sich das Vorgehen nicht vom Anzeigen der Fahrzeugdetails aus dem Formular für einen neuen Vertrag unterscheidet, stellen wir es hier nicht noch einmal vor.

Damit das Formular nur dann angezeigt wird, wenn es auch vermietete Fahrzeuge gibt, können Sie nach dem Laden noch eine Abfrage ergänzen, ob das Kombinationsfeld für die Fahrzeuge Einträge enthält. Wenn nicht, schließen Sie das Formular direkt wieder. Da es dabei ebenfalls keine Besonderheiten gibt, stellen wir Ihnen die Anweisungen hier nicht weiter vor. Praktisch umgesetzt finden Sie sie im Download-Bereich.

Speichern Sie jetzt sämtliche Änderungen und testen Sie das Formular für die Rückgabe. Fügen Sie dazu im Formular **Form1** eine Schaltfläche zum Anzeigen des Formulars ein und lassen Sie das neue Formular beim Anklicken dieser Schaltfläche wie gewohnt modal anzeigen.

Damit wäre auch die Rückgabe von Fahrzeugen möglich und wir können uns um die Reservierungen kümmern.

3 Die Reservierungen

In diesem Kapitel setzen wir die Reservierungen um.

Dabei gibt es eigentlich keine Besonderheiten. Wir können hier im Wesentlichen genauso vorgehen wie bei den Verträgen.

Zuerst erstellen wir ein Formular zum Anlegen einer Reservierung. In diesem Formular greifen wir wie beim Vermieten eines Fahrzeugs über Kombinationsfelder auf Daten in den Tabellen **Kunde** und **Fahrzeug** zurück. Beim Fahrzeug sorgen wir über einen Filter dafür, dass nur solche Fahrzeuge angezeigt werden, die aktuell vermietet sind. Für alle anderen Fahrzeuge muss ja keine Reservierung erzeugt werden.

Die Nummer für die Reservierung wird automatisch vom Datenbank-Management-System erzeugt. Das Datum setzen wir auf das aktuelle Tagesdatum + 30 Tage.

Genau wie beim Vermieten und auch bei der Rückgabe bieten wir dem Anwender zusätzlich die Möglichkeit, über Schaltflächen Details zu einem Fahrzeug und einem Kunden abzurufen.

Hinweis:

Filter haben wir ja auch im letzten Kapitel zum Beispiel bei der Rückgabe erstellt.

Das Abrufen der Details für ein Fahrzeug und einen Kunden haben wir ebenfalls im letzten Kapitel programmiert.

Da es ansonsten keinerlei Besonderheiten gibt, stellen wir Ihnen die einzelnen Schritte nur noch einmal in kompakter Form vor. Bei Bedarf lesen Sie bitte noch einmal im letzten Kapitel nach.

Legen Sie ein neues Formular für die Reservierungen an. Wir benutzen in unserem Beispiel den Namen **Reservierung**. Wechseln Sie in das Register mit den Datenquellen und öffnen Sie den Zweig für die Tabelle **Reservierungen** im Zweig **Kunde**.



Denken Sie bitte daran:

Das Erstellen der Auswahllisten funktioniert nur dann korrekt, wenn Sie die Daten über die untergeordnete Tabelle im Zweig der Tabelle **Kunde** verarbeiten. Wenn Sie die Tabellen aus der obersten Ebene verwenden, können die Listen nicht erstellt werden.

Lassen Sie die Spalten **kNummer** und **fNummer** als ComboBoxen anzeigen. Markieren Sie anschließend den Eintrag der untergeordneten Tabelle **Reservierungen** im Zweig der Tabelle **Kunde** und wählen Sie über das Symbol die Ansicht **Details** aus. Ziehen Sie danach die untergeordnete Tabelle **Reservierungen** mit gedrückter linker Maustaste in das Formular.

Ziehen Sie dann den Eintrag der kompletten Tabelle **Kunde** auf das Steuerelement zur Anzeige der Kundennummer und wiederholen Sie diesen Schritt mit der Tabelle **Fahrzeug** für das Steuerelement zur Anzeige der Fahrzeugnummer.

Denken Sie daran:

Sie müssen jeweils den Eintrag der **kompletten Tabelle** aus der Ansicht der Datenquellen auf das Steuerelement ziehen. Überprüfen Sie bitte auch noch einmal, ob bei der Eigenschaft **(DataBindings)/SelectedValue** für die Kombinationsfelder die richtigen Werte angezeigt werden – nämlich jeweils die Spalten in **reservierungBindingSource**.



Legen Sie anschließend für das Steuerelement **fahrzeugBindingSource** einen Filter `ausgeliehen = 1` fest. Damit werden nur die Fahrzeuge angezeigt, die gerade vermietet sind.

Ändern Sie dann die Texte vor den Steuerelementen für die Dateneingabe, löschen Sie die Navigationsleiste und passen Sie gegebenenfalls auch noch den Text in der Titelleiste des Formulars an. Verschieben Sie – falls erforderlich – außerdem die Steuerelemente so im Formular, dass alles korrekt dargestellt wird. Lassen Sie dabei aber bitte wieder unten und rechts ein wenig Platz für weitere Steuerelemente.

Ergänzen Sie hinter den Feldern für das Fahrzeug und den Kunden zwei Schaltflächen mit drei Punkten, über die Sie die Detailanzeige starten. Unten links in das Formular setzen Sie bitte eine Schaltfläche mit dem Text **Übernehmen** und unten rechts eine Schaltfläche mit dem Text **Abbrechen**.

Ändern Sie dann noch die Eigenschaft **ReadOnly** für das Feld mit der Reservierungsnummer auf `True`. Da die Reservierungsnummer genau wie die Kunden- oder die Fahrzeugnummer automatisch vergeben wird, müssen in diesem Feld ja keine Eingaben durch den Anwender erfolgen.

Nach diesen Änderungen könnte das Formular zum Beispiel so aussehen:

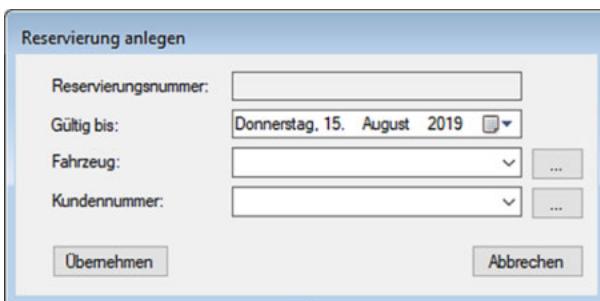


Abb. 3.1: Das Formular für die Reservierungen

Übernehmen beziehungsweise erweitern Sie im nächsten Schritt die Methoden aus dem folgenden Code. Sie sorgen dafür, dass beim Öffnen ein neuer Datensatz angelegt wird, setzen das Datum auf einen Vorschlagswert und übernehmen das Speichern der Daten. Diese Techniken haben Sie bereits in sehr ähnlicher Form auch bei den Verträgen eingesetzt.

```
private void Reservierung_Load(object sender, EventArgs e)
{
    //Diese Codezeile lädt Daten in die Tabelle
    //"auto2030DataSet.Fahrzeug". Sie können sie bei
    //Bedarf verschieben oder entfernen.
```

```
//bitte in einer Zeile eingeben
this.fahrzeugTableAdapter.Fill(this.auto2030DataSet.
Fahrzeug);
//Diese Codezeile lädt Daten in die Tabelle
//"auto2030DataSet.Kunde". Sie können sie bei
//Bedarf verschieben oder entfernen.
this.kundeTableAdapter.Fill(this.auto2030DataSet.Kunde);
//Diese Codezeile lädt Daten in die Tabelle
//"auto2030DataSet.Reservierung". Sie können sie
//bei Bedarf verschieben oder entfernen.
this.reservierungTableAdapter.Fill
(this.auto2030DataSet.Reservierung);
//gibt es überhaupt Fahrzeuge und Kunden, für die eine
//Reservierung vorgenommen werden kann?
//bitte in einer Zeile eingeben
if (fNummerComboBox.Items.Count == 0 ||
kNummerComboBox.Items.Count == 0)
{
    //bitte in einer Zeile eingeben
    MessageBox.Show("Es gibt keine Fahrzeuge oder Kunden für
    eine Reservierung !", "Hinweis", MessageBoxButtons.OK,
    MessageBoxIcon.Exclamation);
    Close();
}
else
{
    //einen neuen Datensatz erzeugen
    reservierungenBindingSource.AddNew();
    //den Wert für das Fahrzeug setzen
    fNummerComboBox.SelectedIndex = 0;
    //den Wert für den Kunden setzen
    kNummerComboBox.SelectedIndex = 0;
    //den Wert für das Reservierungsdatum setzen
    //hier 30 Tage auf das aktuelle Datum addieren
    //bitte in einer Zeile eingeben
    gueltigBisDateTimePicker.Value = DateTime.Now.
    AddDays(30);
}
}

private void ButtonUebernehmen_Click(object sender,
EventArgs e)
{
    this.Validate();
    this.reservierungenBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(this.auto2030DataSet);
    Close();
}

private void ButtonAbbrechen_Click(object sender,
EventArgs e)
{
    Close();
}

private void ButtonDetailFahrzeug_Click(object sender,
```

```
EventArgs e)
{
    Fahrzeugdetail detailAnzeigeFahrzeug = new Fahrzeugdetail();
    //bitte in einer Zeile eingeben
    detailAnzeigeFahrzeug.DetailsLaden(Convert.ToInt32
        (fNummerComboBox.SelectedValue));
    detailAnzeigeFahrzeug.ShowDialog();
}

private void ButtonDetailKunde_Click(object sender,
EventArgs e)
{
    Kundendetail detailAnzeigeKunde = new Kundendetail();
    //bitte in einer Zeile eingeben
    detailAnzeigeKunde.DetailsLaden(Convert.ToInt32
        (kNummerComboBox.SelectedValue));
    detailAnzeigeKunden.ShowDialog();
}
```

Code 3.1: Die Methoden für die Verarbeitung einer Reservierung

Speichern Sie dann alle Änderungen und ergänzen Sie im Formular `Form1` eine Schaltfläche, um das Formular für die Reservierungen modal zu öffnen. Führen Sie anschließend einen Test durch.

Hinweis:

Die Methoden zum Anzeigen der Kunden- und Fahrzeugdetails haben wir immer wieder in identischer Form benutzt. Wenn Sie das stört, können Sie sie ja auch in eine eigene Klasse auslagern.

Damit wollen wir die Arbeit an dem Formular für die Reservierungen beenden. Im nächsten Kapitel werden wir noch sicherstellen, dass die Integrität der Daten beim Löschen von Kunden und Fahrzeugen erhalten bleibt.

Zusammenfassung

Sie können in einem Formular für eine Datenbanktabelle Daten aus einer anderen Datenbanktabelle anzeigen lassen. Dazu erstellen Sie zunächst wie gewohnt die Steuerelemente für die erste Tabelle. Ziehen Sie dann die vollständige andere Tabelle aus der Ansicht der Datenquellen auf das gewünschte Steuerelement der ersten Tabelle.

Wenn sich die beiden Tabellen nicht im selben Zweig der Datenquellen befinden, müssen Sie die Eigenschaft **(DataBindings)/SelectedValue** unter Umständen selbst auf die gewünschte Spalte der Datenbanktabelle setzen.

Mit dem Steuerelement **DateTimePicker** können Sie ein Eingabefeld für ein Datum und eine Uhrzeit erstellen.

Mit dem Konfigurations-Assistenten für TableAdapter-Abfragen können Sie sehr einfach Methoden erstellen, die Daten über eine SQL-Abfrage aus einer Datenbanktabelle selektieren. Sie starten den Assistenten im Kontextmenü einer Tabelle im Tabellen-Designer.

Über die Methoden, die der Assistent erstellt, können Sie sich in einem Formular Daten aus einer Tabelle beschaffen.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie wollen in einem Formular für eine untergeordnete Datenbanktabelle über ein Feld nur Werte aus einer Spalte einer übergeordneten Datenbanktabelle anzeigen lassen. Die beiden Tabellen sind bereits über eine Beziehung miteinander verbunden. Beschreiben Sie kurz, wie Sie diese Aufgabe lösen können.

- 3.2 Sie wollen für eine Datenbanktabelle direkt im Quelltext einen neuen Datensatz anlegen. Welche Methode für welches Steuerelement benutzen Sie?

- 3.3 Sie haben über den Assistenten eine TableAdapter-Abfrage für eine Tabelle **Bestellungen** erzeugt und die Methode unter dem Namen **FillByDatum** gespeichert. Die Abfrage liefert nur die Datensätze, bei denen das Datum mit einem Wert übereinstimmt, der über eine Variable übergeben wird. Wie lautet die Anweisung, um diese Methode im Quelltext aufzurufen? Für das DataSet können Sie dabei einen beliebigen Namen verwenden.

4 Die Datenintegrität

In diesem Kapitel sorgen wir dafür, dass die Daten in unserer Anwendung auch beim Löschen von Kunden und Fahrzeugen weiterhin gültig bleiben.

Noch einmal zur Auffrischung

Wir haben die Tabellen mit den Verträgen und Reservierungen über Fremdschlüssel mit den Tabellen für die Kunden und die Fahrzeuge verbunden. Wenn Sie jetzt einen Kunden oder ein Fahrzeug löschen, zu dem bereits ein Vertrag oder eine Reservierung angelegt wurde, scheitert auch der Zugriff auf die entsprechenden Daten in den anderen Tabellen.



Das können Sie ganz einfach selbst ausprobieren. Legen Sie einmal einen neuen Kunden zum Test an und erfassen Sie zu diesem Kunden eine Reservierung oder eine Vermietung. Löschen Sie danach den Kunden wieder und versuchen Sie, eine neue Reservierung oder einen neuen Vertrag anzulegen. Das Programm löst dann eine Ausnahme aus, da für den Wert in der Spalte **kNummer** keine Entsprechung mehr in der Tabelle **Kunde** gefunden wird.

Bitte beachten Sie:

Das Löschen eines Kunden, zu dem bereits eine Reservierung oder Vermietung erfasst ist, kann zu dauerhaften Problemen führen, wenn Sie die Datenbank beim Ausführen des Programms nicht in den Ordner mit der Anwendung kopieren lassen. Verzichten Sie dann bitte auf den Test.



Wenn Sie dagegen die Standardeinstellungen unverändert gelassen haben, wird beim Ausführen eine Kopie der Datenbank erstellt und der Kunde nur in dieser Kopie gelöscht.

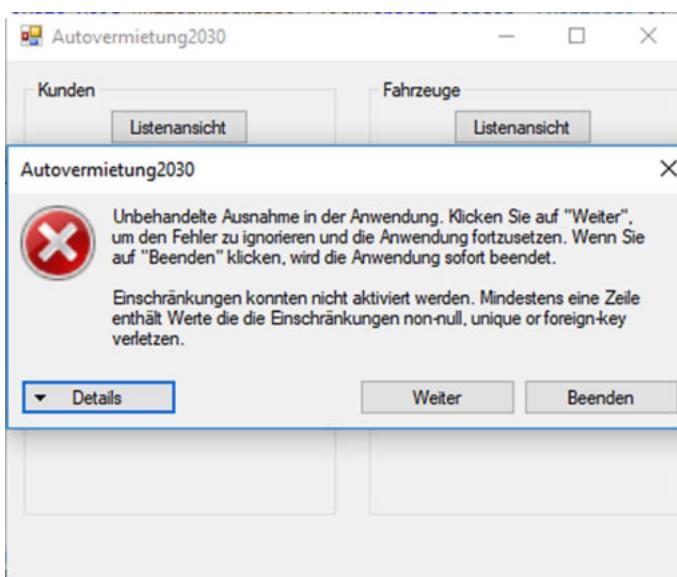


Abb. 4.1: Die Ausnahme bei einem gelöschten Kunden

Hinweis:

Die Ausnahme erscheint, obwohl wir gar nicht alle Datensätze anzeigen. Allerdings beschaffen wir ja beim Anzeigen des Formulars nach wie vor sämtliche Datensätze aus der Tabelle.

Sie könnten das Programm nun zwar nach einem Klick auf die Schaltfläche **Weiter** in der Meldung fortsetzen, besonders elegant ist diese Lösung aber nicht. Sehr viel besser wäre es natürlich, wenn solche Probleme erst gar nicht auftreten könnten.

Wenn Sie noch einmal an das Anlegen der Beziehungen zwischen den Tabellen zurückdenken, sollten Sie jetzt ein wenig stutzig werden. Denn dort haben wir ja eigentlich über die referentielle Integrität dafür sorgen wollen, dass genau diese Schwierigkeiten verhindert werden. Beim Löschen eines Fahrzeugs oder eines Kunden sollten auch die entsprechenden Datensätze in der Tabelle mit den Reservierungen oder den Verträgen gelöscht werden.

Grundsätzlich funktioniert das auch – allerdings nur dann, wenn sich die Tabellen gleichzeitig in einem Formular befinden. Das heißt nun aber nicht, dass Sie bei den Kunden auch immer alle Reservierungen und Verträge anzeigen lassen müssen. Wir nehmen die Tabellen mit den Reservierungen und den Verträgen einfach in die Formulare für die Kunden auf und löschen die Steuerelemente für die Anzeige dann wieder. Die nicht visuellen Steuerelemente für den Zugriff bleiben dabei weiter in den Formularen und ermöglichen einen Zugriff auf die Daten.

Schauen wir uns das am Beispiel der Listenanzeige für die Kunden an.

Öffnen Sie bitte das Formular **Kundenliste**. Setzen Sie dann die Eigenschaft **Dock** des Steuerelements **kundeDataGridView** auf **None** und vergrößern Sie das Formular ein wenig nach unten. Diese Schritte sind erforderlich, damit wir die Tabellen mit den Reservierungen und den Verträgen getrennt im Formular ablegen können.

Hinweis:

Wenn Sie die Eigenschaft **Dock** der Tabelle für die Kunden nicht verändern, nimmt das Steuerelement immer den gesamten Platz im Formular ein. Sie können zwar die Tabellen mit den Verträgen und den Reservierungen auf dem Steuerelement **kundeDataGridView** ablegen, allerdings funktioniert dann die Verbindung nicht. Die Tabellen mit den Reservierungen und den Verträgen **müssen** in einem eigenen Bereich im Formular abgelegt werden.

Lassen Sie anschließend die Datenquellen anzeigen und fügen Sie die untergeordneten Tabellen **Reservierungen** und **Vertrag** aus dem Zweig der Tabelle **Kunde** als DataGridView unten im freien Bereich des Formulars ein. Achten Sie auch hier bitte darauf, dass Sie die untergeordneten Tabellen verwenden und nicht die Tabellen, die sich in der obersten Ebene der Baumstruktur befinden.

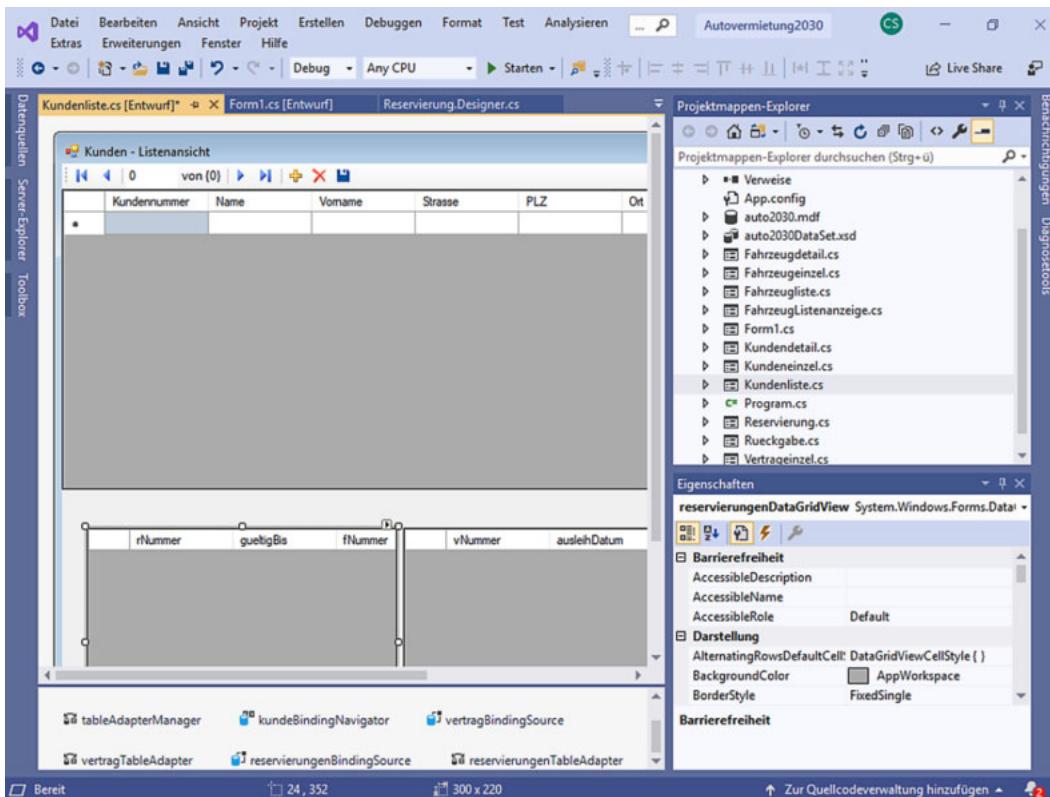


Abb. 4.2: Die eingefügten Steuerelemente für die untergeordneten Tabellen (unten in der Mitte im Formular)

Bevor wir die Steuerelemente für die Anzeige der Reservierungen und Verträge wieder löschen, testen wir erst einmal, ob überhaupt die Daten der Kunden mitsamt ihren Reservierungen und Verträgen angezeigt werden. Speichern Sie die Änderungen und starten Sie das Programm. Legen Sie einen neuen Vertrag an und rufen Sie dann die Listenansicht für die Kunden auf.

Wahrscheinlich wird dabei nun ebenfalls eine Ausnahme ausgelöst. Denn das Programm sucht auch nach den Daten für die Fahrzeuge – und kann sie aus dem aktuellen Formular nicht abrufen. Ergänzen Sie daher bitte auch noch die Tabelle mit den Fahrzeugen im Formular für die Kundenliste. Überprüfen Sie dann, ob die Daten für die Tabellen in der Methode `KundenListe_Load()` in der richtigen Reihenfolge geladen werden. Zuerst müssen die Daten für die übergeordneten Tabellen beschafft werden, dann die Daten für die untergeordneten Tabellen. Die Anweisungen in der Methode `KundenListe_Load()` sollten also so aussehen:

```
private void Kundenliste_Load(object sender, EventArgs e)
{
    //Diese Codezeile lädt Daten in die Tabelle
    // "auto2030DataSet.Fahrzeug". Sie können sie bei
    // Bedarf verschieben oder entfernen.
    this.fahrzeugTableAdapter.Fill(this.auto2030DataSet.Fahrzeug);
    //Diese Codezeile lädt Daten in die Tabelle
    // "auto2030DataSet.Kunde". Sie können sie bei
    // Bedarf verschieben oder entfernen.
    this.kundeTableAdapter.Fill(this.auto2030DataSet.Kunde);
```

```

//Diese Codezeile lädt Daten in die Tabelle
//"auto2030DataSet.Reservierungen". Sie können sie
//bei Bedarf verschieben oder entfernen.
//bitte jeweils in einer Zeile eingeben
this.reservierungTableAdapter.Fill(this.auto2030DataSet.
Reservierungen);
//Diese Codezeile lädt Daten in die Tabelle
//"auto2030DataSet.Vertrag". Sie können sie bei
//Bedarf verschieben oder entfernen.
this.vertragTableAdapter.Fill(this.auto2030DataSet.
Vertrag);
}

```

Code 4.1: Die Reihenfolge zum Beschaffen der Daten

Testen Sie das Programm dann noch einmal. Legen Sie einen neuen Vertrag an, und lassen Sie anschließend die Kundenliste anzeigen.

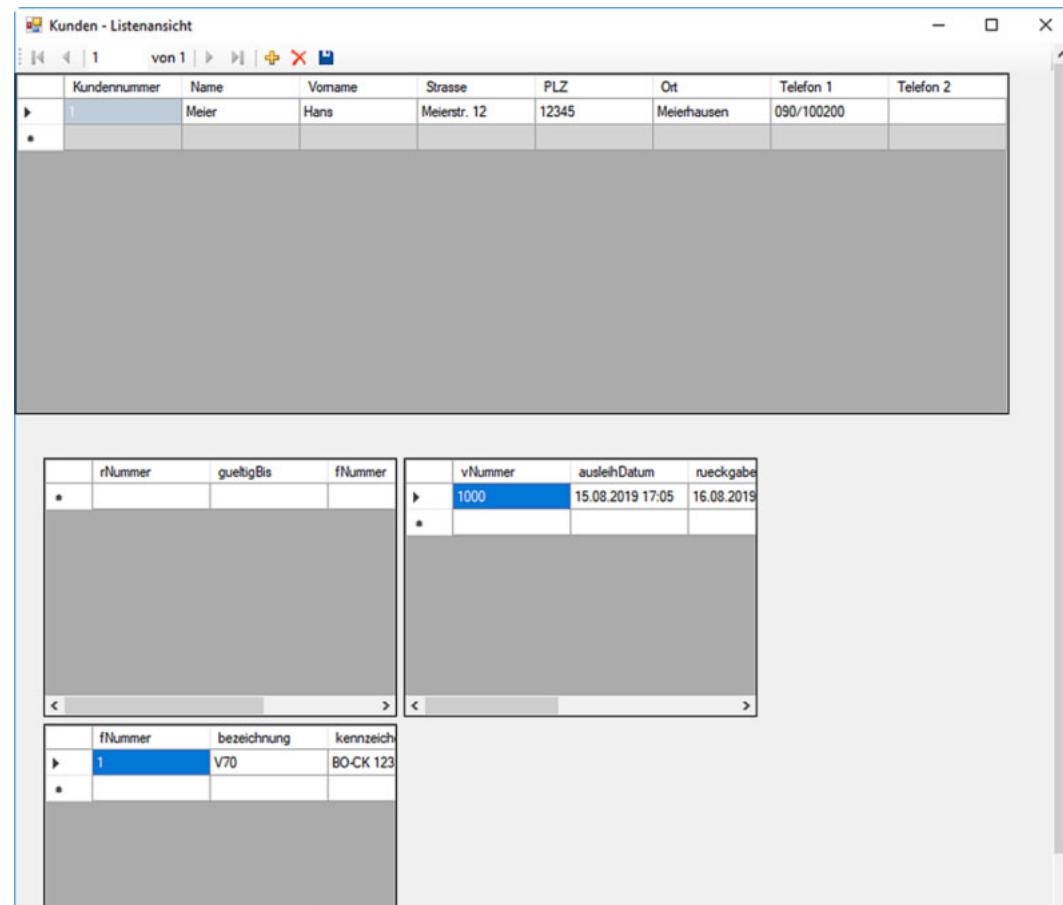


Abb. 4.3: Die Anzeige der untergeordneten Tabellen

Im unteren Bereich des Formulars sollten nun auch die drei anderen Tabellen angezeigt werden. Was genau in den Tabellen für die Verträge und Reservierungen erscheint, hängt vom aktuell markierten Kunden in der oberen Tabelle ab.

Sie könnten nun ohne Weiteres einen Kunden löschen. Dabei werden auch die dazugehörigen Daten in den Tabellen für die Verträge und Reservierungen unten im Formular entfernt. Beim Speichern werden die Änderungen in sämtliche Datenbanktabellen übernommen.

Das erledigt der TableAdapterManager.

Bitte beachten Sie:

Unter Umständen werden die Einstellungen für den TableAdapterManager nicht automatisch korrekt gesetzt. Wenn Änderungen in den untergeordneten Tabellen zwar korrekt angezeigt, aber nicht gespeichert werden, prüfen Sie die Einstellungen für den TableAdapterManager. Suchen Sie dazu bei den Steuerelementen im Bereich unterhalb des Formulars nach dem TableAdapterManager. Markieren Sie das Steuer-element und setzen Sie dann die Eigenschaften **FahrzeugTableAdapter**, **ReservierungenTableAdapter** und **VertragTableAdapter** auf die entsprechenden gleichnamigen Adapter.



Nun können wir das Formular für die Listenanzeige wieder in seinem ursprünglichen Zustand anzeigen lassen. Markieren Sie dazu die drei Tabellen unten im Formular und löschen Sie sie dann. Dabei werden nur die Steuerelemente für die Anzeige entfernt. Die nicht visuellen Steuerelemente werden nach wie vor unten im grauen Bereich unterhalb des Formulars im Editor angezeigt und stellen die Daten zur Verfügung.

Verkleinern Sie anschließend gegebenenfalls das Formular wieder ein wenig und setzen Sie die Eigenschaft **Dock** für das Steuerelement **kundeDataGridView** wieder auf **Fill**.

Sie können die TableAdapter für Datenbanktabellen auch über die Toolbox einfügen. Sie finden die entsprechenden Steuerelemente in der Regel ganz oben in der Toolbox.

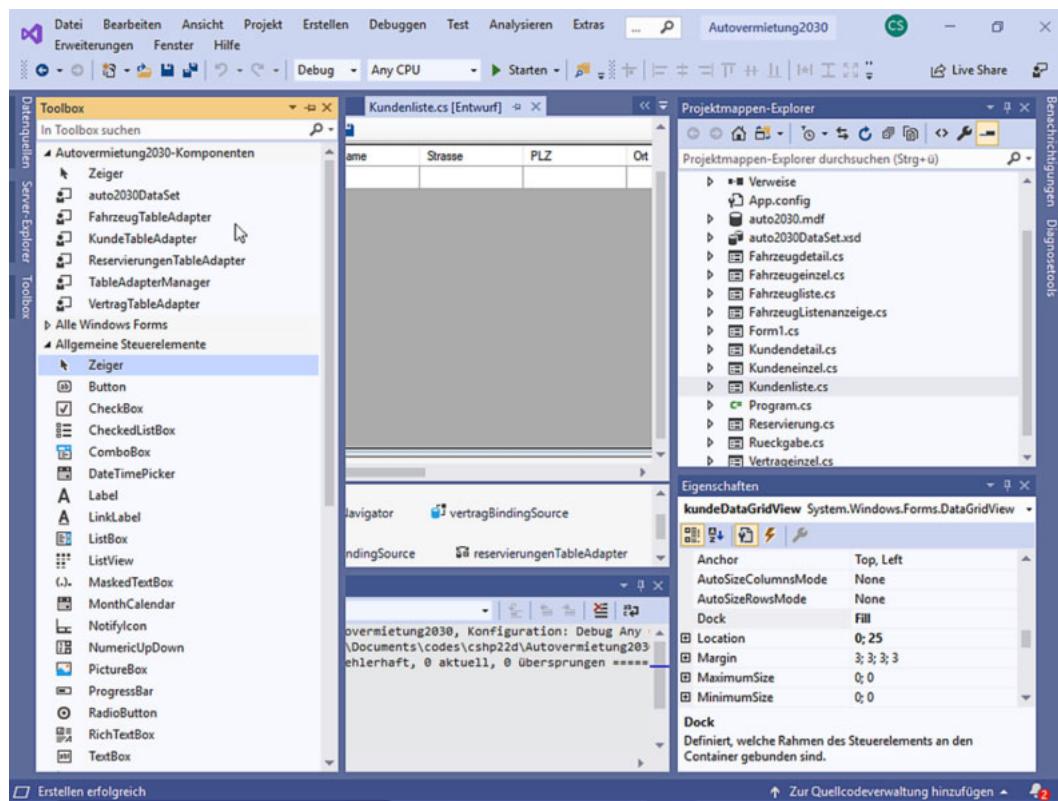


Abb. 4.4: Die Steuerelemente für die TableAdapter (links oben in der Toolbox)

Beim manuellen Einfügen müssen Sie aber auch selbst mit den entsprechenden Anweisungen dafür sorgen, dass die Daten über die Methode `Fill()` beschafft werden. Diese Anweisungen können Sie zum Beispiel beim Laden des Formulars ausführen lassen.

Über die Eigenschaft **EnforceConstraints**⁹ für ein DataSet können Sie die Prüfung der referenziellen Integrität auch abschalten. Dazu setzen Sie die Eigenschaft auf `False`. Sie finden sie in der Gruppe **Sonstiges** des Eigenschaftenfensters.

9. *Enforce Constraints* bedeutet übersetzt so viel wie „erzwinge Beschränkungen“.

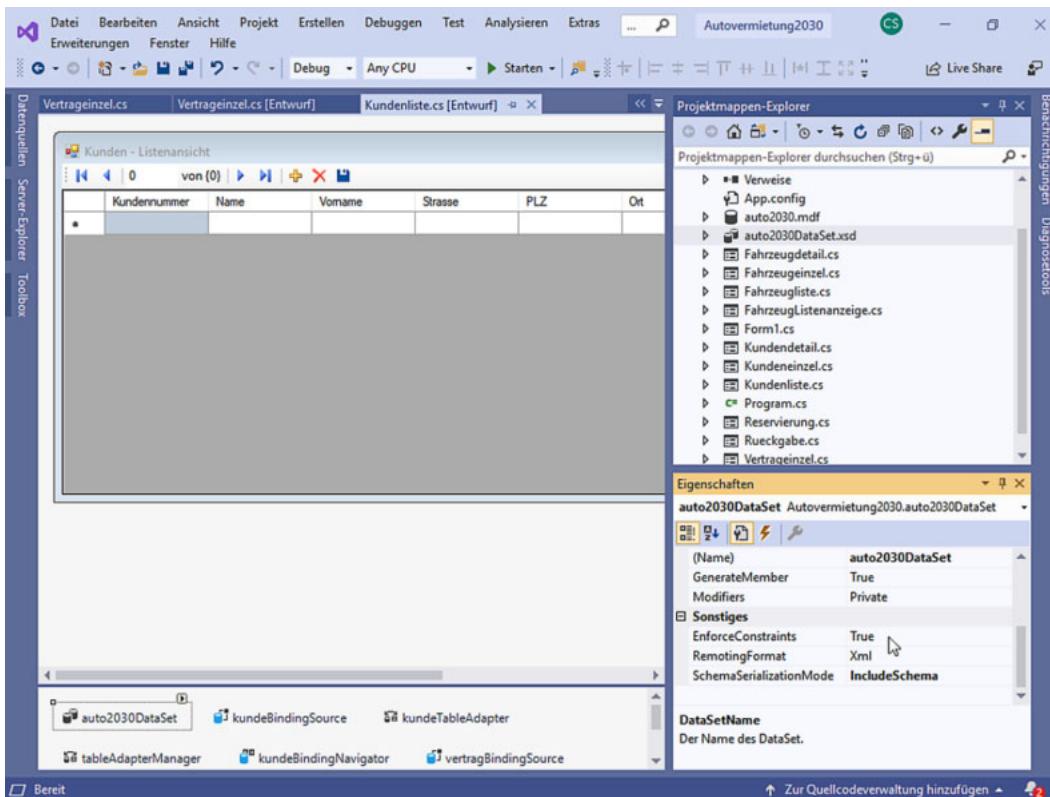


Abb. 4.5: Die Eigenschaft **EnforceConstraints** (unten rechts im Eigenschaftenfenster)

Die Prüfung sollten Sie aber nur dann abschalten, wenn Sie zum Beispiel ausschließlich Daten aus einer untergeordneten Tabelle anzeigen lassen wollen. Denn hier erfolgen in der Standardeinstellung ebenfalls Prüfungen, ob die Daten in den übergeordneten Tabellen vorhanden sind. Wenn der Zugriff auf diese übergeordneten Tabellen nicht möglich ist, wird eine Ausnahme ausgelöst.

Sie müssen dann allerdings selbst dafür sorgen, dass die Informationen beziehungsweise die Daten für die Verbindung zu den übergeordneten Tabellen nicht geändert werden können. Denn dann können wieder „Karteileichen“ entstehen. Wenn Sie ganz sicher gehen wollen, sollten Sie die Prüfung der referenziellen Integrität also nur dann abschalten, wenn es zwingend erforderlich ist.

Missbrauchen Sie die Eigenschaft **EnforceConstraints** auch nicht, um mit Datenbanktabellen weiterarbeiten zu können, bei denen die Daten eigentlich nicht mehr „stimmen“. Denn durch die Prüfungen können Sie wirkungsvoll verhindern, dass der Zusammenhalt der Daten in den verschiedenen Tabellen verloren geht. Einen Mietvertrag ohne Kunden oder ohne Fahrzeug kann es nicht geben. Also dürfen auch in der entsprechenden Datenbanktabelle keine Einträge auftauchen, bei denen die entsprechenden Spalten leer sind beziehungsweise ungültige Schlüssel enthalten. Dass solche „Karteileichen“ erst gar nicht entstehen können, müssen Sie über entsprechende Anweisungen in Ihrem Programm sicherstellen.

Damit auch das Löschen von Kunden in der Einzelansicht und das Löschen von Fahrzeugen in der Einzel- beziehungsweise Listenansicht nicht zu Verletzungen der referentiellen Integrität führen kann, müssen Sie die gesamte Prozedur, die wir Ihnen gerade für die Listenanzeige der Kunden vorgestellt haben, für diese Formulare wiederholen. Dabei gibt es allerdings keine Besonderheiten.

Denken Sie bitte lediglich daran, dass Sie unter Umständen die Reihenfolge beim Beschaffen der Daten im Ereignis **Load** der Formulare verändern müssen. Zuerst müssen die Daten der übergeordneten Tabellen beschafft werden, dann die Daten der untergeordneten Tabellen. Achten Sie auch darauf, dass Sie die Tabellen mit den Reservierungen und den Verträgen jeweils korrekt über den untergeordneten Knoten bei den Datenquellen an einer freien Stelle im Formular einbinden.

Bei Problemen überprüfen Sie bitte auch die Einstellungen für den TableAdapterManager.

Damit wollen wir die Arbeit an unserem Beispielprojekt beenden – auch wenn vielleicht an der einen oder anderen Stelle noch Funktionen fehlen. Sie verfügen jetzt aber über die nötigen Kenntnisse, um zum Beispiel das Löschen von Reservierungen oder die verschiedenen Listenanzeigen für die Fahrzeuge umzusetzen.

Versuchen Sie auch, Erweiterungen der Funktionalität selbst umzusetzen. So könnten Sie zum Beispiel bei Verträgen mehr als nur eine Position ermöglichen. Die einzelnen Positionen könnten Sie dabei in einer weiteren Tabelle ablegen, die über die Vertragsnummer mit dem jeweiligen Vertrag verknüpft ist. Auch die Funktionen für die Reservierungen können Sie noch weiter ausbauen. Denn jetzt lassen sich Reservierungen ja lediglich anlegen. Es gibt weder eine Erinnerung, wenn reservierte Fahrzeuge zurückgegeben werden, noch werden die Fahrzeuge tatsächlich reserviert.

Auch an der Rückgabe von Fahrzeugen können Sie noch Erweiterungen durchführen. Jetzt ist ja zum Beispiel nicht nachvollziehbar, wann das Fahrzeug zurückgegeben wurde. Auch das Anzeigen von Verträgen fehlt noch.

Machen Sie sich bei Änderungen und Erweiterungen die Mühe und führen Sie gegebenenfalls eine neue Analyse und einen neuen Entwurf für das System durch. Andernfalls haben Sie nämlich sehr schnell eine Anwendung, bei der Analyse und Entwurf nicht mehr zum fertigen Produkt passen. Alternativ können Sie auch mit einer agilen Methode Änderungen umsetzen. Wichtig ist aber auch hier ein geplantes und strukturiertes Vorgehen.

Abschließend noch ein Hinweis:

In dem Beispielprojekt für dieses Studienheft haben wir die Eingabeprüfungen für das Formular zur Einzelansicht der Kunden und der Fahrzeuge nicht umgesetzt, da es sich um die Einsendeaufgabe aus dem letzten Studienheft handelt. Sie können diese Prüfungen ja selbst in das Projekt einbauen, wenn Sie die Einsendeaufgaben korrekt gelöst haben.

Zusammenfassung

Die Regeln für die referentielle Integrität greifen nur dann, wenn sich die Tabellen beziehungsweise die TableAdapter gemeinsam in einem Formular befinden.

Um Daten aus einer untergeordneten Tabelle anzuzeigen, müssen Sie die Steuerelemente in einem freien Bereich des Formulars ablegen, in dem die Daten aus der übergeordneten Tabelle angezeigt werden.

Sie können selbst festlegen, ob Einschränkungen in verbundenen Tabellen gelten sollen oder nicht.

Aufgaben zur Selbstüberprüfung

- 4.1 Über welche Eigenschaft legen Sie fest, ob die Einschränkungen zwischen zwei verbundenen Tabellen gelten?

- 4.2 Sie wollen in einem Formular die Daten aus einer über- und einer untergeordneten Tabelle anzeigen. Sie haben die Datenquellen korrekt in das Formular eingefügt, trotzdem wird beim Öffnen des Formulars eine Ausnahme ausgelöst. Wo liegt wahrscheinlich das Problem?

5 Wartung

Nach dem Programmieren und dem Start des produktiven Einsatzes sind die Arbeiten an der Software keineswegs beendet. Denn jetzt folgt die Wartung. Damit werden wir uns in diesem Kapitel beschäftigen. Sie erfahren, warum Software überhaupt gewartet werden muss, welche Formen bei der Wartung unterschieden werden und welche Probleme dabei auftreten können.



Wartung und Pflege

In der Literatur wird gelegentlich zwischen **Software-Wartung** und **Software-Pflege** unterschieden.¹⁰ Die Wartung meint dann das Beheben „echter“ Fehler und die Pflege Anpassungen und Erweiterungen der Software. Da sich die beiden Tätigkeiten allerdings nicht immer eindeutig trennen lassen, verwenden wir hier nur den Begriff Wartung. Er schließt sowohl die Fehlerkorrektur als auch die Weiterentwicklung der Software ein.

Schauen wir uns zuerst an, warum Software überhaupt gewartet werden muss.

5.1 Warum muss Software gewartet werden?

Ein Grund für die Wartung sind Fehler, die sich erst im produktiven Einsatz bemerkbar machen. Zwar lassen sich solche Fehler theoretisch eigentlich ausschließen, in der Praxis zeigt sich aber doch immer wieder, dass selbst sorgfältigste Tests nicht sämtliche Fehler finden können.



Der produktive Einsatz einer Software wird auch als **Realbetrieb** bezeichnet.

Das liegt nicht zuletzt auch an den grafischen Benutzeroberflächen für die Bedienung. Hier kann ein Anwender quasi jederzeit sehr viele verschiedene Aktionen starten und damit auch sehr viele verschiedene Wege durch das Programm nehmen. Jeder dieser theoretisch denkbaren Wege müsste dann auch ausgiebig getestet werden. Das ist schon allein aus Zeitgründen kaum machbar.

Sehen Sie sich zum Beispiel einfach einmal Ihre Textverarbeitung an: Wenn Sie ein Dokument bearbeiten, stehen Ihnen unzählige verschiedene Funktionen gleichzeitig zur Verfügung. Ein Test müsste nicht nur den Aufruf jeder einzelnen Funktion prüfen, sondern auch alle möglichen Folgeaktionen. Dabei müssten dann auch noch sehr viele unterschiedliche Zustände der Software berücksichtigt werden – zum Beispiel: Ist bereits Text eingegeben? Wie lang ist der Text? Ist ein Wort markiert? Ist eine bestimmte Formatierung gesetzt? und so weiter. Sie werden schnell feststellen, dass es Tausende von Möglichkeiten gibt, die im Prinzip alle einzeln getestet werden müssten.

10. Vgl. zum Beispiel Balzert, Band 1, S. 1090.

Schätzungen gehen davon aus, dass für zehn Fehler, die vor dem Realbetrieb im Test gefunden werden, ein Fehler erst im Realbetrieb auftritt.¹¹ Das heißt also: Wenn Sie im Test 100 Fehler entdeckt haben, enthält die Software wahrscheinlich zehn weitere Fehler, die beim Test nicht gefunden wurden.



Ein weiterer Grund für die Wartung liegt in der „Alterung“ von Software. Damit haben wir uns ja bereits ganz kurz bei den typischen Besonderheiten von Software beschäftigt. Schauen wir uns jetzt die Gründe für die „Softwarealterung“ einmal genauer an.

Anders als ein Auto, das über kurz oder lang verschleift, kann Software theoretisch unendlich lange „leben“. Software besteht ja nicht aus einem Material, das nur eine bestimmte Zeit lang verwendet werden kann. Trotzdem hat auch eine Computeranwendung nur eine begrenzte Lebensdauer. Denn die Einsatzbedingungen und auch die Umwelt der Software ändern sich ständig. Damit ändern sich auch die Vorgaben für den Einsatz, die im Analysemodell definiert wurden.

Beispiele für solche Änderungen in der Umwelt kennen Sie ja ebenfalls bereits: der Jahrtausendwechsel und die Euroeinführung. Anwendungen, die mit dem Jahr 2000 oder der neuen Währung nicht zureckkamen, wurden auf einen Schlag unbrauchbar.

Andere Beispiele für den Wechsel in der Umwelt sind zum Beispiel neue Betriebssystemversionen. Hier ändert sich zwar nicht das Analysemodell, trotzdem muss die Software unter Umständen angepasst werden, damit sie unter der neuen Version des Betriebssystems arbeiten kann.

Nicht immer führen Änderungen in der Umwelt gleich zu einer komplett neuen Software. Häufig sind es lediglich vermeintliche Kleinigkeiten – wie zum Beispiel geänderte gesetzliche Vorschriften, geänderte Verfahren oder auch neue Rechtschreibregeln. Aber auch diese Kleinigkeiten müssen sich in der Software wiederfinden.

Die Umwelt einer Software ändert sich ständig. Damit muss eigentlich auch die Software ständig angepasst werden.



Die Auswirkungen von Umweltänderungen lassen sich mit ein wenig Voraussicht und entsprechend flexibler Gestaltung der Software noch im Rahmen halten. Ein anderes Problem ist aber kaum zu lösen: die Veränderung der Umwelt durch die Software selbst. Die Anwendung wird damit quasi selbst Auslöser für ihre eigenen Änderungen.

Schauen wir uns auch dazu ein Beispiel an:

Beispiel 5.1:



Bei unserem Kunden wurde ja bisher noch keine Software eingesetzt. Nach kurzer Einarbeitungszeit sind die Mitarbeiter von der Anwendung Autovermietung 2030 begeistert und machen Vorschläge, wofür die Software noch eingesetzt werden könnte. Ein Mitarbeiter schlägt zum Beispiel vor, auch die Beschaffung von Fahrzeugen komplett in das System zu integrieren. Damit erspare man sich das lästige Erfassen der Fahrzeugdaten per Hand.

11. Quelle: Grady R.B. (1992). *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, Prentice Hall. Zitiert nach Balzert, Band 1, S. 1091.

Werden diese Wünsche berücksichtigt, müssen zwangsläufig auch das Analysemodell überarbeitet und das System erweitert werden. Die Weiterentwicklung kann dann wiederum zu neuen Wünschen bei den Anwendern führen. So entsteht eine Art Spirale.

Im Laufe der Zeit werden die Wünsche der Anwender immer mehr und die Erweiterungen am System halten nicht mehr Schritt. Die Abweichungen zwischen dem gewünschten Zustand und dem tatsächlichen Zustand werden immer größer – die Software verliert schlechend an Nutzen. Etwas übertrieben ausgedrückt: Sie „altet“ und muss irgendwann überholt werden.



Software muss in nahezu allen Fällen mehr oder weniger regelmäßig gewartet werden.

So viel zu den Gründen für die Wartung. Kommen wir nun zu den verschiedenen Formen der Wartung.

5.2 Wartungsformen

Grundsätzlich lassen sich bei der Wartung zwei verschiedene Formen unterscheiden:

- die korrektive Wartung und
- die progressive Wartung.¹²

Bei der **korrekten Wartung** werden echte Fehler behoben und das System optimiert – zum Beispiel die Verarbeitungsgeschwindigkeit verbessert.

Bei der **progressiven Wartung** geht es um die Weiterentwicklung des Systems. Hier werden Anpassungen vorgenommen und neue Funktionen eingebaut.

Schätzungen gehen davon aus, dass die korrektive Wartung knapp 40 Prozent der gesamten Wartung umfasst und die progressive Wartung knapp 60 Prozent.

Die beiden groben Wartungsformen lassen sich auch noch feiner untergliedern:

- die **präventive Wartung**¹³ beugt Fehlern vor,
- die **korrigierende Wartung** behebt Fehler,
- die **adaptierende Wartung**¹⁴ setzt Anpassungen und Änderungen um,
- die **perfektionierende Wartung** sorgt für die Optimierung und
- die **wertsteigernde Wartung** entwickelt das System weiter.

12. Progressiv bedeutet so viel wie fortschrittlich oder sich stufenweise entwickelnd.

13. Prävention bedeutet so viel wie Vorbeugung oder Verhütung.

14. Adaption bedeutet so viel wie Anpassung oder Umbau.

Für die korrigierende Wartung findet sich auch der Begriff **Bug fixing**. Übersetzt bedeutet er so viel wie „Fehlerbehebung“.



Für Wartungstätigkeiten, die nicht nur die Fehlerbeseitigung oder -vorbeugung als Ziel haben, wird auch der Begriff **Redesign** benutzt.

Neben Wartungstätigkeiten, die die Funktion der Software verändern, werden häufig auch interne Optimierungen am Quelltext vorgenommen, die zum Beispiel die Wartung oder die Fehlerkorrektur erleichtern sollen. Solche Tätigkeiten, die an der Funktionalität nichts verändern, werden auch unter dem Begriff **Refactoring** zusammengefasst.

5.3 Probleme bei der Wartung

Die Software-Wartung verläuft häufig nicht reibungslos, sondern weist einige typische Schwierigkeiten auf.

Das erste Problem: Der Aufwand wird häufig unterschätzt.

Mit dem Übergang in den Realbetrieb gilt das Projekt bei vielen Entwicklern und auch beim Projektmanagement als „erledigt“. Tatsächlich aber beansprucht die Wartung sehr viel Zeit und auch Personal. Schätzungen gehen davon aus, dass auch bei vergleichsweise einfacher Software der Aufwand für die Wartung 30 Prozent des gesamten Projektaufwands ausmacht. Bei sehr komplexen Systemen, die lange eingesetzt werden, kann dieser Anteil sogar auf bis zu 80 Prozent steigen.

Wartung ist viel mehr als nur ein paar abschließende Korrekturen, die nebenbei erledigt werden können.



Das zweite Problem: Wenn eine Software in den Realbetrieb geht, werden die Entwickler in der Regel in neuen Projekten eingesetzt.

Damit stehen oft die Personen, die ein System erstellt haben, für die Wartung entweder gar nicht mehr oder nur noch eingeschränkt zur Verfügung. Die Wartung erfolgt dann nicht selten durch Personal, dem Detailkenntnisse zu der Software fehlen. Dadurch schleichen sich schnell neue Fehler ein, die dann natürlich auch wieder behoben werden müssen.

Werden die Entwickler dagegen gleichzeitig in der Entwicklung eines neuen Software-Systems und in der Wartung eines bereits fertigen Systems eingesetzt, entsteht oft ein Zeitproblem. Die Wartung wird nebenbei erledigt, entsprechend ist dann auch das Ergebnis.

Das dritte Problem: Fehlerkorrekturen müssen oft unter großem Zeitdruck durchgeführt werden.

Die Software wird ja bereits produktiv eingesetzt und **muss** korrekt funktionieren. Eine zeitaufwendige Fehlersuche und -korrektur, die unter Umständen auch mit einer zeitweiligen Abschaltung des Systems verbunden ist, wäre für viele Unternehmen eine Katastrophe. Nicht selten wird dann vor allem an den Symptomen eines Fehlers „herumgedoktert“, um das System möglichst schnell wieder funktionstüchtig zu machen.

Auf Regressionstests, die eigentlich nach einer Korrektur zwingend erforderlich sind, wird aus Zeitgründen verzichtet. Im Ergebnis führt eine Korrektur dann nicht selten gleich zu mehreren neuen Fehlern, die sich aber erst später bemerkbar machen.



Ein **Regressionstest** wiederholt sämtliche Tests, die vor einer Änderung durchgeführt wurden.

Auch die ordentliche Dokumentation der Korrektur bleibt aus Zeitgründen nicht selten auf der Strecke. Das wiederum kann dazu führen, dass die Dokumentation irgendwann nicht mehr zum System passt. Damit wird sie unbrauchbar.

Das vierte Problem: der „Titanic-Effekt“

Dieses Phänomen tritt häufig bei Programmierern auf, die auch an der Entwicklung beteiligt waren. Die Wartung wird als „Kleinigkeit“ abgetan, da das System ja eigentlich schon funktioniert und auch im Detail bekannt ist. Auch Fehler werden nicht selten unterschätzt – nach dem Motto: „Das mache ich mal eben“. Dass eine Änderung möglicherweise Auswirkungen auf die gesamte Software haben kann, wird gerne „übersehen“.

Hinweis:

Der „Titanic-Effekt“ hat seinen Namen von dem amerikanischen Luxusdampfer. Das Schiff galt als unsinkbar. Entsprechend leichtfertig begab man sich auf die Jungfern fahrt. Der Glaube „Uns kann nichts passieren“ führte zu einer riesigen Katastrophe: Das Schiff rammte einen Eisberg und ging unter. Dabei ertranken die meisten Passagiere.

Die Wartung kann also dazu führen, dass die Wartbarkeit eines Systems schlechend immer weiter abnimmt – die Software wird „verschlimmbessert“. Das liegt nicht zuletzt daran, dass die Wartung sehr oft immer nur einzelne Teile der Software betrachtet und das System als Ganzes dabei aus den Augen verliert.



Je anspruchsvoller die Wartung wird, desto qualifizierter muss das Personal sein und desto höher werden die Kosten für die Wartung.

Um solche Probleme möglichst zu vermeiden, muss die Wartung durch ein Konfigurationsmanagement genau dokumentiert und geplant werden. Damit beschäftigen wir uns im nächsten Kapitel.

Zusammenfassung

Nach dem Start des Realbetriebs muss die Software gewartet werden.

Bei der Wartung werden Fehler korrigiert sowie Anpassungen und Erweiterungen vorgenommen.

Die Wartung kann zu einer ständig abnehmenden Wartbarkeit der Software führen. Das liegt oft daran, dass der Aufwand unterschätzt wird und Korrekturen nebenbei erfolgen.

Aufgaben zur Selbstüberprüfung

- 5.1 Nennen Sie mindestens zwei Gründe, warum Software gewartet werden muss.

- 5.2 In einer Software werden in der Testphase 50 Fehler entdeckt. Wie viele Fehler machen sich wahrscheinlich erst im Realbetrieb bemerkbar?

- 5.3 Welche beiden grundsätzlichen Formen der Wartung kennen Sie? Beschreiben Sie die beiden Formen bitte kurz.

- 5.4 Was verstehen Sie unter dem Begriff „Bug fixing“?

6 Konfigurationsmanagement

In diesem Kapitel erfahren Sie, wie Sie Änderungen an einer Software durch das Konfigurationsmanagement dokumentieren und planen.



Für das Konfigurationsmanagement werden auch der englische Begriff **Software Configuration Management** und die Abkürzung **SCM** benutzt.

Die Abkürzung SCM findet sich auch in der Betriebswirtschaft. Dort steht sie allerdings für **Supply Chain Management** (übersetzt etwa: „Verwaltung von Lieferketten“). Aufgabe des *Supply Chain Managements* ist – etwas vereinfacht dargestellt – eine möglichst optimale Zusammenarbeit der verschiedenen Teile in einer komplexen Lieferkette – zum Beispiel der Lieferkette eines Automobilherstellers.

6.1 Aufgaben des Konfigurationsmanagements

In der Praxis kommen im Realbetrieb von vielen Seiten Fehlermeldungen und Änderungswünsche auf einen Entwickler zu – direkt von den Anwendern, vom Vertrieb des Softwareherstellers, vielleicht auch von möglichen Kunden. Auch die Formen der Meldungen sind dabei sehr unterschiedlich – angefangen bei einer Fehlermeldung über eine einfache Notiz bis hin zu einer Unterhaltung beim Mittagessen.

Diese Flut gilt es jetzt zu strukturieren und zu verwalten. Andernfalls entsteht ein großes Durcheinander: Niemand weiß mehr so recht, was alles schon geändert worden ist und was noch geändert werden soll, ob gerade eine Änderung durchgeführt wird, wer diese Änderung durchführt, welche Auswirkungen diese Änderung hat und so weiter.

In einem ersten Schritt werden zunächst einmal alle Meldungen an einer zentralen Stelle gesammelt. Hier wird dann geprüft, ob es sich um einen Fehler oder um einen Änderungswunsch handelt, und eine Priorität für die Korrektur beziehungsweise die Erweiterung festgelegt. Über die Priorität kann dann auch eine erste grobe Zeitplanung vorgenommen werden.

Tipp:

Für Fehlermeldungen der Anwender sollten Sie ein eigenes Formular entwickeln. Dadurch stellen Sie sicher, dass die Meldungen einen identischen Aufbau haben. Bitten Sie die Anwender in dem Formular mindestens um folgende Informationen:

- *Wer hat die Fehlermeldung wann erfasst?*
- *Wo ist der Fehler genau aufgetreten?*
- *Welche Auswirkungen hatte der Fehler?
Bitten Sie hier um eine möglichst exakte Beschreibung.*
- *Tritt der Fehler immer wieder auf?*
- *Wenn bereits unterschiedliche Versionen der Software vorhanden sind:
Mit welcher Version arbeitet der Anwender?*

Die Formulare können Sie dann entweder in Papierform oder elektronisch verteilen – zum Beispiel über das Internet oder das Netzwerk des Unternehmens, in dem die Anwendung eingesetzt wird.

6.2 Die Versionsverwaltung

Neben der Sammlung und Strukturierung der Fehlermeldungen und Änderungswünsche übernimmt das Konfigurationsmanagement noch eine weitere wichtige Aufgabe: Es verwaltet unterschiedliche Versionen der Software.

Schauen wir uns auch dazu ein Beispiel an:

Beispiel 6.1:



Unser Kunde hat beschlossen, die Anwendung Autovermietung 2030 weiterzuentwickeln. Dabei sollen auch neue Funktionen eingebaut werden, die von den Anwendern gewünscht wurden.

Diese Änderungen können jetzt natürlich in die bereits vorhandene Version der Software aufgenommen werden. Dabei muss aber die erste Version in jedem Fall erhalten bleiben. Es müssen also zwei Versionen der Software angelegt und getrennt verwaltet werden – auch wenn die erste Version unter Umständen überhaupt nicht mehr eingesetzt werden soll.

Der Grund für diesen scheinbar überflüssigen Aufwand ist schnell gefunden: Bei den Erweiterungen in der neuen Version werden mit ziemlicher Sicherheit Teile der alten Version überarbeitet und geändert. Dabei können sich auch neue Fehler einschleichen. Die Teile der alten Version, die vor den Änderungen funktioniert haben, arbeiten in der neuen Version auf einmal nicht mehr richtig. Wenn Sie jetzt die alte Version mit den Änderungen überschrieben haben, sind die korrekt funktionierenden Teile unwiederbringlich verloren.

Wenn Sie sich dann noch vor Augen halten, dass in der Regel mehrere Entwickler parallel an einem System arbeiten, wird schnell klar, dass ohne Versionsverwaltung ein gigantisches Chaos droht. Der eine Entwickler überschreibt mit seinen Änderungen unter Umständen die Änderungen eines anderen Entwicklers. Fehler, die scheinbar behoben sind, tauchen urplötzlich wieder auf, weil ein Programmierer – ohne es zu wissen – eine nicht mehr aktuelle Version eines Quelltextes benutzt hat.

Deshalb werden sämtliche Bestandteile einer Version zu einer Art Paket gebündelt und sauber von den anderen Versionen getrennt abgelegt. In dem Paket befinden sich dabei nicht nur die Quelltexte, sondern zum Beispiel auch alle Dokumentationen, Diagramme, Testfälle und so weiter, die zu der jeweiligen Version gehören.

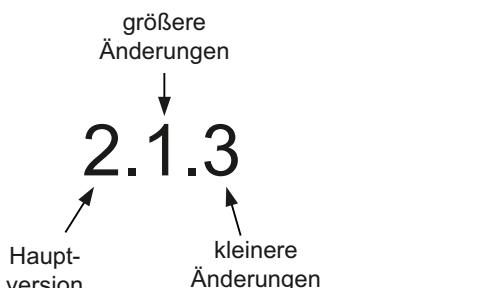


Abb. 6.1: Schema für Versionsnummern

Jede einzelne Version wird dann eindeutig identifiziert – zum Beispiel durch eine **Versionsnummer** mit drei Zahlen. Die erste Zahl gibt dabei die Hauptversion an. Dann folgt eine Zahl für größere Änderungen – zum Beispiel Anpassungen. Die dritte Zahl schließlich markiert kleinere Änderungen – zum Beispiel die Behebung von „Schönheitsfehlern“.

Bei jeder durchgeführten Änderung wird die entsprechende Zahl um den Wert 1 erhöht. Ausgangspunkt ist dabei normalerweise die Version 1.0.0. Bei einer kleineren Änderung würde aus dieser Version dann die Version 1.0.1. Wird an dieser Version eine größere Änderung vorgenommen, erhält die neue Version die Nummer 1.1.1 und so weiter.

Falls mehrere Entwickler gleichzeitig an einer Version arbeiten, kann auch noch eine weitere Zahl am Ende ergänzt werden. Sie steht dann für den **Entwicklungszweig**. An der Version 1.1.1.1 könnte zum Beispiel ein Entwicklerteam arbeiten und an der Version 1.1.1.2 ein anderes Entwicklerteam.

Hinweis:

Die Versionsnummern unter 1 – also zum Beispiel 0.0.1 oder 0.1.2 – werden üblicherweise für Testversionen benutzt.

Anhand der Versionsnummer lassen sich dann nicht nur die verschiedenen Versionen eindeutig unterscheiden, sondern es ist auch jederzeit klar, welche Version im Moment die aktuellste Version der Software ist.

Wie die Versionsnummern nun genau vergeben werden, ist die Entscheidung jedes einzelnen Softwareunternehmens. Zum Teil werden auch Kombinationen aus Buchstaben und Zahlen verwendet – zum Beispiel 4.7b – oder auch Versionsnummern aus Marketinggründen einfach übersprungen. Wenn beispielsweise ein Konkurrent die Version 7 seines Produkts auf den Markt bringt, erhält die eigene Software kurzerhand die Versionsnummer 8 – selbst wenn es sich vielleicht erst um die dritte Hauptversion handelt. Damit soll der Eindruck entstehen, das eigene Produkt sei weiterentwickelt als das Produkt des Konkurrenten.



Beim Versionsmanagement finden sich häufig auch die Begriffe **Release** und **Update**. Ein Release¹⁵ bezeichnet Versionen, die tatsächlich auch an Kunden ausgeliefert werden. Ein Update¹⁶ ist eine aktualisierte Version einer Software.

Bei Visual Studio können Sie die Versionsnummer bei den Projekteigenschaften einstellen. Klicken Sie dazu unten im Menü **Projekt** auf den Eintrag **Eigenschaften...** und wechseln Sie anschließend in das Register **Veröffentlichen**.

15. Release bedeutet übersetzt so viel wie „Freigabe“ oder „Veröffentlichung“.

16. Update bedeutet übersetzt so viel wie „Aktualisierung“ oder „Neufassung“.

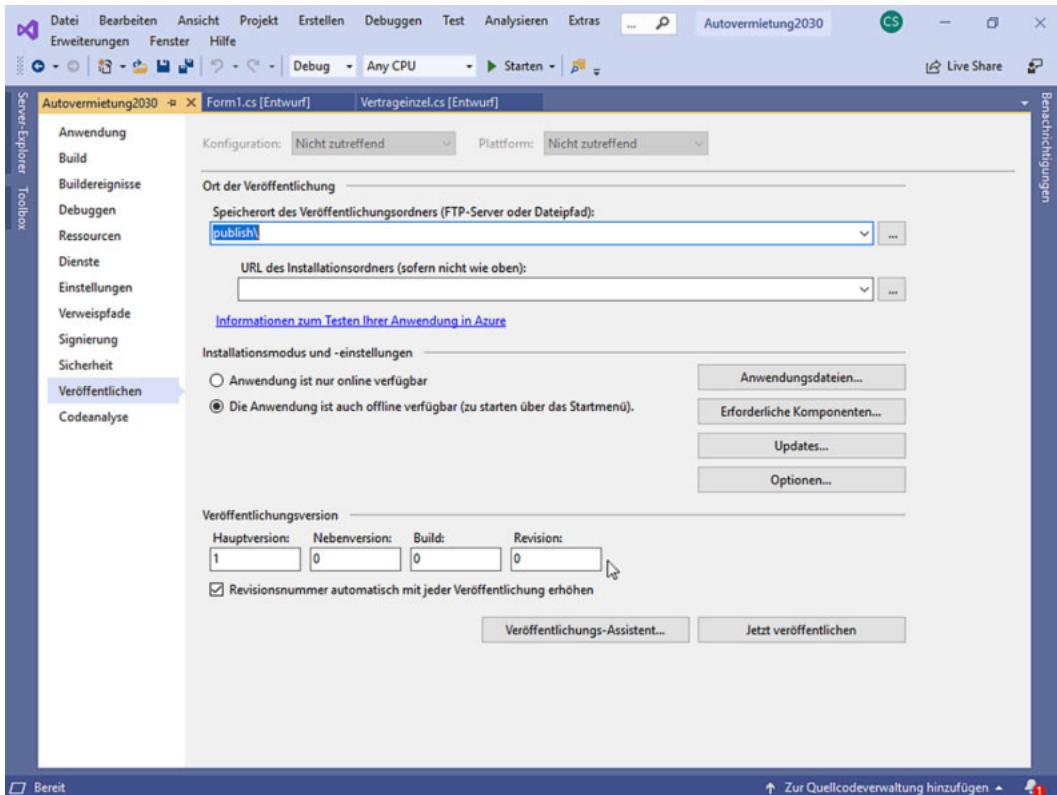


Abb. 6.2: Die Einstellungen für die Versionsnummer

Über die vier Felder im Bereich **Veröffentlichungsversion** können Sie dann eine vierstellige Versionsnummer vergeben.

6.3 Geplante Wartung und Schnellschüsse

Grundsätzlich muss beim Konfigurationsmanagement zwischen einer geplanten Wartung und einem Schnellschuss unterschieden werden.

Bei der **geplanten Wartung** wird zunächst eine grobe Skizze entworfen, welche Funktionen in welcher Version enthalten sein sollen und wann die jeweilige Version fertiggestellt werden soll. Ausgangspunkt dieser ersten Planung können zum Beispiel Marktanalysen und auch die Vorstellungen des Softwareherstellers sein. Der grobe Zeitplan wird dann mit aktuellen Änderungswünschen von Anwendern und Kunden abgeglichen und dabei gegebenenfalls auch erweitert.

Neben der Priorität der einzelnen Änderungen beziehungsweise Erweiterungen wird außerdem immer der zeitliche Aufwand geschätzt. Auf diese Weise können mehr oder weniger verlässliche Aussagen getroffen werden, welche Fehler in welcher Version behoben sein werden beziehungsweise welche Änderungen in welcher Version berücksichtigt werden. Es entsteht ein fester Zeitplan mit festen Änderungsvorgaben.

Anhand dieses Zeitplans werden dann die neuen Versionen von den Entwicklern umgesetzt. Dabei durchläuft eine Version – je nach Umfang und Art der Änderung – unter Umständen auch wieder die einzelnen Projektphasen von Anfang an. Wartung und Entwicklung werden damit eng verzahnt. Auch dem Anwender kann durch den Zeitplan mitgeteilt werden, wann und in welchem Umfang seine Wünsche berücksichtigt werden.

Durch die Versionsverwaltung können auch mehrere Versionen gleichzeitig bearbeitet werden, die sich alle in unterschiedlichen Stadien befinden. Wird als aktuelle Version die Nummer 2.6 eingesetzt, kann sich Version 2.7 zum Beispiel bereits in den abschließenden Tests befinden und Version 2.8 im Entwurf.

Bei schweren Fehlern im Realbetrieb ist der normale Wartungsablauf unter Umständen aber zu langsam. Dann wird ein **Schnellschuss** durchgeführt, der außerhalb der geplanten Wartung erfolgt.



Bei einem Schnellschuss wird ein gemeldeter Fehler möglichst direkt und möglichst schnell behoben.

Ein Schnellschuss ist zwar eine Abweichung von der geplanten Wartung, er muss aber trotzdem in der Versionsverwaltung abgebildet werden. Andernfalls tauchen die Probleme, die der Schnellschuss eigentlich beheben sollte, in späteren Versionen der Software wieder auf.

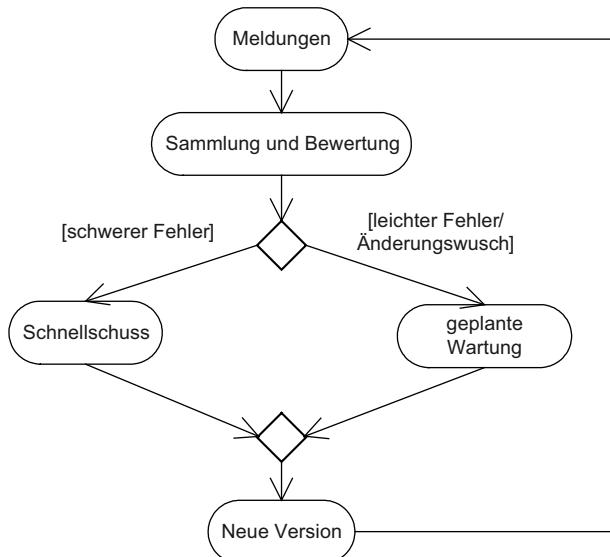


Abb. 6.3: Schnellschuss und geplante Wartung

Beispiele für Schnellschüsse finden sich reichlich. Denken Sie nur an die Service Packs und Updates von Microsoft für die Aktualisierung der Office-Produkte und der Windows-Betriebssysteme.

6.4 Werkzeuge für das Konfigurationsmanagement

Abschließend wollen wir noch einen kurzen Blick auf Werkzeuge für das Konfigurationsmanagement werfen.

Bei diesen Werkzeugen handelt es sich um Programme, die zahlreiche Tätigkeiten automatisch ausführen beziehungsweise gezielt unterstützen. Dazu gehören zum Beispiel:

- das Anlegen neuer Versionen,
- die Zuordnung von Versionsnummern zu Dokumenten und umgekehrt,
- der automatische Vergleich zweier Versionen eines Dokuments,

- das Zusammenführen von Änderungen in einem neuen Dokument und
- die automatische Dokumentation von Unterschieden zwischen zwei Versionen.

Ein bekannter Vertreter solcher Werkzeuge ist zum Beispiel RCS (Revision Control System¹⁷). Das Programm ist frei erhältlich. Mittlerweile verfügen aber auch viele Entwicklungsumgebungen über eigene Werkzeuge, die zumindest Änderungen an Quelltexten dokumentieren und auch verwalten können. Auch in Visual Studio ist solch ein Werkzeug integriert – der Team Foundation Server von Microsoft. Er wird aber bei Visual Studio Community nicht mitgeliefert.

So viel zum Konfigurationsmanagement.

Zusammenfassung

Unterschiedliche Versionen einer Software müssen sauber voneinander getrennt abgelegt werden. Für die Identifikation kann eine eindeutige Versionsnummer verwendet werden.

Neben der geplanten Wartung gibt es noch Schnellschüsse.

Das Konfigurationsmanagement wird durch spezielle Programme unterstützt.

Aufgaben zur Selbstüberprüfung

- 6.1 Welche beiden wesentlichen Teilaufgaben hat das Konfigurationsmanagement?

- 6.2 Sie finden bei einer Software die Versionsnummer 2.1.6. Welche Informationen liefert Ihnen diese Nummer?

- 6.3 Was ist ein Schnellschuss?

17. *Revision* bedeutet übersetzt so viel wie „Änderung“ oder „Überarbeitung“.

Schlussbetrachtung

Sie dürfen stolz auf sich sein! Sie haben eine anspruchsvolle Anwendung geplant, analysiert, entworfen und umgesetzt. Auch wenn Ihnen der eine oder andere vorbereitende Schritt im Moment vielleicht noch ein wenig umständlich erscheint, sollten Sie auf die Planung, die Analyse und den Entwurf bei komplexeren Anwendungen nicht verzichten. Die Zeit, die Sie damit verbringen, sparen Sie nämlich bei der Umsetzung sehr schnell wieder ein. So konnten wir ja zum Beispiel die komplette Datenbankstruktur unserer Anwendung direkt übernehmen.

Christoph Siebeck

A. Lösungen zu den Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Die drei möglichen Beziehungen sind die 1:n-Beziehung, die 1:1-Beziehung und die n:m-Beziehung.

Bei der 1:n-Beziehung gibt es für genau einen Datensatz in der einen Tabelle beliebig viele Datensätze in der anderen Tabelle.

Bei der 1:1-Beziehung gibt es für einen Datensatz in der einen Tabelle genau einen Datensatz in der anderen Tabelle.

Bei der n:m-Beziehung gibt es für beliebige viele Datensätze in der einen Tabelle beliebig viele Datensätze in der anderen Tabelle.

- 1.2 Die referentielle Integrität beschreibt Regeln für die Verarbeitung von Daten in verbundenen Tabellen. Dazu gehört zum Beispiel, dass in der einen Tabelle keine Werte in das verbundene Feld eingegeben werden können, die in der anderen Tabelle nicht vorhanden sind.

- 1.3 Neu angelegte Tabellen werden nicht automatisch zu den Datenquellen hinzufügt. Sie müssen die Tabelle selbst über den Assistenten hinzufügen.

- 1.4 Im ersten Schritt markieren Sie den Primärschlüssel in der übergeordneten Tabelle durch einen Mausklick in den entsprechenden Zeilenkopf. Danach ziehen Sie die Spalte am Zeilenkopf auf die Spalte des Fremdschlüssels in der untergeordneten Tabelle.

- 1.5 Nein, die Beziehung kann nicht hergestellt werden. Die Spalten müssen denselben Datentyp haben. Der Name spielt keine Rolle.

Kapitel 3

- 3.1 Im ersten Schritt legen Sie das Formular an und blenden die Datenquellen ein. Wechseln Sie dann in den Zweig, in dem sowohl die über- als auch die untergeordnete Tabelle angezeigt werden. Legen Sie zunächst für die untergeordnete Tabelle die gewünschten Steuerelemente fest und stellen Sie **Details** als Ansicht ein. Ziehen Sie anschließend die Tabelle in das Formular.

Ziehen Sie dann die komplette übergeordnete Tabelle, aus der Sie die Daten anzeigen wollen, auf das gewünschte Steuerelement im Formular.

- 3.2 Um einen neuen Datensatz anzulegen, verwenden Sie die Methode `AddNew()` für das Steuerelement **BindingSource** der Datenbanktabelle.

- 3.3 Die Anweisung für den Aufruf der Methode lautet

```
BestellungenTableAdapter.FillByDatum(minidbDataSet.  
Bestellungen, datum);
```

Wichtig sind vor allem die Angabe des TableAdapters und die Angabe der Tabelle als erstes Argument. Der Name des DataSets beim ersten Argument und die Bezeichnung des zweiten Arguments sind beliebig.

Kapitel 4

- 4.1 Die Einschränkungen werden über die Eigenschaft **EnforceConstraints** des DataSets ein- beziehungsweise ausgeschaltet.
- 4.2 Wahrscheinlich werden die Daten für die Tabellen in der falschen Reihenfolge beschafft. Sie müssen sicherstellen, dass erst die Daten für die übergeordnete Tabelle geladen werden und dann die Daten für die untergeordnete Tabelle.

Kapitel 5

- 5.1 Für die Wartung gibt es mehrere Gründe:
- Die Software kann noch echte Fehler enthalten.
 - Die Umwelt der Software ändert sich.
 - Die Anwender stellen neue Anforderungen an die Software.
- Für die richtige Antwort reicht es aus, wenn Sie zwei dieser Gründe angegeben haben.
- 5.2 Die Software enthält wahrscheinlich noch fünf Fehler, die erst im Realbetrieb auftreten. Schätzungen gehen davon aus, dass für 10 gefundene Fehler ein Fehler erst im Realbetrieb sichtbar wird.
- 5.3 Grundsätzlich werden die korrektive Wartung und die progressive Wartung unterschieden. Die korrektive Wartung behebt echte Fehler und optimiert das System. Die progressive Wartung entwickelt das System weiter.
- 5.4 *Bug fixing* ist eine andere Bezeichnung für die korrigierende Wartung – also die Behebung von Fehlern.

Kapitel 6

- 6.1 Das Konfigurationsmanagement hat zwei wesentliche Aufgaben:
- Es sammelt und bewertet Fehlermeldungen und Änderungswünsche.
 - Es verwaltet unterschiedliche Versionen einer Software.
- 6.2 Es handelt sich um die zweite Hauptversion der Software. In dieser Hauptversion wurden bisher einmal größere Änderungen durchgeführt und sechs Mal kleinere Änderungen.
- 6.3 Ein Schnellschuss ist eine möglichst schnelle Reaktion auf einen schweren Fehler. Mit dem Schnellschuss wird von der geplanten Wartung abgewichen.

B. Glossar

1:1-Beziehung	Eine 1:1-Beziehung ist eine mögliche Beziehung zwischen Datenbanktabellen. Für genau einen Eintrag in der einen Tabelle kann es nur genau einen Eintrag in der anderen Tabelle geben.
1:n-Beziehung	Eine 1:n-Beziehung ist eine mögliche Beziehung zwischen Datenbanktabellen. Für genau einen Eintrag in der einen Tabelle kann es mehrere Einträge in der anderen Tabelle geben.
Abfrage	Über eine Abfrage werden Daten aus einer Datenbank anhand bestimmter Kriterien selektiert.
Adaptierende Wartung	Die adaptierende Wartung setzt Anpassungen und Änderungen um.
Adaption	Adaption bedeutet so viel wie Anpassung oder Umbau.
ADO.NET	ADO.NET ist eine Schnittstelle für den Zugriff auf Daten in Datenbanken. Sie gehört zum <i>.NET Framework</i> .
Datenbank	Eine Datenbank ist eine strukturierte Sammlung von Daten in elektronisch verarbeitbarer Form.
Datenbank-Management-System	Ein Datenbank-Management-System übernimmt die Verwaltung von Datenbanken und stellt Anwendungsprogramme für die Verwaltung zur Verfügung.
Datenfeld	Ein Datenfeld speichert Teilinformationen eines Datensatzes. Jedes Datenfeld hat bestimmte Eigenschaften – zum Beispiel die Länge oder den Datentyp.
Datenprovider	Die Datenprovider gehören zu ADO.NET. Sie ermöglichen den Zugriff auf unterschiedliche Datenbank-Management-Systeme.
Datenquelle	Eine Datenquelle wird – etwas vereinfacht – von einem Datenprovider zur Verfügung gestellt und bietet die Daten aus Datenbanktabellen an. Die Datenquelle kann mit Steuerelementen in einem Formular verbunden werden.
Datensatz	Ein Datensatz speichert logisch zusammengehörige Informationen in einer Datenbank beziehungsweise einer Datenbanktabelle.
DBMS	Abkürzung für Datenbank-Management-System.
Fremdschlüssel	Ein Fremdschlüssel ist ein Primärschlüssel einer Datenbanktabelle, der in einer anderen Tabelle zum Herstellen einer Beziehung zwischen den Tabellen genutzt wird.

Konfigurationsmanagement	Das Konfigurationsmanagement verwaltet und strukturiert Änderungswünsche und Fehlermeldungen. Außerdem übernimmt es die Verwaltung unterschiedlicher Versionen einer Software.
Korrektive Wartung	Bei der korrekiven Wartung werden „echte“ Fehler behoben und das System optimiert – zum Beispiel die Verarbeitungsgeschwindigkeit verbessert.
Korrigierende Wartung	Die korrigierende Wartung behebt „echte“ Fehler.
n:m-Beziehung	Eine n:m-Beziehung ist eine mögliche Beziehung zwischen Datenbanktabellen. Für mehrere Einträge in der einen Tabelle kann es mehrere Einträge in der anderen Tabelle geben.
Normalisierung	Bei der Normalisierung werden Daten auf mehrere einzelne Datenbanktabellen verteilt. Diese einzelnen Tabellen werden dann miteinander verbunden.
Perfektionierende Wartung	Die perfektionierende Wartung soll die Arbeitsweise eines Systems optimieren.
Primärschlüssel	Über den Primärschlüssel kann ein Datensatz in einer Datenbanktabelle eindeutig identifiziert werden.
Progressive Wartung	Bei der progressiven Wartung wird ein System angepasst und weiterentwickelt.
Realbetrieb	Als Realbetrieb wird der produktive Einsatz einer Software beim Kunden bezeichnet.
Redesign	<i>Redesign</i> ist eine zusammenfassende Bezeichnung für alle Wartungsaktivitäten, die nicht nur Fehler beheben oder Fehlern vorbeugen. Dazu gehören zum Beispiel Optimierungen und Weiterentwicklungen.
Refactoring	Das <i>Refactoring</i> fasst Wartungsmaßnahmen zur internen Optimierung eines Quelltextes zusammen – zum Beispiel, um die Wartung oder Fehlersuche zu vereinfachen. An der Funktionalität der Software werden keine Änderungen vorgenommen.
Referenzielle Integrität	Die referenzielle Integrität beschreibt Regeln für die Verarbeitung von Daten in verbundenen Datenbanktabellen. Sie verhindert zum Beispiel, dass ungültige Werte eingegeben werden oder dass Datensätze gelöscht werden, die für einen Bezug zwingend erforderlich sind.
Regressionstest	In einem Regressionstest werden alle Tests, die vor einer Änderung durchgeführt wurden, noch einmal wiederholt.

Relation	Eine Relation ist im relationalen Datenmodell eine Tabelle.
Relationales Datenmodell	Beim relationalen Datenmodell werden Daten in Tabellen abgelegt, die miteinander verbunden werden. Die Spalten der Tabelle – im relationalen Datenmodell Attribut genannt – beschreiben dabei die Eigenschaften der Daten.
Schnellschuss	Ein Schnellschuss dient zur schnellen und direkten Behebung von schweren Fehlern im Realbetrieb. Ein Schnellschuss erfolgt außerhalb der geplanten Wartung, muss aber in die Versionsverwaltung integriert werden.
SCM	SCM steht als Abkürzung für <i>Software Configuration Management</i> oder als Abkürzung für <i>Supply Chain Management</i> .
Server-Explorer	Der Server-Explorer ist ein Teil der Oberfläche von Visual Studio. Über den Server-Explorer können Sie zum Beispiel Verbindungen zu einer Datenbank herstellen.
Software Configuration Management (SCM)	<i>Software Configuration Management</i> (SCM) ist der englische Ausdruck für Konfigurationsmanagement.
Software-Pflege	Bei der Software-Pflege wird eine bestehende Software angepasst und erweitert.
Software-Wartung	Bei der Software-Wartung werden Fehler im Realbetrieb korrigiert und die Anwendung angepasst und erweitert. Teilweise ist mit der Wartung aber auch nur die Korrektur echter Fehler gemeint.
SQL	SQL steht für <i>Structured Query Language</i> . SQL ist ein Sprachstandard für die Kommunikation zwischen Anwendungsprogrammen und relationalen Datenbank-Management-Systemen.
SQL Server	SQL Server ist ein Datenbank-Management-System, das zu Visual Studio Community gehört.
Structured Query Language	Siehe SQL.
Supply Chain Management (SCM)	Aufgabe des <i>Supply Chain Managements</i> (SCM) ist – etwas vereinfacht dargestellt – eine möglichst optimale Zusammenarbeit der verschiedenen Teile in einer komplexen Lieferkette – zum Beispiel der Lieferkette eines Automobilherstellers.
Übergeordnete Tabelle	Die übergeordnete Tabelle ist bei Beziehungen zwischen Tabellen die Tabelle, die die vollständigen Daten enthält.

Untergeordnete Tabelle	Die untergeordnete Tabelle ist bei Beziehungen zwischen Tabellen die Tabelle, aus der Informationen über den Schlüssel in der anderen Tabelle beschafft werden.
Update	Ein Update ist eine aktualisierte Version einer Software.
Versionsnummer	Die Versionsnummer wird für die eindeutige Identifikation einer Software verwendet. Häufig wird ein Schema mit drei Zahlen benutzt, die durch einen Punkt getrennt werden. Die erste Zahl steht dabei für die Hauptversion, die zweite Zahl für größere Änderungen und die dritte Zahl für kleinere Änderungen.
Versionsverwaltung	Die Versionsverwaltung trennt unterschiedliche Versionen einer Software voneinander und vergibt für jede Version eine eindeutige Identifikation.

C. Literaturverzeichnis

Literaturnachweis

Balzert, H. (2000). *Lehrbuch der Software-Technik. Band 1: Software-Entwicklung.* 2. Auflage. Heidelberg: Spektrum Akademischer Verlag.

Verwendete Literatur

Balzert, H. (2009). *Lehrbuch der Softwaretechnik. Basiskonzepte und Requirements Engineering.* 3. Auflage. Heidelberg: Spektrum Akademischer Verlag.

Balzert, H. (2008). *Lehrbuch der Softwaretechnik. Softwaremanagement.* 2. Auflage. Heidelberg: Spektrum Akademischer Verlag.

Empfohlene Literatur

Kühnel, A. (2019) *C# 8 mit Visual Studio: Das umfassende Handbuch. Spracheinführung, Objektorientierung, Programmietechniken.* 8. Auflage. Bonn: Rheinwerk.

Theis, T. (2019) *Einstieg in C# mit Visual Studio 2019. Ideal für Programmieranfänger.* 6. Auflage. Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 1.1	Die Tabellen für die Autovermietung	3
Abb. 1.2	Die neue Tabelle im Tabellen-Designer	6
Abb. 1.3	Die Tabelle Vertrag	8
Abb. 1.4	Die Tabelle Reservierungen	9
Abb. 1.5	Die Auswahl der neu erstellten Tabellen im Assistenten	10
Abb. 1.6	Die aktualisierten Datenquellen (oben links in der Abbildung)	11
Abb. 1.7	Der DataSet-Designer (die weiteren Bereiche sind zur besseren Übersicht ausgeblendet)	12
Abb. 1.8	Das Fenster Beziehung	13
Abb. 1.9	Die geänderte Option für die Beziehung	14
Abb. 1.10	Die Beziehung zwischen den Tabellen Kunde und Vertrag (die Linie für die Beziehung verläuft hinter der Darstellung der anderen beiden Tabellen)	15
Abb. 1.11	Die neue Beziehung (die Linie oben zwischen den Tabellen Fahrzeug und Vertrag)	16
Abb. 1.12	Die Beziehungen zwischen den Tabellen (die Tabellen sind zur besseren Übersicht verschoben)	17
Abb. 2.1	Die Tabelle Kunde mit den untergeordneten Tabellen	21
Abb. 2.2	Die Auswahlmöglichkeiten für das Feld kNummer der untergeordneten Tabelle Vertrag	22
Abb. 2.3	Die Detailansicht mit den ComboBoxen	22
Abb. 2.4	Die Verbindung zur Tabelle Kunde (rechts im Eigenschaftenfenster)....	23
Abb. 2.5	Die Liste der Spalten für vertragBindingSource	25
Abb. 2.6	Das Formular für die Verträge im praktischen Einsatz	25
Abb. 2.7	Das überarbeitete Formular für die Vermietung	27
Abb. 2.8	Das überarbeitete Formular im praktischen Einsatz	29
Abb. 2.9	Das Formular für die Detailanzeige der Kunden	30
Abb. 2.10	Der Konfigurations-Assistent für TableAdapter-Abfragen	31
Abb. 2.11	Der zweite Schritt des Assistenten	31
Abb. 2.12	Der dritte Schritt des Assistenten	32
Abb. 2.13	Die SELECT-Anweisung für das Beschaffen der Kundendaten	33
Abb. 2.14	Der vierte Schritt des Assistenten	33
Abb. 2.15	Die Methode FillBykundeDetail() im Tabellen-Designer (unten im Bereich KundeTableAdapter)	34
Abb. 2.16	Die Detailanzeige des Kunden im praktischen Einsatz	36

Abb. 2.17	Der zweite Schritt des Assistenten beim Anlegen einer UPDATE-Abfrage	37
Abb. 2.18	Der dritte Schritt	38
Abb. 2.19	Der Abfrage-Generator	38
Abb. 2.20	Die übernommene Tabelle	39
Abb. 2.21	Die übernommene Spalte	40
Abb. 2.22	Die vollständige SQL-Anweisung (im Feld in der Mitte)	41
Abb. 2.23	Die SQL-Anweisung im Assistenten für TableAdapter-Abfragen	42
Abb. 2.24	Der letzte Schritt	42
Abb. 2.25	Die neue UPDATE-Abfrage (unten im Bereich FahrzeugTableAdapter)	43
Abb. 2.26	Das gesetzte Kennzeichen ausgeliehen	44
Abb. 2.27	Der gesetzte Filter (unten rechts im Eigenschaftenfenster)	46
Abb. 2.28	Die Einstellungen für die ComboBox-Aufgaben	47
Abb. 2.29	Das Formular für die Rückgabe	48
Abb. 3.1	Das Formular für die Reservierungen	51
Abb. 4.1	Die Ausnahme bei einem gelöschten Kunden	55
Abb. 4.2	Die eingefügten Steuerelemente für die untergeordneten Tabellen (unten in der Mitte im Formular)	57
Abb. 4.3	Die Anzeige der untergeordneten Tabellen	58
Abb. 4.4	Die Steuerelemente für die TableAdapter (links oben in der Toolbox)	60
Abb. 4.5	Die Eigenschaft EnforceConstraints (unten rechts im Eigenschaftenfenster)	61
Abb. 6.1	Schema für Versionsnummern	71
Abb. 6.2	Die Einstellungen für die Versionsnummer	73
Abb. 6.3	Schnellschuss und geplante Wartung	74

E. Tabellenverzeichnis

Tab. 1.1	Die Spalten für die Tabelle mit den Vertragsdaten	7
Tab. 1.2	Die Spalten für die Tabelle mit den Reservierungen.....	9
Tab. 2.1	Die geänderten Eigenschaften für das Steuerelement kNummerComboBox	24

F. Codeverzeichnis

Code 2.1	Das Erzeugen eines neuen Datensatzes und das Setzen der Standardwerte	28
Code 2.2	Das Speichern der Änderungen und das Schließen des Formulars	28
Code 2.3	Die Methode zum Laden der Daten	35
Code 2.4	Die Anweisungen für die Anzeige des Detailformulars.....	35
Code 2.5	Das Ausführen der UPDATE-Abfrage	43
Code 2.6	Die Methode PreisBerechnen()	45
Code 2.7	Das Ausführen der UPDATE-Abfrage für die Rückgabe	49
Code 3.1	Die Methoden für die Verarbeitung einer Reservierung	53
Code 4.1	Die Reihenfolge zum Beschaffen der Daten	58

G. Medienverzeichnis

Video 1.1	Beziehungen zwischen Tabellen über den DataSet-Designer herstellen	16
Video 2.1	Daten aus einer verbundenen Tabelle über ein Kombinationsfeld anzeigen	26

H. Sachwortverzeichnis

#	
1:1-Beziehung	5
1:n-Beziehung	5
A	
Abfrage	30
Aggregatfunktion	44
B	
Beziehung	
Erstellen der	10
zwischen Tabellen	3
Bug fixing	67
D	
Datenintegrität	55
Detailanzeige	
für Kunden und Fahrzeuge	29
E	
Entwicklungszweig	72
F	
Formular	
für die Verträge	20
Freindschlüssel	4
I	
Integrität	
referenzielle	5
K	
Konfigurationsmanagement	70
Werkzeuge für das	74
N	
n:m-Beziehung	5
Normalisierung	4
P	
Primärschlüssel	4
R	
RCS	75
Realbetrieb	64
Redesign	67
Refactoring	67
Regressionstest	68
Release	72
Reservierung	50
Rückgabe	47
S	
Schnellschuss	74
SCM	70
Software	
Alterung von	65
Software Configuration Management	70
Software-Pflege	64
T	
Tabelle	
übergeordnete	12
untergeordnete	12
Titanic-Effekt	68
U	
Update	72
UPDATE-Abfrage	37
V	
Vermietung	37
Versionsnummer	71
Versionsverwaltung	71
W	
Wartung	64
adaptierende	66
geplante	73
korrektive	66
korrigierende	66
perfektionierende	66
präventive	66
progressive	66
wertsteigernde	66
Wartungsform	66

I. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH22D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Erstellen Sie für die Datenbankanwendung eine Suchfunktion, über die gezielt in der Datenbanktabelle **Kunde** nach Kunden mit einem bestimmten Nachnamen gesucht werden kann. Die Suchfunktion soll über eine Schaltfläche aus dem Formular **Form1** gestartet werden können. Danach soll zuerst ein Formular zur Eingabe des Suchkriteriums erscheinen. Die Ergebnisse der Suche sollen anschließend in einer DataGridView in einem weiteren Formular erscheinen. Wenn die Suche keine Treffer liefert, kann diese DataGridView auch leer bleiben.

Bitte sorgen Sie bei der Lösung dafür, dass die Daten nur angezeigt werden können. Änderungen an den Daten und eine Navigation durch die Datensätze sollen nicht möglich sein.

Schicken Sie bitte das vollständige Projekt mit allen Unterordnern und Dateien ein. Um Übertragungszeit und -kosten zu sparen, packen Sie bitte das Projekt mit einem geeigneten Programm – zum Beispiel mit WinZip oder direkt über Windows.

Beschreiben Sie außerdem, welche grundsätzlichen Schritte für die Erweiterungen erforderlich sind – also zum Beispiel, welche Steuerelemente Sie einfügen und wie Sie die Eigenschaften dieser Elemente setzen.

100 Pkt.

Datenabfragen mit LINQ

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1219N01

CSHP23D

Datenabfragen mit LINQ

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Datenabfragen mit LINQ

Inhaltsverzeichnis

Einleitung	1
1 Was ist LINQ?	3
Zusammenfassung	6
2 Die ersten Schritte mit LINQ	7
2.1 Der grundsätzliche Ablauf einer Abfrage	7
2.2 Der Aufbau einer Abfrage	10
2.3 Weitere Abfrageoperatoren	14
2.4 Ein komplexeres Beispiel	20
2.5 Exkurs: Generics	24
Zusammenfassung	28
3 LINQ to XML	29
3.1 Daten lesen	29
3.2 XML-Elemente erzeugen	33
3.3 Daten schreiben	35
3.4 Ein komplexeres Beispiel	35
Zusammenfassung	38
4 Eine kleine Datenbankanwendung	39
4.1 Vorüberlegungen und Vorbereitungen	39
4.2 Erstellen der Datenbank	40
4.3 Installation der Tools	41
4.4 Entitätsklasse für den Zugriff erstellen	43
4.5 Das Programm	45
Zusammenfassung	50
Schlussbetrachtung	52

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	53
B.	Glossar	56
C.	Literaturverzeichnis	58
D.	Abbildungsverzeichnis	59
E.	Tabellenverzeichnis	60
F.	Codeverzeichnis	61
G.	Sachwortverzeichnis	62
H.	Einsendeaufgabe	63

Einleitung

In diesem Studienheft beschäftigen wir uns mit LINQ (*Language Integrated Query*¹). Dabei handelt es sich um eine universelle Abfragesprache, die fester Bestandteil des .NET Frameworks und damit auch von C# ist. Mit LINQ können Sie sich mit einer einheitlichen Syntax Daten aus unterschiedlichen Datenquellen wie Arrays, Aufzählungen, XML-Dateien und SQL-Datenbanken beschaffen.

In diesem Studienheft lernen Sie:

- was LINQ ist,
- wie LINQ-Abfragen aufgebaut sind,
- was die Abfrage- und die Methodensyntax in LINQ unterscheidet,
- was Lambda-Ausdrücke sind,
- welche Abfrageoperatoren LINQ kennt,
- was sich hinter Generics verbirgt,
- wie Sie Daten aus einer XML-Datei mit LINQ to XML beschaffen,
- wie Sie mit LINQ to XML XML-Elemente erzeugen,
- wie Sie mit LINQ to XML XML-Dateien erzeugen,
- wie Sie die Anzeige in einem DataGridView an eine XML-Datei binden,
- wie Sie mit LINQ to SQL Daten aus einer Datenbank beschaffen,
- wie Sie Entitätsklassen für den Zugriff erzeugen,
- was sich hinter dem Datenkontext verbirgt und
- wie Sie Daten aus einer Datenbanktabelle in einer WPF-Anwendung in einem DataGridView anzeigen und bearbeiten können.

Sehen wir uns zuerst an, was sich überhaupt hinter LINQ verbirgt.

Christoph Siebeck

1. Übersetzt bedeutet *Language Integrated Query* so viel wie „in die Sprache integrierte Abfrage“.

1 Was ist LINQ?

In diesem Kapitel stellen wir Ihnen LINQ im Überblick vor.

LINQ ist eine Abfragesprache, die Daten aus verschiedenen Quellen verarbeiten kann. Dazu gehören Arrays, Auflistungen, XML-Dateien, aber auch Datenbanktabellen des SQL Servers und DataSets von ADO.NET.

LINQ lässt sich bei jedem Objekt einsetzen, das die Schnittstelle `IEnumerable<T>` oder `IEnumerable` unterstützt.



Was sich genau hinter diesen Schnittstellen verbirgt, erfahren Sie gleich.

LINQ ist fest in das .NET Framework integriert. Sie können daher Abfragen formulieren, die sich eng an die gewohnte C#-Syntax anlehnen. Außerdem kann der Code wie gewohnt bei der Übersetzung geprüft werden. Sie sehen damit also direkt bei der Übersetzung, ob die Abfrage einen Syntaxfehler enthält. Bei den Abfragetechniken, die wir bisher bei Datenbanktabellen eingesetzt haben, würden die Fehler erst beim Zugriff auf die Datenbank – also zur Laufzeit – auftreten.

Der Zugriff auf die Daten wird über LINQ-Provider – auch LINQ-Anbieter genannt – ermöglicht. Zu diesen LINQ-Providern gehören zum Beispiel:

- LINQ to SQL für den Zugriff auf SQL Server-Datenbanken,
- LINQ to XML für den Zugriff auf XML-Dateien,
- LINQ to Objects für den direkten Zugriff auf Objekte wie Arrays oder Auflistungen,
- LINQ to DataSet für den Zugriff auf DataSets aus dem ADO.NET-Framework oder
- LINQ to Entities für den Zugriff auf relationale Datenstrukturen aus dem ADO.NET-Framework.

Zusätzlich können auch LINQ-Provider von anderen Anbietern eingesetzt werden, oder bei Bedarf eigene Provider erstellt werden.

Die Syntax für den Zugriff auf die Daten ist in allen Fällen gleich. Es spielt also grundsätzlich keine Rolle, ob Sie mit einem Code auf eine XML-Datei oder eine SQL Server-Datenbank zugreifen.

Schauen wir uns zum Einstieg den Einsatz von LINQ an zwei einfachen Beispielen an. Das erste Beispiel soll bestimmte Zeichen in einer Zeichenkette ermitteln. Das zweite Beispiel soll in einer Liste alle Zahlen ermitteln, die sich glatt durch 3 teilen lassen.

Der erste Code sieht so aus:

```
using System;
using System.Linq;

namespace Cshp23d_01_01
{
    class Program
    {
```

```

static void Main(string[] args)
{
    //bitte in einer Zeile eingeben
    string zeichenkette = "Ich bin ein Beispieltext, der in
    einer LINQ-Abfrage benutzt wird";
    char suche;

    suche = 'e';

    //die Abfrage
    var abfrage =
        from zeichen in zeichenkette
        where zeichen == suche
        select zeichen;

    //die Ausgabe
    foreach (char ausgabe in abfrage)
        Console.Write(ausgabe + " ");
    Console.WriteLine();

    //die Vorkommen zählen
    int anzahl = abfrage.Count();
    Console.WriteLine("Anzahl für {0} = {1}", suche, anzahl);
}
}
}

```

Code 1.1: Ein einfaches Beispiel

Sehen wir uns die einzelnen Zeilen der Reihe nach an.

Zunächst vereinbaren wir eine Zeichenkette und eine Variable vom Typ `char` für die Suche. Dieser Variablen weisen wir den Wert „e“ zu.

Mit der Anweisung

```

var abfrage =
    from zeichen in zeichenkette
    where zeichen == suche
    select zeichen;

```

formulieren wir eine LINQ-Abfrage. Sie sucht in der Zeichenkette `zeichenkette` nach allen Zeichen, die unserem Suchbegriff entsprechen und wählt sie aus. Gespeichert wird das Ergebnis in der Variablen `abfrage`, die wir über das Schlüsselwort `var` vereinbaren.

Hinweis:

Bei der Abfrage handelt es sich um **eine** Anweisung, die zur besseren Übersicht auf mehrere Zeilen verteilt wird. Sie dürfen daher auch nur an das Ende der letzten Zeile ein Semikolon setzen.

Zur Erinnerung:

Über das Schlüsselwort `var` wird die Typinferenz oder Typableitung in C# umgesetzt. Der Typ einer Variablen wird dann durch die erste Zuweisung automatisch ermittelt.

**Hinweis:**

Was sich genau hinter den einzelnen Teilen der Abfrage verbirgt, erklären wir Ihnen gleich im Detail.

Mit der Schleife

```
foreach (char ausgabe in abfrage)
    Console.WriteLine(ausgabe + " ");
```

geben wir dann alle Werte, die in der Variablen `abfrage` gespeichert sind, aus. Hier werden also alle Treffer ausgegeben.

Die beiden Anweisungen

```
int anzahl = abfrage.Count();
Console.WriteLine("Anzahl für {0} = {1}",suche, anzahl);
```

zählen die Anzahl der Treffer und geben sie aus. Dazu benutzen wir die Methode `Count()` für die Variable `abfrage`. Sie liefert uns die Anzahl der Elemente.

Hinweis:

Die Variable `anzahl` ist in unserem Beispiel vom Typ `IEnumerable<T>`.

So viel zum ersten Beispiel.

Der Code für das zweite Beispiel sieht so aus:

```
using System;
using System.Linq;

namespace Cshp23d_01_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] zahlen = {4, 99, 13, 3, 9, 12, 77, 43};

            //die Abfrage
            var abfrage =
                from zahl in zahlen
                where zahl % 3 == 0
                orderby zahl
                select zahl;
```

```
//die Ausgabe  
foreach (int wert in abfrage)  
    Console.WriteLine(wert + " ");  
}  
}  
}
```

Code 1.2: Ein weiteres Beispiel

Sehen wir uns auch hier die Anweisungen der Reihe nach.

Zunächst vereinbaren wir ein Array mit einigen Zahlen.

Danach erfolgt die Abfrage:

```
var abfrage =  
    from zahl in zahlen  
    where zahl % 3 == 0  
    orderby zahl  
    select zahl;
```

Sie wählt in diesem Beispiel alle Werte aus `zahlen` aus, die sich glatt durch 3 teilen lassen und legt sie wieder in der Variablen `abfrage` ab. Zusätzlich lassen wir die Daten aufsteigend sortieren. Das erledigt die Zeile

```
orderby zahl
```

Zusammenfassung

LINQ ist eine universelle Abfragesprache. Sie ist fest in das .NET Framework integriert.

Zu LINQ gehören verschiedene Provider, die den Zugriff auf Daten ermöglichen.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Für welche Objekte lässt sich LINQ einsetzen?
- 1.2 Nennen Sie mindestens drei LINQ-Provider.
- 1.3 Sie wollen sich mit LINQ Daten aus einer XML-Datei und einer Datenbank beschaffen. Wie unterscheidet sich die Syntax der Abfrage?

2 Die ersten Schritte mit LINQ

In diesem Kapitel stellen wir Ihnen den grundsätzlichen Ablauf und den grundlegenden Aufbau von LINQ-Abfragen vor. Außerdem zeigen wir Ihnen an einigen etwas komplexeren Beispielen, welche Möglichkeiten LINQ bietet. Zum Abschluss unternehmen wir noch einen kurzen Ausflug in die Welt der Generics.

2.1 Der grundsätzliche Ablauf einer Abfrage

Eine LINQ-Abfrage besteht immer aus drei verschiedenen Teilen:

1. das Erstellen einer Datenquelle,
2. dem Erstellen der Abfrage und
3. dem Ausführen der Abfrage.

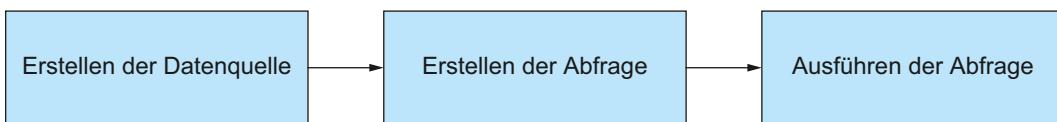


Abb. 2.1: Die drei Teile einer LINQ-Abfrage

Schauen wir uns diese drei Bestandteile noch einmal an den beiden Codes aus dem vorigen Kapitel an. Beginnen wir mit Code 1.1.

Das Erstellen der Datenquelle erfolgt hier durch die Vereinbarung der Zeichenkette in der Zeile

```
string zeichenkette = "Ich bin ein Beispieltext, der in einer
LINQ-Abfrage benutzt wird";
```

Die Abfrage wird dann mit der folgenden Anweisung erstellt:

```
var abfrage =
    from zeichen in zeichenkette
    where zeichen == suche
    select zeichen;
```

Hier werden – wie Sie ja bereits wissen – alle Zeichen in der Zeichenkette in der Variablen `abfrage` abgelegt, die dem Suchkriterium `zeichen` entsprechen.

Ausgeführt wird die Abfrage mit der Schleife

```
foreach (char ausgabe in abfrage)
    Console.Write(ausgabe + " ");
Console.WriteLine();
```

oder mit dem Zugriff über die Methode `Count()`.

Die drei Bestandteile finden sich auch im Code 1.2 wieder.

Der Zugriff erfolgt mit dem Erstellen des Arrays in der Zeile

```
int[] zahlen = {4, 99, 13, 3, 9, 12, 77, 43};
```

Erstellt wird die Abfrage mit der Anweisung:

```
var abfrage =  
    from zahl in zahlen  
    where zahl % 3 == 0  
    orderby zahl  
    select zahl;
```

Ausgeführt wird die Abfrage dann mit der Schleife:

```
foreach (int wert in abfrage)  
    Console.WriteLine(wert + " ");
```

Bitte beachten Sie, dass die Abfragen nicht direkt beim Erstellen ausgeführt werden, sondern erst verzögert – zum Beispiel beim Anzeigen der Elemente.

Diese verzögerte Ausführung können Sie ganz einfach ausprobieren, indem Sie im Code 1.2 nach dem Ausführen der ersten Abfrage einen Wert im Array ändern. Der geänderte Code könnte zum Beispiel so aussehen:

```
using System;  
using System.Linq;  
  
namespace Cshp23d_02_01  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[] zahlen = { 4, 99, 13, 3, 9, 12, 77, 43 };  
  
            //die Abfrage  
            var abfrage =  
                from zahl in zahlen  
                where zahl % 3 == 0  
                orderby zahl  
                select zahl;  
  
            //die Ausgabe  
            foreach (int wert in abfrage)  
                Console.WriteLine(wert + " ");  
  
            //die Vorkommen zählen  
            int anzahl = abfrage.Count();  
            Console.WriteLine("Es gibt {0} Treffer", anzahl);  
  
            //den ersten Wert ändern  
            zahlen[0] = 12;  
  
            //ausgeben und zählen  
            foreach (int wert in abfrage)  
                Console.WriteLine(wert + " ");  
        }  
    }  
}
```

```
        anzahl = abfrage.Count();
        Console.WriteLine("Es gibt {0} Treffer", anzahl);
    }
}
```

Code 2.1: Verzögerte Ausführung der Abfrage

Die ersten Ausgaben liefern die Zahlen 3, 9, 12, 99 und vier Treffer. Die zweiten Ausgaben nach dem Ändern des ersten Wertes im Array dagegen liefern die Werte 3, 9, 12, 12 und 99 und fünf Treffer.

Abfragen werden bei LINQ verzögert ausgeführt und nicht beim Erstellen der Abfrage.



Kommen wir noch einmal kurz zur Datenquelle:

Wie Sie ja bereits wissen, lassen sich als Datenquelle bei LINQ grundsätzlich alle Quellen verwenden, die die Schnittstelle `IEnumerable` oder die Schnittstelle `IEnumerable<T>` implementieren oder eine Schnittstelle, die von diesen Schnittstellen abgeleitet ist. Dazu gehören zum Beispiel Zeichenketten, Arrays und auch Aufzählungen. Für andere Datenquellen, wie zum Beispiel XML-Dateien oder Datenbanktabellen, müssen die Informationen durch den LINQ-Provider in das passende Format umgesetzt werden.

Einfache Typen wie zum Beispiel `int` lassen sich nicht als Datenquelle verwenden. Allerdings macht eine Abfrage bei solchen Typen ja auch nur wenig Sinn, da immer nur ein Wert gespeichert werden kann.

Der folgende Code lässt sich also nicht übersetzen:

```
using System;
using System.Linq;

namespace Cshp23d_02_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl = 10;

            var abfrage = from wert in zahl
                          where wert == 10
                          select wert;

            foreach (int wert in abfrage)
                Console.WriteLine("Der Wert ist {0}", wert);
        }
    }
}
```

Code 2.2: Eine ungültige Datenquelle (der Code lässt sich nicht übersetzen)

Beim Übersetzen erscheint die etwas verwirrende Fehlermeldung, dass kein Abfragemuster für den Quelltyp `int` und für `Where` gefunden wurde. Sie besagt aber eigentlich nur, dass Sie ein einzelnes Objekt als Datenquelle verwenden wollen und keine Sammlung von Objekten.

2.2 Der Aufbau einer Abfrage

Nehmen wir jetzt einmal den Aufbau einer Abfrage ein wenig genauer unter die Lupe. Dazu nehmen wir das Beispiel aus dem Code 1.1:

```
var abfrage =
    from zeichen in zeichenkette
    where zeichen == suche
    select zeichen;
```

Eingeleitet wird die Abfrage durch die Vereinbarung der Variablen, die das Ergebnis aufnimmt – in unserem Beispiel also die Variable `abfrage`. Den Typ lassen wir über das Schlüsselwort `var` durch die erste Zuweisung bestimmen.

Hinter dem Zuweisungsoperator folgen das Schlüsselwort `from`, die Bereichsvariable, das Schlüsselwort `in` und die Datenquelle.

Das Schlüsselwort `from` legt dabei fest, woher die Daten stammen sollen. Die Bereichsvariable speichert beim Ausführen der Abfrage jedes einzelne Element, das die Abfrage liefert. Sie lässt sich mit der Variablen einer `foreach`-Schleife vergleichen, die beim Ausführen der Schleife jedes Element aus der Aufzählung speichert.

Einen Typ müssen Sie für die Bereichsvariable nicht angeben. Er wird aus der Datenquelle abgeleitet. Die Angabe ist aber auch nicht verboten. Möglich ist also die folgende Konstruktion:

```
from int zahl in zahlen
```

Allerdings muss die Datenquelle dann auch zum Typ der Bereichsvariablen passen. In unserem Fall muss es sich bei `zahlen` also um eine Datenquelle vom Typ `int` handeln. Andernfalls wird beim Ausführen eine Ausnahme ausgelöst, da die Konvertierung nicht möglich ist.

Über das Schlüsselwort `where` filtern Sie die Ergebnisse der Abfrage. Dabei sind alle Ausdrücke erlaubt, die einen booleschen Wert als Ergebnis liefern. In unserem Beispiel überprüfen wir zum Beispiel, ob der Wert dem Wert einer Variablen `suche` entspricht. Das übernimmt der Ausdruck

```
where zeichen == suche
```

Denkbar sind aber auch andere Vergleiche und auch Verknüpfungen von logischen Ausdrücken. Die folgende Abfrage liefert zum Beispiel alle ungeraden Zahlen aus der Datenquelle, die größer sind als 100.

```
var abfrage =
    from zahl in zahlen
    where zahl > 100 && zahl % 2 != 0
    select zahl;
```

Auch der Zugriff auf Eigenschaften der Bereichsvariablen ist möglich. Die folgende Abfrage liefert zum Beispiel alle Wörter in einer Datenquelle vom Typ `string`, die länger als fünf Zeichen sind.

```
var abfrage =
    from wort in liste
    where wort.Length > 5
    select wort;
```

Es lassen sich auch Methoden für den Vergleich verwenden. Im folgenden Beispiel liefert die Abfrage über die Methode für den Vergleich entweder alle ungeraden Zahlen, die größer sind als 100, oder alle ungeraden Zahlen, die kleiner sind als 100.

```
using System;
using System.Linq;

namespace Cshp23d_02_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] zahlen = {14, 101, 213, 33, 99, 121, 177, 43, 112};

            //die Abfrage
            var abfrage =
                from zahl in zahlen
                where MeinVergleich(zahl, false) == true
                select zahl;

            //die Ausgabe
            foreach (int wert in abfrage)
                Console.Write(wert + " ");
        }

        static bool MeinVergleich(int wert, bool was)
        {
            if (was == true)
                return (wert > 100 && wert % 2 != 0);
            else
                return (wert < 100 && wert % 2 != 0);
        }
    }
}
```

Code 2.3: Einsatz einer Methode in einem Filter

Über das Schlüsselwort `select`² legen Sie fest, welchen Typ die Werte haben, die die Abfrage liefert. Im einfachsten Fall geben Sie hier die Bereichsvariable an – so wie in unseren bisherigen Beispielen.

2. `Select` lässt sich mit „wähle aus“ übersetzen.

Sie können aber auch den Wert der Bereichsvariablen verändern. Die folgende Abfrage addiert immer 1000 auf den Wert der Bereichsvariablen:

```
var abfrage =
    from zahl in zahlen
    where zahl < 1000
    select (zahl + 1000);
```

Sie können auch komplett neue Werte erzeugen. Der folgende Code liefert zum Beispiel bei einem Treffer nicht mehr den Wert, sondern nur noch das Wort „Treffer“.

```
using System;
using System.Linq;
using System.Text;

namespace Cshp23d_02_04
{
    class Program
    {
        static void Main(string[] args)
        {

            //bitte in einer Zeile eingeben
            int[] zahlen = {14, 101, 213, 33, 99, 121, 177, 43, 112};
            //die Abfrage
            var abfrage =
                from zahl in zahlen
                where zahl < 100
                select (new StringBuilder("Treffer"));

            //die Ausgabe
            foreach (StringBuilder wert in abfrage)
                Console.Write(wert + " ");
        }
    }
}
```

Code 2.4: Geänderter Typ

Interessant ist hier vor allem die Abfrage

```
var abfrage =
    from zahl in zahlen
    where zahl < 100
    select (new StringBuilder("Treffer"));
```

Der `select`-Ausdruck erzeugt hier über `new StringBuilder("Treffer")` eine neue Zeichenkette mit dem Wert „Treffer“. Dadurch ändert sich auch der Typ, den die Abfrage liefert. Sie enthält jetzt keine Zahlen mehr, sondern Zeichenketten vom Typ `StringBuilder`. Daher müssen Sie auch die Schleife für die Ausgabe entsprechend anpassen. Sie muss jetzt so aussehen:

```
foreach (StringBuilder wert in abfrage)
    Console.Write(wert + " ");
```

Hinweis:

Denken Sie bitte an das Einbinden des Namensraums `System.Text`. Sonst lässt sich der Code nicht übersetzen.

In vielen Fällen müssen Sie die Ergebnisse einer Abfrage auch sortieren. Dazu verwenden Sie das Schlüsselwort `orderby`³. Wir haben es ja bereits im Code 1.2 eingesetzt. Die Abfrage sah dort so aus:

```
var abfrage =
    from zahl in zahlen
    where zahl % 3 == 0
    orderby zahl
    select zahl;
```

Im Standard werden die Werte aufsteigend sortiert. Für eine absteigende Sortierung ergänzen Sie noch das Schlüsselwort `descending`⁴. Die folgende Abfrage liefert die Werte entsprechend in absteigender Reihenfolge.

```
var abfrage =
    from zahl in zahlen
    where zahl % 3 == 0
    orderby zahl descending
    select zahl;
```

Abfrage- und Methodensyntax

Bisher haben wir in unseren Beispielen immer die Abfragesyntax verwendet. Abfragen lassen sich bei LINQ aber auch in der Methodensyntax formulieren. Sie ist ein wenig kompakter, aber auch nicht ganz so einfach nachzuvollziehen.

Die Abfrage

```
var abfrage =
    from zeichen in zeichenkette
    where zeichen == suche
    select zeichen;
```

würde in Methodensyntax zum Beispiel so aussehen:

```
var abfrage = zeichenkette
    .Where(zeichen => zeichen == suche);
```

Hier liefert die Methode `Where()` über den Ausdruck `zeichen => zeichen == suche` alle Zeichen, die dem Suchkriterium `suche` entsprechen.

Bei dem Ausdruck `zeichen => zeichen == suche` handelt es sich um einen Lambda-Ausdruck. Das Zeichen `=>` ist dabei der Lambda-Operator. Er lässt sich als „wird zu“ lesen.

3. *Order by* lässt sich mit „sortiere nach“ übersetzen.

4. *Descending* bedeutet übersetzt so viel wie „absteigend“.

Der Ausdruck

```
var abfrage =
    from zahl in zahlen
    where zahl % 3 == 0
    orderby zahl
    select zahl;
```

sieht in Methodensyntax so aus:

```
var abfrage = zahlen
    .Where(zahl => zahl % 3 == 0)
    .OrderBy(n => n);
```

Ob Sie die Abfrage- oder die Methodensyntax verwenden, spielt in der Regel keine Rolle. Das Ergebnis ist identisch. Wir werden in diesem Studienheft der einfachen Lesbarkeit halber aber vor allem die Abfragesyntax verwenden.

2.3 Weitere Abfrageoperatoren

Eine weitere interessante Möglichkeit für Abfragen besteht im Gruppieren von Daten. Damit können Sie Daten anhand von Kriterien in Gruppen zusammenfassen. Schauen wir uns dazu ein Beispiel an.

Wir erstellen eine Liste mit Autos, die über die Eigenschaften Bezeichnung, Farbe und Typ verfügen. Für diese Liste erstellen wir eine Abfrage, die alle Autos einer bestimmten Farbe liefert und dabei die Anzeige nach dem Typ gruppiert. Die einzelnen Autos bilden wir dabei über Instanzen einer Klasse `Auto` ab. Sie verfügt lediglich über die drei Eigenschaften. Der Code für die Klasse sieht so aus:

```
namespace Autoliste
{
    public class Auto
    {
        //die Eigenschaften
        public string Bezeichnung {get; set;}
        public string Farbe {get; set;}
        public string Typ {get; set;}
    }
}
```

Code 2.5: Eine sehr einfache Klasse für Autos

Vereinbart werden hier drei Eigenschaften vom Typ `string`. Sie können gelesen und geschrieben werden.

Für die Klasse erstellen wir einige anonyme Instanzen und legen sie in einem Array ab. Dieses Array dient dann als Datenquelle für die LINQ-Abfragen. Der entsprechende Code sieht so aus:

```
using System;
using System.Linq;
```

```

namespace Autoliste
{
    class Program
    {
        static void Main(string[] args)
        {
            //ein Array mit Autos erzeugen
            Auto[] autos =
            {
                //bitte jeweils in einer Zeile eingeben
                new Auto {Bezeichnung = "V70", Farbe = "Rot",
                Typ = "Kombi"},

                new Auto {Bezeichnung = "V90", Farbe = "Schwarz",
                Typ = "Kombi"},

                new Auto {Bezeichnung = "Golf", Farbe = "Rot",
                Typ = "Limousine"},

                new Auto {Bezeichnung = "Golf 2", Farbe = "Schwarz",
                Typ = "Limousine"},

                new Auto {Bezeichnung = "Goggomobil", Farbe = "Rot",
                Typ = "Limousine"},

            };
        }

        //alle schwarzen Autos filtern
        var abfrageSchwarz = from meinAuto in autos
            where meinAuto.Farbe == "Schwarz"
            select meinAuto;

        foreach (Auto wagen in abfrageSchwarz)
            Console.WriteLine(wagen.Bezeichnung);

        //und gruppieren
        var abfrageRotGruppe = from meinAuto in autos
            where meinAuto.Farbe == "Rot"
            group meinAuto by meinAuto.Typ;

        Console.WriteLine();

        foreach (var autoGruppe in abfrageRotGruppe)
        {
            Console.WriteLine();
            Console.WriteLine(autoGruppe.Key);
            foreach (var meinAuto in autoGruppe)
            {
                //bitte in einer Zeile eingeben
                Console.WriteLine("{0} {1}", meinAuto.Bezeichnung,
                meinAuto.Farbe);
            }
        }
    }
}

```

Code 2.6: Der Zugriff auf die Autoliste

Schauen wir uns auch hier die wichtigsten Anweisungen der Reihe nach an.

Mit der Anweisung

```
Auto[] autos =
{
    new Auto {Bezeichnung = "V70", Farbe = "Rot", Typ = "Kombi"},
    new Auto {Bezeichnung = "V90", Farbe = "Schwarz",
    Typ = "Kombi"},
    new Auto {Bezeichnung = "Golf", Farbe = "Rot",
    Typ = "Limousine"},
    new Auto {Bezeichnung = "Glf 2", Farbe = "Schwarz",
    Typ = "Limousine"},
    new Auto {Bezeichnung = "Ggomobil", Farbe = "Rot",
    Typ = "Limousine"},
};
```

erzeugen wir ein Array `autos` vom Typ `Auto`. Das Array enthält danach fünf Elemente.

Für das Erzeugen der einzelnen Elemente benutzen wir dabei einen Objektinitialisierer. Hier können Sie den einzelnen Eigenschaften direkt Werte zuweisen. Die Zuweisungen setzen Sie dabei in geschweifte Klammern und trennen sie jeweils durch ein Komma.

Die Anweisung

```
new Auto {Bezeichnung = "Ggomobil", Farbe = "Rot",
Typ = "Limousine"}
```

entspricht dabei dem Aufruf eines Konstruktors

```
public Auto(string bezeichnung, string farbe, string typ)
```

mit der Anweisung

```
new Auto("Ggomobil", "Rot", "Limousine"),
```



Über einen Objektinitialisierer können Sie Instanzen einer Klasse erzeugen und direkt die Werte von Eigenschaften setzen. Ein Objektinitialisierer entspricht in der Wirkung einem Konstruktor.

Mit den Anweisungen

```
var abfrageSchwarz = from meinAuto in autos
    where meinAuto.Farbe == "Schwarz"
    select meinAuto;

foreach (Auto wagen in abfrageSchwarz)
    Console.WriteLine(wagen.Bezeichnung);
```

filtern wir alle Autos mit der Farbe Schwarz und lassen sie ausgeben. Dabei gibt es keine Besonderheiten.

Interessant ist dann wieder die Anweisung

```
var abfrageRotGruppe = from meinAuto in autos
    where meinAuto.Farbe == "Rot"
    group meinAuto by meinAuto.Typ;
```

Hier filtern wir alle roten Autos und gruppieren sie nach dem Typ. Das Gruppieren übernimmt dabei der Ausdruck `group meinAuto by meinAuto.Typ`.

Da dieser Ausdruck auch den Typ der Abfrage bestimmt, brauchen wir keinen eigenen `select`-Ausdruck mehr.

Die gruppierte Ausgabe erfolgt über eine geschachtelte `foreach`-Schleife.

```
foreach (var autoGruppe in abfrageRotGruppe)
{
    Console.WriteLine();
    Console.WriteLine(autoGruppe.Key);
    foreach (var meinAuto in autoGruppe)
    {
        Console.WriteLine("{0} {1}", meinAuto.Bezeichnung,
            meinAuto.Farbe);
    }
}
```

Sie durchläuft zunächst mit der Schleife

```
foreach (var autoGruppe in abfrageRotGruppe)
```

alle Einträge in der Gruppe – also alle unterschiedlichen Einträge für den Typ. Auf den Wert, nach dem gruppiert wurde, können Sie dabei über die Eigenschaft `Key` zugreifen. In unserem Beispiel enthält die Eigenschaft den jeweiligen Namen des Typs – also zum Beispiel „Limousine“.

Hinweis:

Der Typ der Schleifenvariablen der äußeren Schleife ist `IGrouping<string, Auto>`. Dadurch wird eine Gruppe mit Informationen vom Typ `string` und Elementen vom Typ `Auto` abgebildet.

Mit der inneren Schleife

```
foreach (var meinAuto in autoGruppe)
{
    Console.WriteLine("{0} {1}", meinAuto.Bezeichnung,
        meinAuto.Farbe);
}
```

werden dann alle Elemente der jeweiligen Gruppe durchlaufen. Für jedes Element werden dabei die Bezeichnung und die Farbe ausgegeben.

Die Ausgaben sehen so aus:

```
V90
Golf 2
```

```
Kombi
V70 Rot
```

```
Limousine
Golf Rot
Goggomobil Rot
```

Zuerst werden alle schwarzen Autos ausgegeben, danach gruppiert alle roten Autos.

Hinweis:

Das komplette Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Autoliste**.

Neben den Abfrageoperatoren, die Sie direkt in einer Abfrage verwenden, gibt es auch noch Abfrageoperatoren, die Sie über Methoden für die Variable der Abfrage aufrufen. Dazu gehören zum Beispiel `Count()`, `Distinct()`⁵, `Sum()`, `Average()`⁶, `Min()` und `Max()`.

Die Methode `Count()` liefert Ihnen die Anzahl der Elemente in einer Menge. Diese Methode haben Sie ja bereits mehrfach eingesetzt.

Die Methode `Distinct()` liefert Ihnen aus einer Menge eine neue Menge mit allen unterschiedlichen Elementen. Das heißt: Identische Elemente werden nicht berücksichtigt.

Die Methode `Sum()` berechnet die Summe der Werte in der Menge. Die Methode `Average()` liefert den Durchschnitt. Die Methoden `Min()` und `Max()` schließlich beschaffen Ihnen den kleinsten beziehungsweise größten Wert in der Menge. Alle vier Methoden liefern einen einzelnen Wert zurück.

Im praktischen Einsatz finden Sie die Operatoren im folgenden Code. Er erzeugt eine Liste mit 10 zufälligen Zahlen und beschafft über eine LINQ-Abfrage alle geraden Zahlen. Danach werden verschiedene Operatoren auf die Abfrage eingesetzt.

```
using System;
using System.Linq;

namespace Cshp23d_02_07
{
    class Program
    {
        static void Main(string[] args)
        {
            //ein Array für 10 Zahlen
            int[] zahlen = new int[10];
            //für das zufällige Erzeugen der Werte
            Random zufall = new Random();

            //das Array mit zufälligen Werten zwischen 1 und 10 füllen
            for (int i = 0; i < zahlen.Length; i++)
                zahlen[i] = zufall.Next(1, 11);

            //zum Test ausgeben
            foreach (int zahl in zahlen)
                Console.WriteLine(zahl);
            Console.WriteLine();
        }
    }
}
```

5. *Distinct* bedeutet übersetzt „deutlich“ oder „eindeutig“.

6. *Average* lässt sich mit „Durchschnitt“ übersetzen.

```
//alle geraden Zahlen beschaffen
var abfrage = from zahl in zahlen
    where zahl % 2 == 0
    orderby zahl
    select zahl;

//und ausgeben
foreach (int zahl in abfrage)
    Console.WriteLine(zahl);
Console.WriteLine();

//eine neue Abfragemenge ohne doppelte Einträge erstellen
var neueAbfrage = abfrage.Distinct();
foreach (int zahl in neueAbfrage)
    Console.WriteLine(zahl);
Console.WriteLine();

//die Anzahl, Summe, Durchschnitt, Minimum und
//Maximum beschaffen
int anzahl = neueAbfrage.Count();
int summe = neueAbfrage.Sum();
double durchschnitt = neueAbfrage.Average();
int minimum = neueAbfrage.Min();
int maximum = neueAbfrage.Max();
//und ausgeben
Console.WriteLine("Die Menge enthält {0} Werte.", anzahl);
//bitte jeweils in einer Zeile eingeben
Console.WriteLine("Die Summe der Werte beträgt {0}. Der Durchschnitt ist {1}", summe, durchschnitt);
Console.WriteLine("Das Minimum ist {0} und das Maximum {1}.", minimum, maximum);
}
}
```

Code 2.7: Weitere Abfrageoperatoren im Einsatz

Bitte beachten Sie, dass Sie Ihnen die Methode `Distinct()` eine neue Menge ohne Duplikate liefert. Die eigentliche Menge wird nicht verändert. Daher weisen wir das Ergebnis der Methode mit der Anweisung

```
var neueAbfrage = abfrage.Distinct();
```

einer neuen Menge zu.

Hinweis:

Viele Abfrageoperatoren, die über Methoden umgesetzt werden, lassen sich für den Typ `IEnumerable` beziehungsweise `IEnumerable<T>` verwenden. Im Code 2.7 könnten Sie die Operatoren daher auch direkt auf das Array anwenden. Der Ausdruck `zahlen.Max()` würde Ihnen also die größte Zahl aus dem Array `zahlen` liefern und der Ausdruck `zahlen.Sum()` die Summe aller Werte im Array `zahlen`.

2.4 Ein komplexeres Beispiel

Schauen wir uns den Einsatz von LINQ jetzt an einem etwas komplexeren Beispiel an. Wir erstellen ein kleines Programm für Textanalysen. Es soll eine Textdatei einlesen und dann auf Wunsch folgende Analysen durchführen:

- Anzahl der kompletten Sätze im Text,
- Anzahl der Nebensätze im Text,
- Anzahl der Wörter im Text,
- durchschnittliche Satzlänge in Wörtern,
- durchschnittliche Wortlänge,
- Anzahl aller Wörter im Text mit mehr als 10 Zeichen.

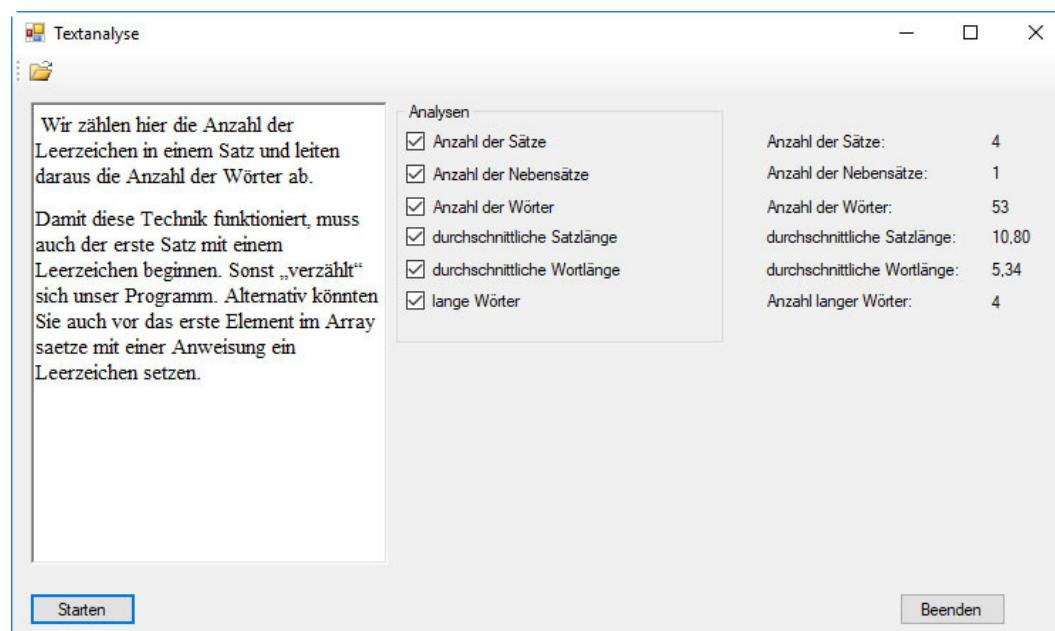


Abb. 2.2: Das Programm im Einsatz

Hinweis:

Das komplette Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **Textanalyse**.

Damit der Aufwand nicht zu groß wird, ermitteln wir die Anzahl der kompletten Sätze über die Anzahl der Punkte im Text und die Anzahl der Nebensätze über die Anzahl der Kommas. Das ist zwar nicht ganz korrekt, erleichtert uns die Arbeit aber erheblich.

Außerdem gehen wir davon aus, dass jeder einzelne Satz mit einem Leerzeichen beginnt. Das gilt auch für den ersten Satz im Text – auch wenn hier eigentlich kein Leerzeichen zu Beginn stehen muss.

Legen Sie eine neue Windows Forms-Anwendung an. Setzen Sie oben in das Formular eine Symbolleiste mit einem Öffnensymbol. Links fügen Sie eine RichTextBox ein. In die Mitte setzen Sie eine Gruppe mit Kontrollkästchen für die Auswahl der gewünschten

Analyse. Eins der Kontrollkästchen markieren Sie als Standardauswahl. Nach rechts kommen Labels für die Anzeige der Informationen. Unten fügen Sie zwei Schaltflächen ein. Eine dient zum Starten der Analyse und die andere zum Beenden der Anwendung.

Die Oberfläche könnte zum Beispiel so aussehen:

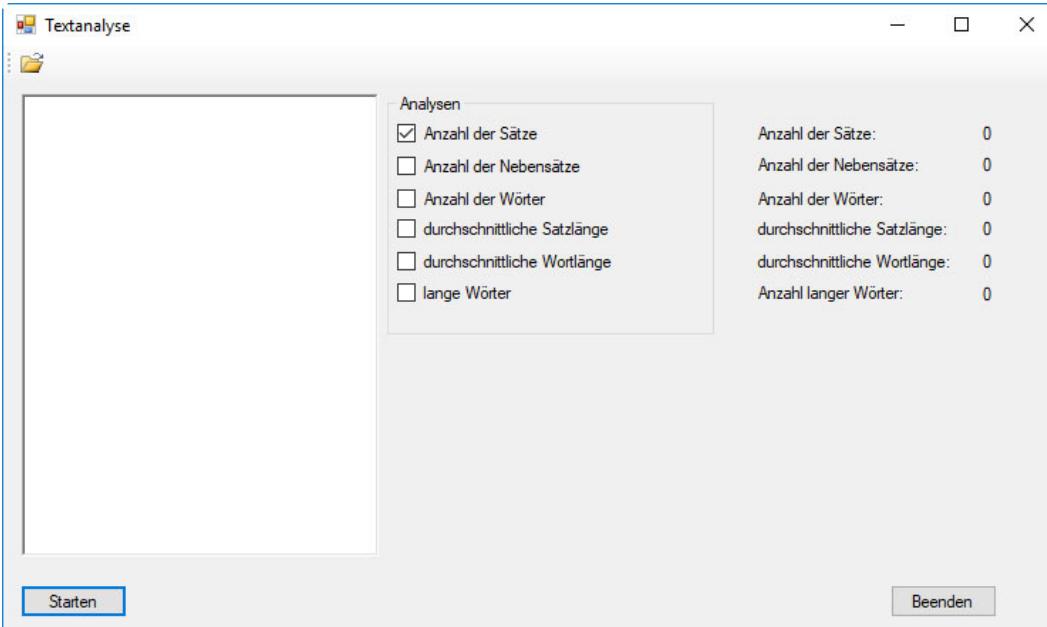


Abb. 2.3: Die Oberfläche für die Textanalyse

Beim Anklicken des Symbols laden Sie über einen Öffnendialog eine Textdatei in eine RichTextBox. Der entsprechende Code sieht so aus:

```
private void ÖffnenToolStripButton_Click(object sender,
EventArgs e)
{
    //den Öffnendialog anzeigen
    openFileDialog1.ShowDialog();
}

private void OpenFileDialog1_FileOk(object sender,
CancelEventArgs e)
{
    //den Dateinamen beschaffen und die Datei dann laden
    string dateiname;
    dateiname = openFileDialog1.FileName;
    //bitte in einer Zeile eingeben
    richTextBox1.LoadFile(dateiname,
    RichTextBoxStreamType.PlainText);
}
```

Code 2.8: Der Code zum Öffnen einer Datei

Beim Anklicken der Startschaltfläche überprüfen wir zunächst, ob überhaupt eine Eingabe im Feld vorliegt. Wenn nicht, beenden wir die Verarbeitung mit einer Meldung sofort wieder.

Danach prüfen wir, welche Kontrollkästchen markiert sind und führen die gewünschten Operationen aus. Beim Zählen benutzen wir die Methode `Count()` für den Text im Eingabefeld und geben durch einen Lambda-Ausdruck vor, was gezählt werden soll. Für das Zählen der Sätze sieht diese Anweisung zum Beispiel so aus:

```
labelAnzahlSaetze.Text = richTextBox1.Text.Count(c => c == '.').ToString();
```

Um uns die einzelnen Sätze und Wörter zu beschaffen, zerlegen wir die Eingabe durch die Methode `Split()` der Klasse `String` in Arrays mit Zeichenketten. Ein Array soll dabei unsere Sätze enthalten und ein Array unsere Wörter. Der Einfachheit halber verarbeiten wir in beiden Fällen immer den gesamten Text im Eingabefeld.

Als Argumente übergeben wir an die Methode `Split()` das Trennzeichen als `char`-Array und die Option `StringSplitOptions.RemoveEmptyEntries`. Sie sorgt dafür, dass leere Einträge nicht berücksichtigt werden. Das kann zum Beispiel passieren, wenn zwei Leerzeichen direkt hintereinander stehen.

Für das Beschaffen der Sätze sieht die Konstruktion dann so aus.

```
char[] punkt = { '.' };
string[] saetze = richTextBox1.Text.Split(punkt,
    StringSplitOptions.RemoveEmptyEntries);
```

Um uns die Anzahl der Wörter in einem Satz zu beschaffen, benutzen wir eine LINQ-Abfrage. Sie sieht so aus:

```
var abfrageSaetze = from satz in saetze
    select satz.Count(c => c == ' ');
```

Wir zählen hier die Anzahl der Leerzeichen in einem Satz und leiten daraus die Anzahl der Wörter ab.

Damit diese Technik funktioniert, muss auch der erste Satz mit einem Leerzeichen beginnen. Sonst „verzählt“ sich unser Programm. Alternativ könnten Sie auch vor das erste Element im Array `saetze` mit einer Anweisung ein Leerzeichen setzen.

Im Array für die Wörter beschaffen wir uns die Länge der einzelnen Wörter ebenfalls durch eine LINQ-Abfrage. Damit Kommas und Punkte nicht mitgezählt werden, entfernen wir sie durch die Methode `Trim()`. Die entsprechenden Anweisungen sehen so aus:

```
char[] entfernen = { ',', '.' };
var abfrageWoerter = from wort in woerter
    select wort.Trim(entfernen).Length;
```

Danach können wir dann mit den Methoden `Count()` und `Average()` für die Abfragevariablen die Anzahl und den Durchschnitt ermitteln. Dabei gibt es keine Besonderheiten.

Die Anzahl der Wörter, die länger sind als 10 Zeichen, können wir uns mit der folgenden LINQ-Abfrage beschaffen:

```
var abfrageLangeWoerter = from wort in woerter
    where wort.Length > 10
    select wort;
```

`woerter` ist dabei das Array, in dem die Wörter abgelegt sind.

Der komplette Code für die Methode zum Anklicken der Startschaltfläche sieht so aus:

```
private void ButtonStarten_Click(object sender, EventArgs e)
{
    //die verschiedenen Trennzeichen
    char[] punkt = { '.' };
    char[] leerzeichen = { ' ' };
    //ist das Textfeld leer?
    if (richTextBox1.Text == String.Empty)
    {
        MessageBox.Show("Sie haben keinen Text eingegeben.");
        return;
    }

    //sollen wir die Sätze zählen?
    if (checkBoxAnzahlSaetze.Checked)
        //bitte in einer Zeile eingeben
        labelAnzahlSaetze.Text = richTextBox1.Text.Count(c => c ==
            '.').ToString();
    else
        labelAnzahlSaetze.Text = "na";

    //die Nebensätze
    if (checkBoxAnzahlNebensaetze.Checked)
        //bitte in einer Zeile eingeben
        labelAnzahlNebensaetze.Text = richTextBox1.Text.Count(c => c
            == ',').ToString();
    else
        labelAnzahlNebensaetze.Text = "na";

    //die Sätze beschaffen
    //bitte jeweils in einer Zeile eingeben
    string[] saetze = richTextBox1.Text.Split(punkt,
        StringSplitOptions.RemoveEmptyEntries);
    //und auch die Wörter
    string[] woerter = richTextBox1.Text.Split(leerzeichen,
        StringSplitOptions.RemoveEmptyEntries);
    //die Länge der Sätze ermitteln
    //dazu zählen wir die Leerzeichen
    var abfrageSaetze = from satz in saetze
        select satz.Count(c => c == ' ');

    //die Länge der Wörter ermitteln
    //Punkt und Komma werden dabei entfernt
    char[] entfernen = { ',', '.' };
    var abfrageWoerter = from wort in woerter
        select wort.Trim(entfernen).Length;

    //die Anzahl der Wörter ermitteln
    if (checkBoxAnzahlWoerter.Checked)
        labelAnzahlWoerter.Text = abfrageWoerter.Count().ToString();
    else
        labelAnzahlWoerter.Text = "na";
```

```

if (checkBoxDurchschnittSatz.Checked)
    //bitte in einer Zeile eingeben
    labelDurchschnittSatz.Text =
        abfrageSaetze.Average().ToString("F2");
else
    labelDurchschnittSatz.Text = "na";
if (checkBoxDurchschnittWort.Checked)
    labelDurchschnittWort.Text =
        abfrageWoerter.Average().ToString("F2");
else
    labelDurchschnittWort.Text = "na";
if (checkBoxLangeWoerter.Checked)
{
    //die Wörter ermitteln, die länger sind als 10 Zeichen
    var abfrageLangeWoerter = from wort in woerter
        where wort.Length > 10
        select wort;
    //bitte in einer Zeile eingeben
    labelAnzahlLangeWoerter.Text =
        abfrageLangeWoerter.Count().ToString();
}
else
    labelAnzahlLangeWoerter.Text = "na";
}

```

Code 2.9: Die Methode für das Anklicken der Startschaltfläche

Hinweis:

Mit der Methode `ToString("F2")` wird die Anzahl der Nachkommastellen auf zwei gesetzt. Bei `F2` handelt es sich um eine Zahlenformatzeichenfolge.

Der Code lässt sich auch ein wenig einfacher gestalten. Sie können ja zum Beispiel die durchschnittliche Satzlänge auch direkt aus der Anzahl der Sätze und der Anzahl der Wörter berechnen. Die Methode `Average()` benötigen Sie dafür nicht.

Wenn die Methode `Split()` kein Trennzeichen in dem Text finden, wird immer der gesamte Text in das erste Element des Arrays gesetzt.

2.5 Exkurs: Generics

Bisher war bereits einige Male von der Schnittstelle `IEnumerable<T>` die Rede. Schauen wir uns einmal etwas genauer an, was sich dahinter verbirgt.

Bei der Schnittstelle `IEnumerable<T>` handelt es sich um eine generische Schnittstelle. Das bedeutet, diese Schnittstelle lässt sich nicht nur für einen bestimmten Typ, sondern für nahezu beliebige Typen verwenden. Welcher Typ konkret verwendet werden muss, wird zur Laufzeit aus dem übergebenen Typ abgeleitet. Damit ist ein generische Schnittstelle sehr viel flexibler als eine nicht generische Schnittstelle, die sich nur für einen bestimmten Typ verwenden lässt.

Neben generischen Schnittstellen gibt es unter anderem auch generische Methoden und generische Klassen. Solche Methoden und Klassen lassen sich dann ebenfalls nicht nur mit einem bestimmten Typ verwenden, sondern für nahezu beliebige Typen. Eine generische Klasse – die Klasse `List<T>` – haben Sie ja bereits kennengelernt.

Zur Auffrischung:

Generics – auch Generika genannt – sind Methoden, Klassen, Strukturen und so weiter, die Platzhalter für die Typen benutzen, die von ihnen verarbeitet werden. Die konkreten Typen für die Platzhalter werden erst zur Laufzeit des Programms eingesetzt.

Generics unterscheiden sich nur durch die Typparameter von normalen Methoden oder Klassen. Der Typparameter wird hinter dem Bezeichner in spitzen Klammern angegeben.

Im folgenden Code wird zum Beispiel eine generische Methode `Ausgabe()` vereinbart, die den Wert von beliebigen Typen ausgeben kann.

```
using System;

namespace Cshp23d_02_10
{
    class Program
    {
        //die generische Methode zur Ausgabe
        static void Ausgabe<TAusgabe>(TAusgabe wert)
        {
            //bitte in einer Zeile eingeben
            Console.WriteLine("Der Typ ist {0} und der Wert {1}.",
                wert.GetType(), wert);
        }

        static void Main(string[] args)
        {
            Ausgabe(10);
            Ausgabe(11.12);
            Ausgabe(true);
            Ausgabe("Eine Zeichenkette");
        }
    }
}
```

Code 2.10: Eine generische Methode zur Addition

Der Typparameter ist hier `TAusgabe`. Darüber wird auch der Typ des Parameters `wert` festgelegt.

Das .NET Framework kennt zahlreiche generische Klassen – zum Beispiel eben die Klasse `List<T>`. Mit ihr können Sie – wie Sie ja bereits wissen – Listen für nahezu beliebige Datentypen erstellen.

Schauen wir uns diese Klasse einmal ein wenig genauer an.

Ein sehr einfaches Programm, das eine Liste mit `int`-Werten über eine generische Liste erzeugt und ausgibt, finden Sie im folgenden Code:

```
using System;
using System.Collections.Generic;

namespace Cshp23d_02_11
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Liste erzeugen
            List<int> liste = new List<int>(10);
            //Elemente in die Liste eintragen
            for (int i = 1; i < 11; i++)
                liste.Add(i);
            //die Liste ausgeben
            for (int i = 0; i < 10; i++)
                Console.WriteLine("{0}", liste[i]);
        }
    }
}
```

Code 2.11: Eine generische Liste über die Klasse `List`

Hinweis:

Die Klasse `List<T>` befindet sich im Namensraum `System.Collections.Generic`. Dieser Namensraum wird aber in der Regel automatisch durch das Quelltextgerüst von Visual Studio eingebunden.

Mit der Anweisung

```
List<int> liste = new List<int>(10);
```

erzeugen wir zunächst einmal eine Liste für den Typ `int` mit 10 Elementen.

Danach stellen wir in der `for`-Schleife

```
for (int i = 1; i < 11; i++)
    liste.Add(i);
```

10 Elemente über die Methode `Add()` in die Liste. Als Argument übergeben wir dabei den Wert für das Element.



Zur Erinnerung:

Bei den generischen Listen beginnt die Nummerierung wie bei den Arrays mit 0 und nicht mit 1. Der erste Eintrag hat damit den Index 0 und das letzte Element den Index Anzahl -1.

In der zweiten `for`-Schleife

```
for (int i = 0; i < 10; i++)
    Console.WriteLine("{0}", liste[i]);
```

geben wir dann die Elemente wieder aus. Der Zugriff auf jedes Element erfolgt dabei über den Index `i`.

Besonders spannend ist der Code bisher allerdings noch nicht – denn exakt dieselbe Liste lässt sich ja auch mit weniger Aufwand über ein Array erzeugen. Interessant wird es aber beim Erweitern der Liste. Denn bei einer generischen Liste können Sie ohne Weiteres neue Elemente anhängen – auch wenn die Liste dabei größer wird als ursprünglich angegeben.

So können Sie zum Beispiel im vorigen Code die Schleifen einfach laufen lassen, bis `i` den Wert 1000 erreicht hat. Die Liste wird beim Anlegen automatisch erweitert, obwohl die Anfangskapazität lediglich 10 Elemente beträgt. Auch 10 000 Elemente lassen sich ohne Weiteres ablegen. Sie müssen lediglich die Schleifen entsprechend anpassen.

Die Anzahl der Elemente in einer generischen Liste können Sie sehr einfach über die Methode `Count()` ermitteln. Die folgende Schleife gibt zum Beispiel sämtliche Elemente der Liste aus – gleichgültig, wie groß die Liste nun ist:

```
for (int i = 0; i < liste.Count(); i++)
    Console.WriteLine("{0}", liste[i]);
```

Der folgende Code erstellt eine Liste mit 10 000 Zahlen und gibt sie aus:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Cshp23d_02_12
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Liste erzeugen
            List<int> liste = new List<int>(10);
            //Elemente in die Liste eintragen
            for (int i = 1; i < 10000; i++)
                liste.Add(i);
            //die Liste ausgeben
            for (int i = 0; i < liste.Count(); i++)
                Console.WriteLine("{0}", liste[i]);
        }
    }
}
```

Code 2.12: Eine lange Liste

Die Größe einer Liste, die Sie über die Klasse `List` verwalten, ist übrigens nur durch den Speicher begrenzt. Erweitern Sie zum Beispiel einfach einmal die Schleife in dem vorigen Code so, dass 1 000 000 Elemente erstellt werden.

Zusammenfassung

Der grundsätzliche Aufbau einer LINQ-Abfrage ist immer gleich.

Die Abfragen werden verzögert ausgeführt.

Über das Schlüsselwort `where` können Daten selektiert werden.

Der `select`-Ausdruck legt fest, welche Werte die Abfrage liefert.

Neben der Abfragesyntax unterstützt LINQ auch die Methodensyntax. Sie arbeitet mit Lambda-Ausdrücken.

Daten lassen sich in einer Abfrage über das Schlüsselwort `group` gruppieren.

LINQ kennt Abfrageoperatoren wie `Count()`, die Ihnen einzelne Werte zu einer Menge liefern – zum Beispiel die Anzahl.

Über Generics können Sie nahezu beliebige Typen verarbeiten.

Die Klasse `List<T>` kann Listen abbilden, die Sie beliebig erweitern und verändern können.

Aufgaben zur Selbstüberprüfung

- 2.1 Aus welchen drei Teilen besteht eine LINQ-Abfrage?
- 2.2 Was legt das Schlüsselwort `from` bei einer LINQ-Abfrage fest?
- 2.3 Müssen Sie für die Bereichsvariablen einer LINQ-Abfrage einen Datentyp angeben? Wenn nicht: Woher wird der Typ beschafft?
- 2.4 Formulieren Sie eine LINQ-Abfrage, die Ihnen aus einer Datenquelle `zahlen` alle Werte beschafft, die größer als 10 und kleiner als 100 sind.
- 2.5 Formulieren Sie eine LINQ-Abfrage, die Ihnen aus einer Datenquelle `zeichenketten` alle Zeichenketten beschafft, die mit einem großen A beginnen.
- 2.6 Mit welchem Schlüsselwort können Sie Ergebnisse einer LINQ-Abfrage sortieren?
- 2.7 Was ist das Zeichen `=>` bei der LINQ-Abfrage in Methodensyntax? Wofür steht dieses Zeichen?
- 2.8 Was liefert Ihnen der Operator `Distinct()`?

3 LINQ to XML

In diesem Kapitel beschäftigen wir uns mit LINQ to XML. Dazu erstellen wir eine kleine Anwendung, die Daten aus einer XML-Datei verarbeitet.

LINQ to XML ist ein Programmierschnittstelle, die die Verarbeitung von XML-Daten ermöglicht. Die Daten werden dabei komplett im Arbeitsspeicher gehalten.

Zur Auffrischung:

Bei XML handelt sich um ein textbasiertes, strukturiertes Format für die Beschreibung von Daten. Sämtliche Informationen in einer XML-Datei werden als einfacher Text in einer hierarchischen Baumstruktur gespeichert. Diese Struktur besteht aus einem **Wurzelement** und nahezu beliebigen **Unterelementen** – den **Knoten**.

Für die Beschreibung und Strukturierung werden Tags benutzt. Sie stehen in spitzen Klammern < >.



Zu LINQ to XML gehören verschiedene Klassen wie `XDocument` für ein XML-Dokument oder `XElement` für ein XML-Element. Einige dieser Klassen werden wir uns jetzt im praktischen Einsatz ansehen.

3.1 Daten lesen

Beginnen wir mit einem einfachen Beispiel. Wir öffnen eine Bestenliste mit Namen und Punkten, die als XML-Datei gespeichert ist. Der Inhalt der Datei sieht so aus:

```
<?xml version="1.0" encoding="utf-8" ?>
<eintraege>
  <eintrag>
    <name>Müller</name>
    <punkte>100</punkte>
  </eintrag>
  <eintrag>
    <name>Meier</name>
    <punkte>1000</punkte>
  </eintrag>
  <eintrag>
    <name>Schmitz</name>
    <punkte>10</punkte>
  </eintrag>
  <eintrag>
    <name>Strietzel</name>
    <punkte>1</punkte>
  </eintrag>
</eintraege>
```

Danach beschaffen wir uns über eine LINQ-Abfrage alle Daten und geben sie in der Eingabeaufforderung aus.

Der Code sieht so zunächst einmal so aus:

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace Cshp23d_03_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Datei öffnen
            XElement xmlElement =
                XElement.Load("c:\\test\\liste.xml");
            //die Einträge beschaffen
            //bitte in einer Zeile eingeben
            var eintraege = from eintrag in
                xmlElement.Elements(("eintrag"))
                    select eintrag;
            //und ausgeben
            foreach (var eintrag in eintraege)
                Console.WriteLine(eintrag);
        }
    }
}
```

Code 3.1: Das Lesen von XML-Daten über die Klasse XElement

Mit der Anweisung

```
XElement xmlElement = XElement.Load("c:\\test\\liste.xml");
```

laden wir die Datei liste.xml aus dem Ordner c:\\test.

Hinweis:

Die Datei finden Sie bei den Beispielen im heftbezogenen Download-Bereich Ihrer Online-Lernplattform. Wenn Sie eine andere Datei verwenden, müssen Sie den Namen und den Pfad unter Umständen anpassen.

Damit die Klasse XElement bekannt ist, müssen Sie den Namensraum System.Xml.Linq einbinden.

Danach beschaffen wir uns alle Einträge in der Datei, die zum Element eintrag gehören. Das erledigt die Abfrage

```
var eintraege = from eintrag in xmlElement.Elements(("eintrag"))
    select eintrag;
```

Auf die Einträge greifen wir dabei mit der Methode Elements der Klasse XElement zu. Ansonsten gibt es keine Besonderheiten.

Anschließend geben wir die Daten aus. Das erledigt wie gewohnt die Schleife am Ende des Codes.

Ausgegeben wird allerdings die komplette XML-Struktur mitsamt Tags. Die Ausgabe sieht also so aus:

```
<eintrag>
  <name>Müller</name>
  <punkte>100</punkte>
</eintrag>
<eintrag>
  <name>Meier</name>
  <punkte>1000</punkte>
</eintrag>
<eintrag>
  <name>Schmitz</name>
  <punkte>10</punkte>
</eintrag>
<eintrag>
  <name>Strietzel</name>
  <punkte>1</punkte>
</eintrag>
```

Eine sehr einfache Möglichkeit, die Tags zu entfernen, besteht darin, dass Sie das Element in den Typ `string` konvertieren. Die Abfrage könnte dann so aussehen:

```
var eintraege = from eintrag in xmlElement.Elements(("eintrag"))
  select (string)eintrag;
```

Allerdings werden die Einträge jetzt aneinandergehängt. Die Ausgabe sieht so aus:

```
Müller100
Meier1000
Schmitz10
Strietzel1
```

Besonders brauchbar ist das ebenfalls nicht.

Um alle Einträge zu trennen, müssen Sie ein wenig mehr Aufwand betreiben. Sie legen die Einträge getrennt in einem neu erzeugten Element ab. Der entsprechende Code sieht dann so aus:

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace Cshp23d_03_02
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Datei öffnen
            XElement xmlElement =
            XElement.Load("c:\\\\test\\\\liste.xml");
            //die Einträge beschaffen
            //bitte in einer Zeile eingeben
            var eintraege = from eintrag in
            xmlElement.Elements(("eintrag"))
```

```

        select new
    {
        Name = eintrag.Element("name").Value,
        Punkte = eintrag.Element("punkte").Value,
    };
    //und ausgeben
    foreach (var eintrag in eintraege)
        //bitte in einer Zeile eingeben
        Console.WriteLine("Name: {0} Punkte: {1}", eintrag.Name,
        eintrag.Punkte);
    }
}
}

```

Code 3.2: Trennen von Elementen

In der Abfrage erzeugen wir über

```

select new
{
    Name = eintrag.Element("name").Value,
    Punkte = eintrag.Element("punkte").Value,
};

```

ein neues Element für `eintrag`. Es enthält die Eigenschaften `Name` und `Punkte`. Die Werte dieser beiden Eigenschaften setzen wir über die Eigenschaft `Value` für dass entsprechende Element aus der XML-Datei. Das Element beschaffen wir uns dabei über die Methode `Element()`.

Danach können wir die Elemente in der Schleife einzeln ausgeben.

Sie können auch gezielt auf einzelne Werte zugreifen. Im folgenden Code werden zum Beispiel nur die Einträge für den Namen „Müller“ ausgegeben.

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace Cshp23d_03_03
{
    class Program
    {
        static void Main(string[] args)
        {
            //die Datei öffnen
            XElement xmlElement =
            XElement.Load("c:\\test\\liste.xml");
            //die Einträge beschaffen
            //bitte in einer Zeile eingeben
            var eintraege = from eintrag in
            xmlElement.Elements("eintrag")
                where eintrag.Element("name").Value == "Müller"
                select new
            {
                Name = eintrag.Element("name").Value,

```

```
        Punkte = eintrag.Element("punkte").Value,  
    };  
    //und ausgeben  
    foreach (var eintrag in eintraege)  
        //bitte in einer Zeile eingeben  
        Console.WriteLine("Name: {0} Punkte: {1}", eintrag.Name,  
            eintrag.Punkte);  
    }  
}
```

Code 3.3: Gezielter Zugriff auf einen Eintrag

Hinweis:

Die Klasse `XElement` kennt noch weitere Methoden, um Daten zu beschaffen – zum Beispiel die Methode `Descendants()`⁷. Sie beschafft Ihnen die Nachfolgeelemente eines Elements. Mehr zu den verschiedenen Methoden finden Sie in der Hilfe.

3.2 XML-Elemente erzeugen

Mit LINQ to XML können Sie auch sehr einfach XML-Strukturen und neue Elemente erzeugen. Dazu erzeugen Sie eine neue Instanz der Klasse `XElement` und übergeben über den Konstruktor die gewünschten Werte. Dabei können Sie auch geschachtelte Strukturen erzeugen.

Eine Bestenliste mit Einträgen für den Namen und die Punkte ließe sich zum Beispiel so erstellen:

7. Descendants bedeutet übersetzt so viel wie „Nachkommen“.

```

        new XElement("eintrag",
            new XElement("name", "Strietzel"),
            new XElement("punkte", "1")
        )
    );

    foreach (var eintrag in bestenliste.Elements("eintrag"))
        //bitte in einer Zeile eingeben
        Console.WriteLine("Name: {0} Punkte: {1}",
            eintrag.Element("name").Value,
            eintrag.Element("punkte").Value);
    }
}

```

Code 3.4: Das Erzeugen von neuen Strukturen und Elementen

Mit der Methode `Add()` von `XElement` können Sie ganz einfach Daten an das Ende einer Struktur anhängen. Die folgende Anweisung würde zum Beispiel einen neuen Eintrag an das Ende unserer Bestenliste schreiben.

```

bestenliste.Add(new XElement("eintrag",
    new XElement("name", "Gewinner"),
    new XElement("punkte", "100000")
));

```

Über die Methode `SetValue()` von `XElement` können Sie Werte verändern. Das folgende Fragment filtert über eine LINQ-Abfrage alle Einträge in der Bestenliste mit dem Namen „Müller“ und setzt die Punkte auf 500 000.

```

var eintraege = from eintrag in bestenliste.Elements("eintrag")
    where eintrag.Element("name").Value == "Müller"
    select eintrag;

foreach (var eintrag in eintraege)
    eintrag.Element("punkte").SetValue("5000000");

```

Hinweis:

Den vollständigen Code finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **Cshp23d_03_05**.

Sie können über die Klasse `XElement` auch Daten löschen. Dazu verwenden Sie die Methode `Remove()`. Allerdings lassen sich keine Knoten aus einer Abfrage löschen, die Sie gerade verarbeiten. Sie können aber über die Methode `ToList()` für die Abfragevariable eine Liste vom Typ `List<T>` erzeugen und dann die Elemente in dieser Liste löschen. Das folgende Fragment löscht zum Beispiel alle Einträge mit dem Namen „Müller“ aus der Bestenliste:

```

//die Einträge beschaffen
var eintraege = from eintrag in xmlElement.Elements(("eintrag"))
    where (eintrag.Element("name").Value == "Müller")
    select eintrag;

//eine Liste aus den Einträgen erstellen
var meineListe = eintraege.ToList();

```

```
//und jeden Eintrag löschen  
foreach (var eintrag in meineListe)  
    eintrag.Remove();
```

Mit der Anweisung

```
var meineListe = eintraege.ToList();
```

wird eine Liste mit dem Ergebnis der Abfrage erstellt. Die Schleife

```
foreach (var eintrag in meineListe)  
    eintrag.Remove();
```

durchläuft diese Liste und löscht jeden Eintrag über die Methode `Remove()`.

Hinweis:

Ein vollständiges Beispiel zum Löschen von Daten finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **Cshp23d_03_06**.

3.3 Daten schreiben

Für das Schreiben von Daten stellt Ihnen die Klasse `XElement` die Methode `Save()` zur Verfügung. Als Argument übergeben Sie den Namen der Datei.

Die folgende Anweisung speichert zum Beispiel unsere Bestenliste unter dem Namen `fakeliste.xml` im Ordner `c:\test`.

```
XmlElement.Save("c:\\test\\fakeliste.xml");
```

Hinweis:

Ein vollständiges Beispiel zum Speichern finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform im Projekt **Cshp23d_03_07**. Dort werden die Punkte in der Bestenliste verdoppelt und wieder in die Liste geschrieben.

3.4 Ein komplexeres Beispiel

Zum Abschluss dieses Kapitels werden wir wieder eine etwas komplexere Anwendung erstellen. Wir lassen die Daten aus einer XML-Datei in einer WPF-Anwendung über ein `DataGridView` anzeigen. Der Anwender soll dabei Daten bearbeiten und neue Einträge ergänzen können.

Die fertige Anwendung soll später so aussehen:

Name	Punkte
Müller	100
Meier	1000
Schmitz	10
Strietzel	1
<leer>	<leer>

Abb. 3.1: Die fertige Anwendung

Hinweis:

Das vollständige Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **LINQtoXMLWPF**.

Beim ersten Durchlesen hört sich das vielleicht recht anspruchsvoll an. Tatsächlich ist es aber durch die Datenbindung von WPF und LINQ to XML schnell erledigt. Wir binden das DataGrid an unsere XML-Datei, die wir über die Klasse `XElement` laden und auch wieder speichern.

Ein wenig anspruchsvoller ist die Datenbindung für das DataGrid im XAML-Code. Sie sieht so aus:

```
<!-- bitte in einer Zeile eingeben -->
<DataGrid x:Name="bestenlisteGrid" AutoGenerateColumns="False"
ItemsSource="{Binding Path=Elements[eintrag]}" Height="320"
VerticalAlignment="Top">
    <DataGrid.Columns>
        <!-- bitte jeweils in einer Zeile eingeben -->
        <DataGridTextColumn Header="Name" Binding="{Binding
Path=Element [name].Value}" Width="100"/>
        <DataGridTextColumn Header="Punkte" Binding="{Binding
Path=Element [punkte].Value}" Width="*"/>
    </DataGrid.Columns>
</DataGrid>
```

Code 3.5: Das DataGrid für unsere Anwendung

Hier erstellen wir zunächst ein `DataGridView` mit dem Namen `bestenlisteGrid`. Damit nur die Spalten angezeigt werden, die wir selbst erzeugen, setzen wir die Eigenschaft `AutoGenerateColumns` auf `False`. Über die Eigenschaft `ItemsSource` legen wir fest, dass die Daten für das `DataGridView` aus den Elementen mit dem Tag `eintrag` in unserer Datei mit der Bestenliste stammen sollen.

Dann erzeugen wir die beiden Spalten. Eine erhält den Titel „Name“ und die andere den Titel „Punkte“. Die beiden Spalten binden wir über die Eigenschaft `Binding` an den Wert des Namens beziehungsweise der Punkte.

Beginnen wir jetzt mit der praktischen Umsetzung. Legen Sie bitte eine neue WPF-Anwendung an. Setzen Sie oben in das Fenster das `DataGridView` aus dem vorigen Code. Unten fügen Sie drei Schaltflächen ein. Die linke Schaltfläche dient zum Laden der Datei, die mittlere zum Speichern und die rechte zum Anhängen eines neuen leeren Eintrags.

Weisen Sie den drei Schaltflächen die passende Methode aus dem folgenden Code zu:

```
private void ButtonLaden_Click(object sender, RoutedEventArgs e)
{
    bestenliste = XElement.Load("c:\\\\test\\\\testliste.xml");
    bestenlisteGrid.DataContext = bestenliste;
}

private void ButtonSpeichern_Click(object sender,
RoutedEventArgs e)
{
    bestenliste.Save("c:\\\\test\\\\testliste.xml");
}

private void ButtonNeu_Click(object sender, RoutedEventArgs e)
{
    bestenliste.Add(new XElement("eintrag",
        new XElement("name", "<leer>"),
        new XElement("punkte", "<leer>")
    ));
}
```

Code 3.6: Die Methoden für die Schaltflächen

Beim Laden öffnen wir die Datei und verbinden die Daten über die Eigenschaft `DataContext` mit dem `DataGridView` der Bestenliste. Das führt automatisch dazu, dass die Daten nach dem Laden angezeigt werden.

Hinweis:

`bestenliste` im vorigen Code ist als Feld vom Typ `XElement` vereinbart.

Beim Speichern sichern wir die Daten über die Methode `Save()`. Dabei gibt es keine Besonderheiten.

Interessant ist dann noch einmal das Anhängen eines neuen Eintrags. Hier erzeugen wir über die Methode `Add()` einen neuen Eintrag für unsere Bestenliste. Da sie mit dem `DataGridView` verbunden ist, wird dieser neue Eintrag auch automatisch im `DataGridView` angezeigt.

So viel zu LINQ to XML.

Zusammenfassung

Über LINQ to XML können Sie XML-Daten verarbeiten. Die Daten werden dabei im Speicher gehalten.

Zu LINQ to XML gehören verschiedene Klassen wie `XDocument` für ein XML-Dokument oder `XElement` für ein XML-Element.

Über die Methoden `Load()` und `Save()` der Klasse `XElement` können Sie XML-Dateien laden und speichern.

Um die Tags aus einer Liste zu entfernen, die sich über eine LINQ-Abfrage beschafft haben, können Sie die Einträge in eine Zeichenkette konvertieren. Dabei werden allerdings die Einträge aneinandergehängt.

Über den Konstruktor der Klasse `XElement` können Sie neue Einträge erzeugen.

Über die Methode `SetValue()` der Klasse `XElement` können Sie Werte löschen.

Mit der Methode `Remove()` der Klasse `XElement` löschen Sie Werte.

Aufgaben zur Selbstüberprüfung

- 3.1 Sie wollen eine XML-Datei mit dem Namen `meineDatei.xml` aus dem Ordner `meinOrdner` auf dem Laufwerk C: mit der Klasse `XElement` laden. Formulieren Sie eine entsprechende Anweisung. Den Bezeichner können Sie dabei frei wählen.
- 3.2 Sie haben eine XML-Datei über die Klasse `XElement` geladen und wollen sich alle Elemente beschaffen, die zum Eintrag `meinEintrag` gehören. Formulieren Sie eine entsprechende LINQ-Abfrage. Die Bezeichner können Sie dabei wieder frei wählen.
- 3.3 Sie haben Daten aus einer XML-Datei über die Klasse `XElement` geladen. Ein Eintrag besteht jeweils aus dem Vornamen und dem Nachnamen. Die Werte sind in den Elementen `vorname` und `nachname` in der XML-Datei gespeichert.
Erstellen Sie eine LINQ-Abfrage und eine Schleife, die Ihnen die beiden Werte sauber getrennt liefert und in der Eingabeaufforderung ausgibt. Die Bezeichner können Sie auch hier wieder frei wählen.
- 3.4 Mit welcher Methode der Klasse `XElement` können Sie neue Elemente an eine vorhandene Struktur anhängen?
- 3.5 Welche Besonderheit müssen Sie beim Löschen von Elementen über die Klasse `XElement` beachten?

4 Eine kleine Datenbankanwendung

In diesem Kapitel erstellen wir eine kleine Datenbankanwendung, die Daten aus einer Datenbanktabelle über LINQ to SQL verarbeitet. Als Plattform verwenden wir dabei WPF.

4.1 Vorüberlegungen und Vorbereitungen

Die Anwendung soll Kontaktdaten verarbeiten können. Die Anzeige der Daten erfolgt über ein DataGridView. Für die Bearbeitung stellen wir dem Anwender Schaltflächen zum Erstellen, Bearbeiten und Löschen von Einträgen zur Verfügung.

Die fertige Anwendung soll so aussehen:

MiniDB							
Kontaktnummer	Vorname	Name	PLZ	Ort	Straße	Telefon	E-Mail
1	Hans	Franz	12345	Meierhausen	Meierweg 13	01234/56789	hans@franz.de
2	Max	Meier	12345	Meierhausen	Maxweg 1	001/002	max@meier.de

Abb. 4.1: Die Datenbankanwendung

Bei LINQ to SQL erfolgt der Datenzugriff nicht direkt auf das Datenmodell der Datenbank, sondern über ein Objektmodell, das das Datenmodell abbildet. Alle Zugriffe erfolgen dann über das Objektmodell, das seinerseits wieder auf das Datenmodell zugreift.

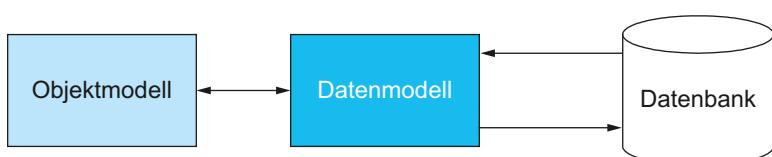


Abb. 4.2: Das Daten- und Objektmodell

Die Abbildung einer Datenbanktabelle im Objektmodell erfolgt über eine Entitätsklasse.

Eine Entität beschreibt ein Objekt. Zu einer Entität gehören Attribute, die das Objekt beschreiben.



Das heißt: Für den Zugriff auf die Datenbanktabelle mit LINQ to SQL benötigen Sie erst einmal eine entsprechende Entitätsklasse. Sie lässt sich aber sehr komfortabel mit dem Object Relational Designer von Visual Studio erstellen.

Legen Sie jetzt bitte eine neue WPF-Anwendung an. Als Namen können Sie zum Beispiel **MiniDBWPF** verwenden. Ändern Sie dann den Fenstertitel – zum Beispiel in **MiniDB**.

Hinweis:

Das Projekt finden Sie im heftbezogenen Download-Bereich Ihrer Online-Lernplattform unter dem Namen **MiniDBWPF**.

4.2 Erstellen der Datenbank

Danach legen Sie die Datenbank und die Datenbanktabelle an. Da Sie diese Schritte bereits kennen, stellen wir sie Ihnen hier nur in kompakter Form vor.

Öffnen Sie den Server-Explorer über die Registerzunge **Server-Explorer**. Klicken Sie dann auf das Symbol **Mit Datenbank verbinden** . Wählen Sie eine Microsoft SQL Server-Datenbankdatei aus und geben Sie einen Namen ein. Wir verwenden in unserem Beispiel den Namen **kontakte**.

Die Einstellungen für die Anmeldung beim Server können Sie unverändert stehen lassen. Klicken Sie anschließend auf **OK** und bestätigen Sie die Abfrage, ob die Datenbankdatei erstellt werden soll.

Im nächsten Schritt müssen wir nun noch die Tabelle anlegen. Klicken Sie im Server-Explorer auf das Symbol vor dem Eintrag **kontakte.mdf**. Wählen Sie im Kontext-Menü für den Eintrag **Tabellen** die Funktion **Neue Tabelle hinzufügen**.

Legen Sie anschließend die Spalten an. Die erforderlichen Daten finden Sie in der folgenden Tabelle. Die Spalte **kNummer** soll dabei als Primärschlüssel dienen

Tab. 4.1: Die Spalten für die Datenbanktabelle

Spaltenname	Datentyp	NULL-Werte zulassen	Länge
kNummer	int	nein	
nachname	nvarchar	nein	50
vorname	nvarchar	nein	50
strasse	nvarchar	nein	50
postleitzahl	nvarchar	nein	5
ort	nvarchar	nein	50
telefon	nvarchar	ja	20
email	nvarchar	ja	30

Legen Sie außerdem fest, dass die Kontaktnummer in der Spalte **kNummer** automatisch vergeben wird. Setzen Sie dazu die Eigenschaft (**Ist Identity**) in der Gruppe **Identitäts-spezifikation** auf **True**.

Ändern Sie den Namen der Tabelle in **Kontakte**. Dazu ändern Sie den Eintrag **[Table]** in der Anweisung

```
CREATE TABLE [dbo] . [Table]
```

oben im Register **T-SQL** in **[Kontakte]**.

Lassen Sie abschließend die Datenbank aktualisieren.

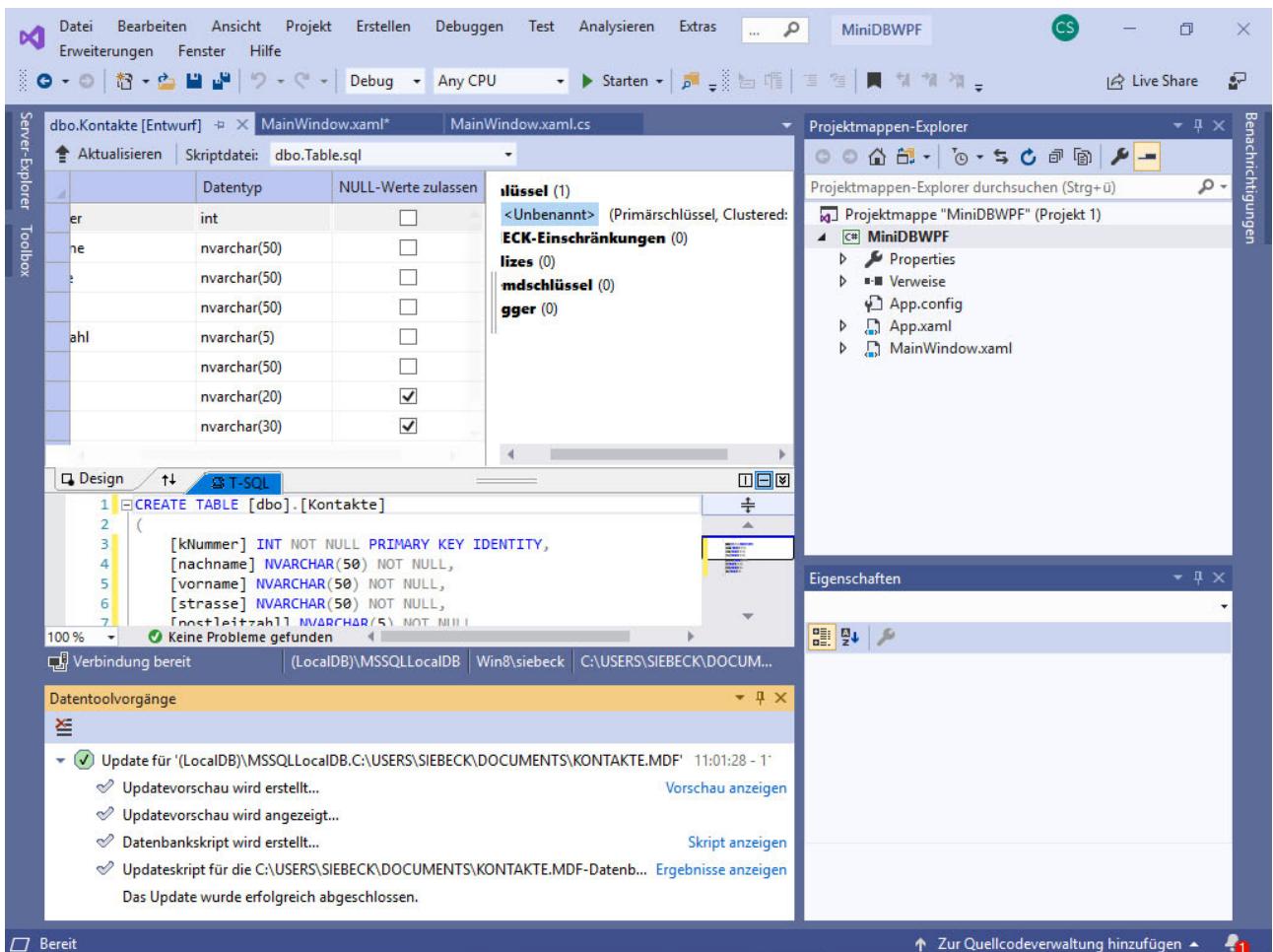


Abb. 4.3: Die aktualisierte Datenbank

4.3 Installation der Tools

Im nächsten Schritt müssen wir die Tools für LINQ to SQL installieren. Sonst steht Ihnen unter anderem der Object Relational Designer nicht zur Verfügung.

Speichern Sie alle Änderungen und schließen Sie Visual Studio. Starten Sie dann den Visual Studio Installer – zum Beispiel über das Startmenü.

Im Visual Studio Installer klicken Sie auf die Schaltfläche **Mehr** neben der installierten Version und wählen im Menü den Eintrag **Ändern**. Wechseln Sie dann in den Bereich **Einzelne Komponenten** und verschieben Sie die Anzeige, bis der Bereich **Codetools** angezeigt wird.

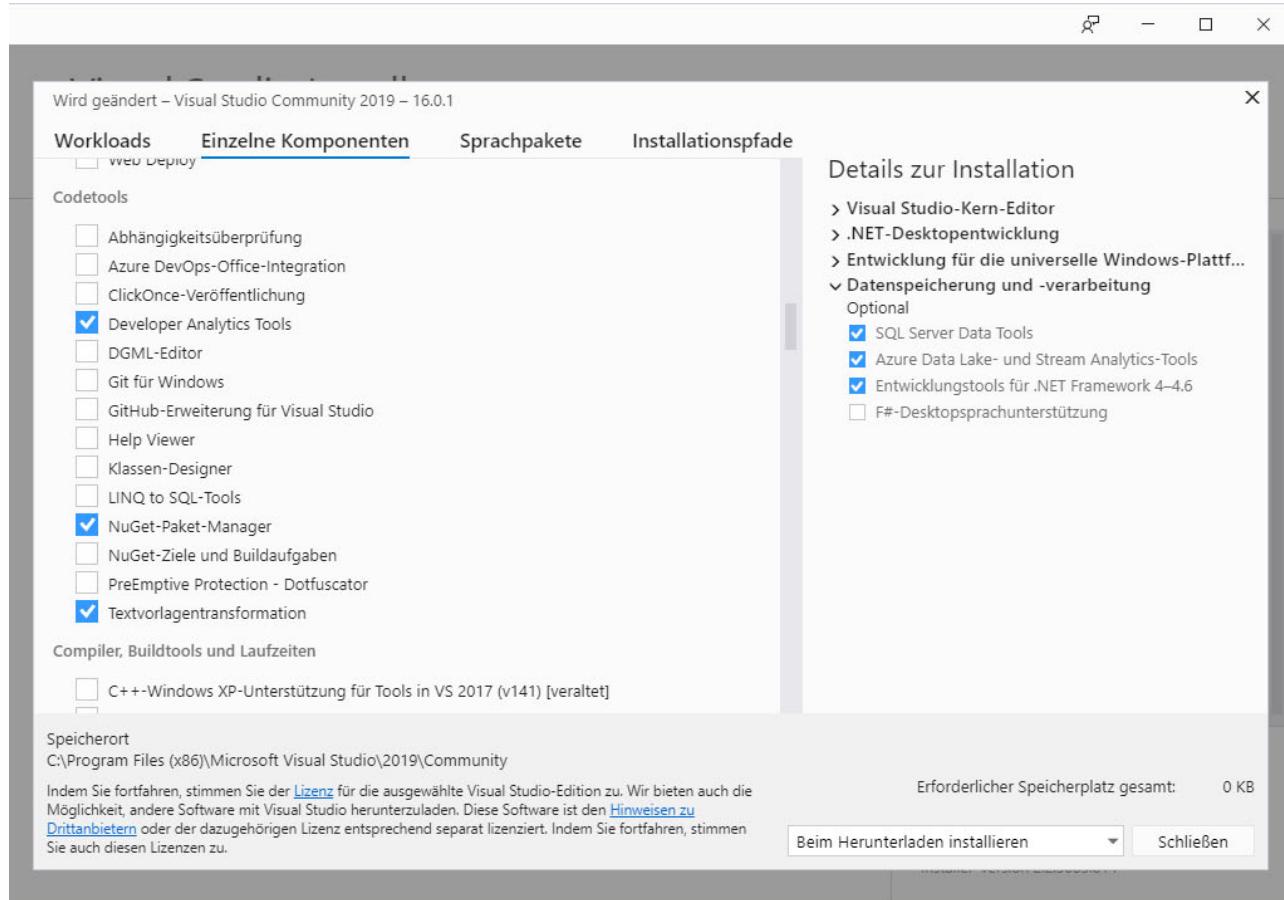


Abb. 4.4: Der Bereich Codetools

Markieren Sie hier den Eintrag **LINQ to SQL-Tools**. Klicken Sie anschließend auf die Schaltfläche **Ändern** rechts unten im Fenster, um die Installation zu starten. Beenden Sie danach den Visual Studio Installer und öffnen Sie das Projekt für die Datenbankanwendung wieder.

4.4 Entitätsklasse für den Zugriff erstellen

Jetzt können wir die Entitätsklasse für den Zugriff erstellen. Fügen Sie dazu über die Funktion **Projekt/Neues Element hinzufügen...** ein neues Element ein. Im Fenster **Neues Element hinzufügen** markieren Sie den Eintrag **LINQ to SQL-Klassen**. Sie finden ihn am schnellsten im Bereich **Daten**.

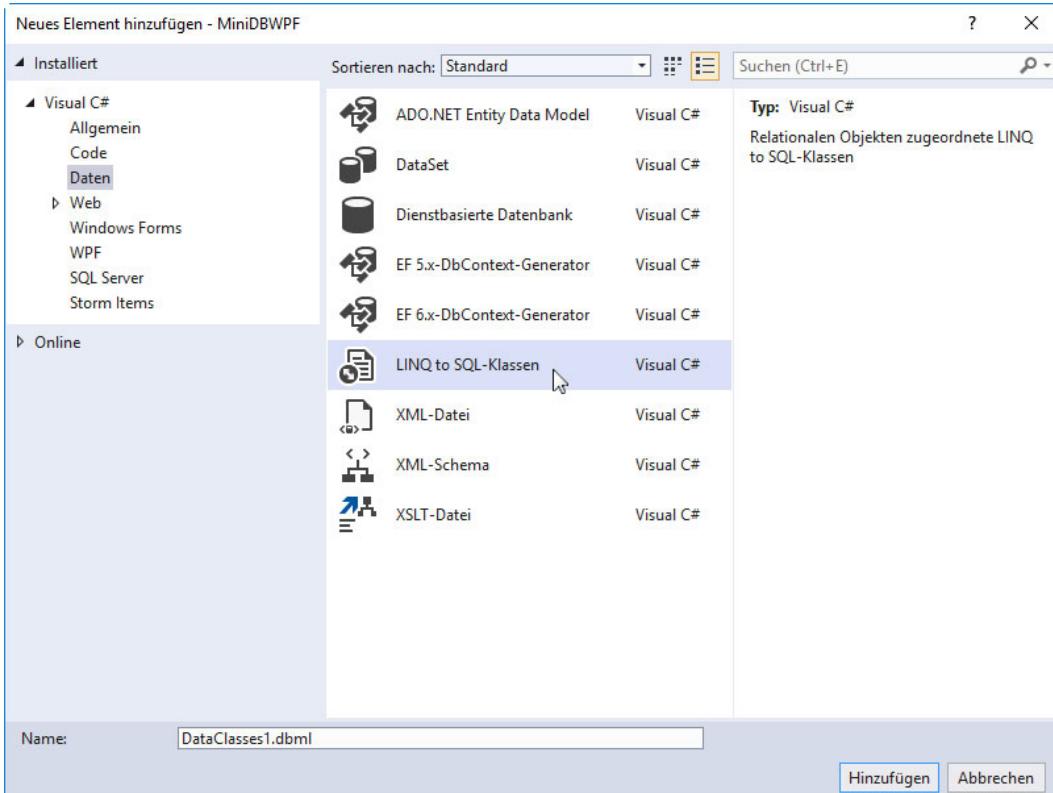


Abb. 4.5: Der Eintrag LINQ to SQL-Klassen im Bereich Daten (in der Mitte der Abbildung am Mauszeiger)

Geben Sie als Namen **Kontakte** ein und klicken Sie auf **Hinzufügen**.

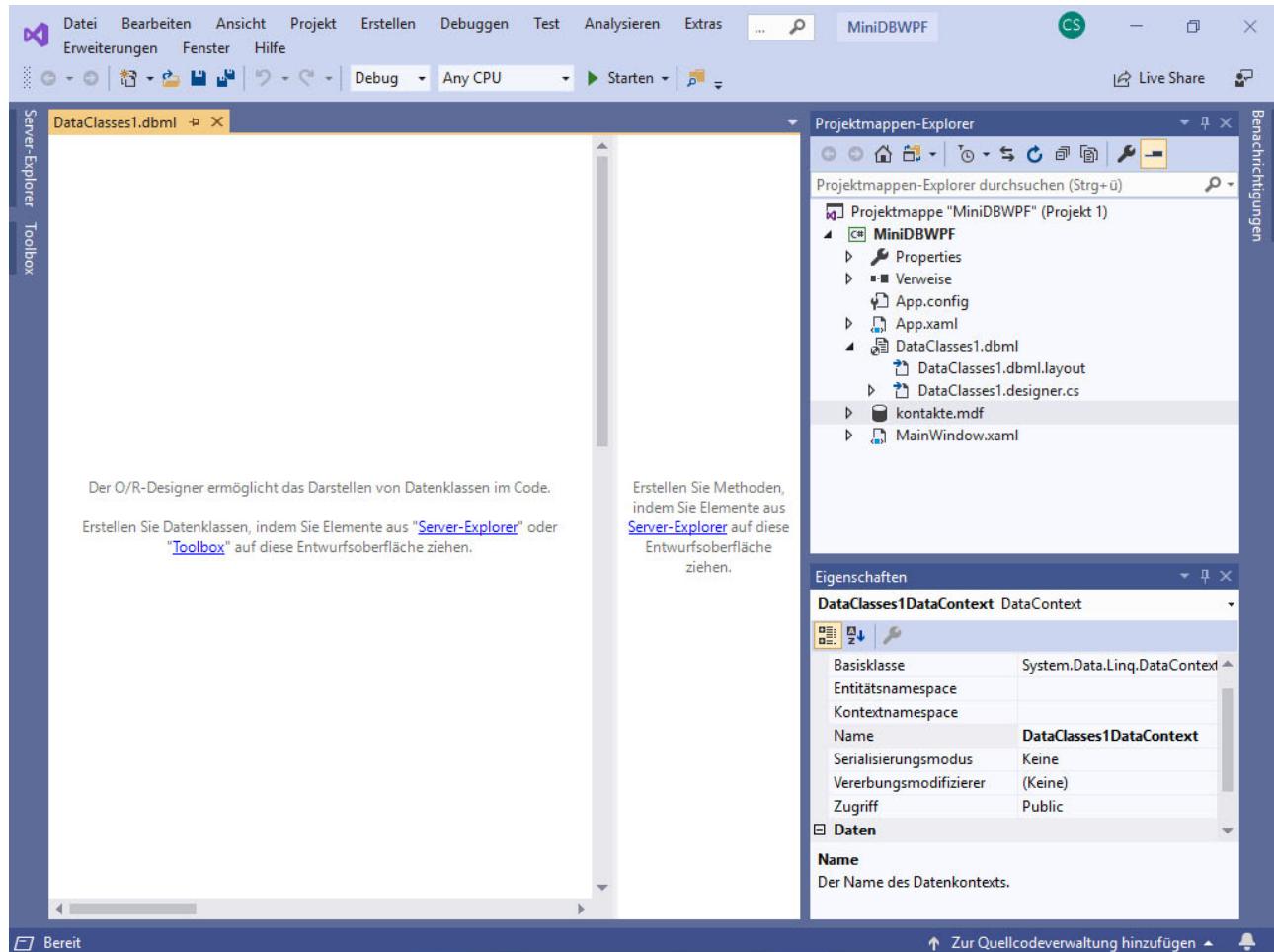


Abb. 4.6: Der Object Relational Designer

Visual Studio öffnet den Object Relational Designer.

Um die Entitätsklasse zu erstellen, ziehen Sie jetzt einfach die gewünschte Datenbanktabelle aus dem Server-Explorer in den linken Bereich des Object Relational Designers. Probieren Sie das bitte mit unserer Tabelle für die Kontakte aus.

Visual Studio fragt Sie zunächst wie beim Anlegen einer Datenquelle, ob die Datendatei in den Projektordner kopiert werden soll. Bestätigen Sie diese Abfrage mit Ja.

Hinweis:

Sie müssen die Datei nicht kopieren. Denken Sie dann aber bitte daran, dass sie nicht im Projektordner liegt.

Visual Studio ruft Informationen aus der Datenbank ab und erstellt die Entitätsklasse. Das kann durchaus einige Zeit dauern.

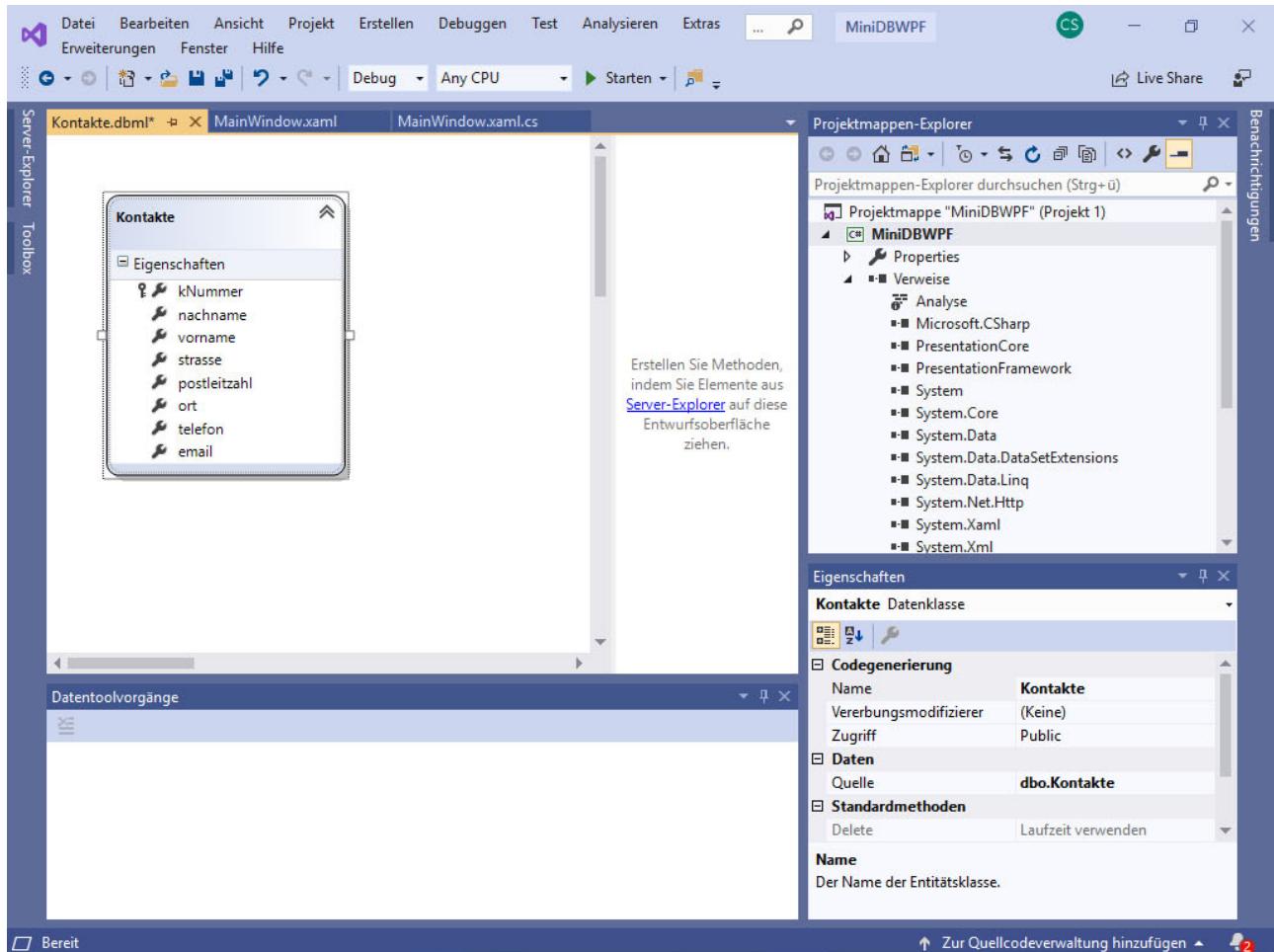


Abb. 4.7: Die Entitätsklasse für die Kontakte

In der Entitätsklasse **Kontakte** finden Sie jetzt alle Spalten aus der Datenbanktabelle für die Kontakte wieder.

4.5 Das Programm

Damit können wir uns nun um das eigentliche Programm kümmern.

Legen Sie im ersten Schritt ein DataGrid an und binden Sie die Spalten an die entsprechenden Felder der Entitätsklasse. Der XAML-Code könnte so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<DataGrid x:Name="KontakteGrid" AutoGenerateColumns="False"
SelectionMode="Single" Margin="0,0,0,50">
    <DataGrid.Columns>
        <!-- bitte in einer Zeile eingeben -->
        <DataGridTextColumn Binding="{Binding Path=kNummer}" 
IsReadOnly="True" Header="Kontaktnummer" Width="100"/>
        <DataGridTextColumn Binding="{Binding Path=vorname}" 
Header="Vorname" Width="100"/>
        <DataGridTextColumn Binding="{Binding Path=nachname}" 
Header="Name" Width="100"/>
```

```

<DataGridTextBoxColumn Binding="{Binding Path=postleitzahl}"
Header="PLZ" Width="100"/>
<DataGridTextBoxColumn Binding="{Binding Path=ort}"
Header="Ort" Width="100"/>
<DataGridTextBoxColumn Binding="{Binding Path=telefon}"
Header="Telefon" Width="100"/>
<DataGridTextBoxColumn Binding="{Binding Path=email}"
Header="E-Mail" Width="100"/>
</DataGrid.Columns>
</DataGrid>

```

Code 4.1: Das DataGrid für die Anzeige

Besonderheiten gibt es eigentlich keine. Wir sorgen zunächst dafür, dass im DataGrid nur unsere selbst erstellten Spalten angezeigt werden und maximal eine Zeile markiert werden kann. Danach erstellen wir die einzelnen Spalten, setzen den Text im Spaltenkopf und binden sie an das jeweilige Feld der Entitätsklasse. Außerdem setzen wir jede Spalte auf die Breite 100. Für die Spalte mit der Kontaktnummer verhindern wir Änderungen durch den Anwender, indem wir die Eigenschaft `IsReadOnly` auf `True` setzen.

Setzen Sie nach unten links eine Schaltfläche mit dem Text „Neuer Eintrag“. In die Mitte kommt eine Schaltfläche mit dem Text „Aktualisieren“ und rechts eine Schaltfläche mit dem Text „Löschen“.

Das Formular sollte dann ungefähr so aussehen:

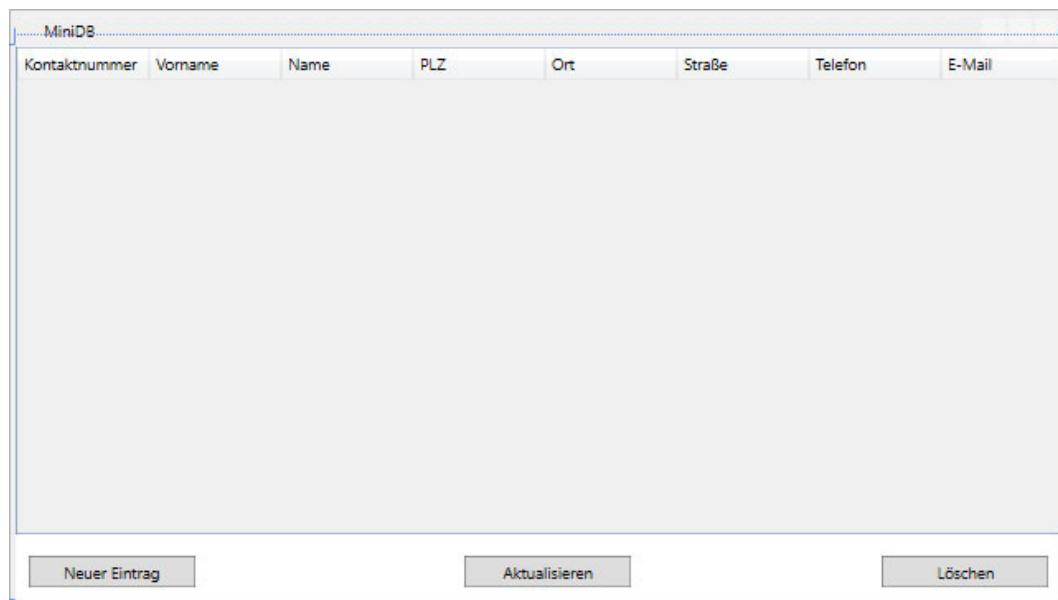


Abb. 4.8: Das Formular

Kommen wir jetzt zur Funktionalität. Im ersten Schritt müssen wir einen Datenkontext für den Zugriff auf das Objektmodell erstellen. Das ist sehr einfach, da Visual Studio eine entsprechende Klasse beim Erstellen der Entitätsklasse gleich mit erstellt hat. Sie heißt in unserem Beispiel `KontakteDataContext`. Über den Datenkontext beschaffen wir uns dann mit einer LINQ-Abfrage die Daten in der Tabelle und setzen sie als Quelle für das DataGrid.

Eine Methode `DatenLaden()`, die entsprechenden Anweisungen ausführt, sieht so aus:

```
private void DatenLaden()
{
    //den Kontext erstellen
    kontext = new KontakteDataContext();
    //die Daten laden
    var kontakte = from kontakt in kontext.Kontakte
                   select kontakt;
    //und als Quelle setzen
    KontakteGrid.ItemsSource = kontakte;
}
```

Code 4.2: Die Methode `DatenLaden()`

Hinweis:

Da wir den Datenkontext gleich noch in anderen Methoden benötigen, haben wir `kontext` als Feld vom Typ `KontakteDataContext` erstellt.

Rufen Sie die Methode bitte nach dem Laden des Fensters in der Methode `Window_Loaded()` auf. Damit werden die Daten nach dem Start der Anwendung beschafft.

Kümmern wir uns jetzt um das Anlegen, Bearbeiten und Löschen von Datensätzen.

Das Anlegen ist mit einer Zeile erledigt. Der Anwender kann ja in der letzten Zeile einen neuen Datensatz erfassen. Wir müssen also lediglich dafür sorgen, dass die Änderungen übernommen werden. Das erledigt die Methode `SubmitChanges()`⁸ für den Datenkontext.

Die komplette Methode für das Anklicken der Schaltfläche sieht so aus:

```
private void ButtonNeu_Click(object sender, RoutedEventArgs e)
{
    //einfach die Änderungen durchführen
    kontext.SubmitChanges();
}
```

Code 4.3: Das Anlegen eines neuen Eintrags

8. *Submit changes* lässt sich etwas holprig mit „reiche Änderungen ein“ übersetzen.

Probieren Sie das bitte aus. Übernehmen Sie den Code. Tragen Sie dann in die letzten Zeile Daten ein und klicken Sie danach auf die Schaltfläche **Neuer Eintrag**. Nach dem Anlegen einige Datensätze könnte die Tabelle zum Beispiel so aussehen:

Kontaktnummer	Vorname	Name	PLZ	Ort	Straße	Telefon	E-Mail
1	Hans	Franz	12345	Meierhausen	Meierweg 13	01234/56789	hans@franz.de
2	Max	Meier	12345	Meierhausen	Maxweg 1	001/002	max@meier.de

Abb. 4.9: Einige Einträge in der Tabelle



Bitte denken Sie daran:

Beim Ausführen des Programms wird automatisch eine weitere Kopie der Datenbankdatei in dem Ordner mit dem ausführbaren Programm erstellt. In dieser Kopie werden dann auch die Datensätze bearbeitet. Dabei gehen bereits erfasste Daten verloren.

Über die Eigenschaft **In Ausgabeverzeichnis kopieren** für die Datenbankdatei können Sie dieses Verhalten gezielt steuern.

In unserem Beispielprojekt, das Sie auf Ihrer Online-Lernplattform finden, werden die Datenbankdateien kopiert, wenn Änderungen erfolgt sind. Damit ist in jedem Fall sichergestellt, dass der Zugriff funktioniert und immer die aktuellste Version benutzt wird.

Für das Bearbeiten und Löschen müssen wir ein wenig mehr Aufwand betreiben. Wir beschaffen uns zuerst die markierte Zeile im DataGridView und selektieren über die Kontaktnummer den entsprechenden Eintrag. Beim Ändern setzen wir die Daten auf die Werte in den einzelnen Spalten und übernehmen die Änderungen. Beim Löschen dagegen rufen wir für die markierte Zeile die Methode `DeleteOnSubmit()` auf und übernehmen danach ebenfalls die Änderungen.

Die Methode für die Änderung von Daten sieht so aus:

```
private void ButtonAktualisieren_Click(object sender,
RoutedEventArgs e)
{
    //die markierte Zeile setzen
    Kontakte kontaktZeile = KontakteGrid.SelectedItem as Kontakte;
    //die Nummer beschaffen
    int suche = kontaktZeile.kNummer;
    //den Eintrag selektieren
    Kontakte kontakt = (from eintrag in kontext.Kontakte
        where eintrag.kNummer == kontaktZeile.kNummer
        select eintrag).Single();
    //die Daten setzen
    kontakt.nachname = kontaktZeile.nachname;
    kontakt.vorname = kontaktZeile.vorname;
    kontakt.postleitzahl = kontaktZeile.postleitzahl;
    kontakt.ort = kontaktZeile.ort;
    kontakt.strasse = kontaktZeile.strasse;
    kontakt.telefon = kontaktZeile.telefon;
    kontakt.email = kontaktZeile.email;

    //und ändern
    kontext.SubmitChanges();
}
```

Code 4.4: Die Methode zum Ändern von Daten

Mit der Anweisung

```
Kontakte kontaktZeile = KontakteGrid.SelectedItem as Kontakte;
```

beschaffen wir uns die Daten aus der markierten Zeile im DataGrid und speichern sie in einer Instanz unserer Entitätsklasse. Dazu müssen wir die Zeile in den Typ Kontakte umwandeln.

Dann beschaffen wir uns die Kontaktnummer aus der markierten Zeile und selektieren mit der Anweisung

```
Kontakte kontakt = (from eintrag in kontext.Kontakte
    where eintrag.kNummer == kontaktZeile.kNummer
    select eintrag).Single();
```

den entsprechenden Eintrag. Dabei ist wichtig, dass Sie über die Methode `Single()` sicherstellen, dass wirklich nur ein Eintrag geliefert wird. Andernfalls scheitert auch die Zuweisung auf eine Instanz der Entitätsklasse.

Anschließend weisen wir den Feldern der Entitätsklasse die Daten aus der markierten Zeile zu. Dabei gibt es keine Besonderheiten.

Mit der letzten Anweisung

```
kontext.SubmitChanges();
```

schließlich lassen wir die Änderungen dann durchführen.

Das Löschen eines Datensatzes erfolgt mit einer ähnlichen Technik. Die Anweisungen für die Methode sehen so aus:

```
//die markierte Zeile setzen
Kontakte kontaktZeile = KontakteGrid.SelectedItem as Kontakte;
//die Nummer beschaffen
int suche = kontaktZeile.kNummer;
//den Eintrag selektieren
Kontakte kontakt = (from eintrag in kontext.Kontakte
    where eintrag.kNummer == kontaktZeile.kNummer
    select eintrag).Single();
//die Daten löschen
kontext.Kontakte.DeleteOnSubmit(kontakt);

//und ändern
kontext.SubmitChanges();
//und neu laden
DatenLaden();
```

Code 4.5: Die Anweisungen für die Methode zum Löschen

Zunächst beschaffen wir uns wieder über die markierte Zeile den Datensatz. Dann löschen wir ihn mit der Anweisung

```
kontext.Kontakte.DeleteOnSubmit(kontakt);
```

und führen die Änderung durch. Danach laden wir die Daten neu, damit der gelöschte Datensatz nicht mehr angezeigt wird.

Damit wollen wir die Arbeit an der Datenbankanwendung beenden – auch wenn es noch reichlich Möglichkeiten für Verbesserungen gibt. So fehlen zum Beispiel Abfragen, ob alle erforderlichen Daten erfasst sind oder ob beim Aufruf der Methoden zum Ändern und Löschen überhaupt eine Zeile markiert ist. Die entsprechenden Erweiterungen können Sie ja selbst programmieren.

Zusammenfassung

Der Zugriff auf die Daten erfolgt bei LINQ to SQL über das Objektmodell. Es bildet das Datenmodell ab.

Die Abbildung einer Datenbanktabelle im Objektmodell erfolgt über eine Entitätsklasse.

Für den Zugriff auf das Objektmodell wird ein Datenkontext benötigt.

Die LINQ to SQL-Tools müssen Sie getrennt installieren.

Mit der Methode `SubmitChanges()` für den Datenkontext können Sie Änderungen übernehmen.

Mit der Methode `DeleteOnSubmit()` für den Datenkontext löschen Sie Einträge.

Aufgaben zur Selbstüberprüfung

- 4.1 Beschreiben Sie, wie Sie eine Entitätsklasse mit Visual Studio erstellen.
- 4.2 Mit welcher Eigenschaft können Sie für ein DataGrid festlegen, dass immer nur eine Zeile markiert werden kann?
- 4.3 Sie haben eine Entitätsklasse mit dem Namen `meineDaten` erstellt. Wie heißt der entsprechende Datenkontext? Wie wird die Klasse für diesen Datenkontext angelegt?
- 4.4 Sie haben sich in einem DataGrid die markierte Zeile beschafft und in den passenden Typ für die Entitätsklasse umgewandelt. Worauf müssen Sie beim Erstellen der LINQ-Abfrage unbedingt achten, damit die Abfragevariable im richtigen Typ erstellt werden kann?

Schlussbetrachtung

In diesem Studienheft haben Sie sich mit LINQ beschäftigt – einer universellen Abfragesprache, die fester Bestandteil des .NET Frameworks ist. Sie wissen jetzt, wie LINQ-Abfragen grundsätzlich aufgebaut sind und welche Möglichkeiten Ihnen die Abfragen bieten. Außerdem haben Sie über LINQ to XML und LINQ to SQL auf XML-Dateien und Datenbanktabellen zugegriffen.

Nutzen Sie die Beispiele aus diesem Studienheft und bauen Sie sie weiter aus. Sie können ja zum Beispiel die kleine Datenbankanwendung robuster machen und weitere Funktionen wie eine Suche ergänzen.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 LINQ lässt sich bei jedem Objekt einsetzen, dass die Schnittstelle `IEnumerable<T>` oder `IEnumerable` unterstützt.
- 1.2 Zu den LINQ-Providern gehören:
 - LINQ to SQL für den Zugriff auf SQL Server-Datenbanken,
 - LINQ to XML für den Zugriff auf XML-Dateien,
 - LINQ to Objects für den direkten Zugriff auf Objekte wie Arrays oder Auflistungen
 - LINQ to DataSet für den Zugriff auf DataSets aus dem ADO.NET-Framework oder
 - LINQ to Entities für den Zugriff auf relationale Datenstrukturen aus dem ADO.NET-Framework.

Für die richtige Lösung reicht es aus, wenn Sie drei Provider genannt haben.

- 1.3 Die Syntax unterscheidet sich grundsätzlich nicht.

Kapitel 2

- 2.1 Eine LINQ-Abfrage besteht aus drei verschiedenen Teilen:
 1. das Erstellen einer Datenquelle,
 2. dem Erstellen der Abfrage und
 3. dem Ausführen der Abfrage.
- 2.2 Das Schlüsselwort `from` legt fest, woher die Daten stammen sollen.
- 2.3 Einen Typ müssen Sie für die Bereichsvariable nicht angeben. Er wird aus der Datenquelle abgeleitet.
- 2.4 Die Abfrage könnte so aussehen:

```
var abfrage =
  from zahl in zahlen
  where zahl > 10 && zahl < 100
  select zahl;
```

- 2.5 Die Abfrage könnte so aussehen:

```
var abfrage =
  from zeichenkette in zeichenketten
  where zeichenkette[0] == 'A'
  select zeichenkette;
```

- 2.6 Zum Sortieren verwenden Sie das Schlüsselwort `orderby`.
- 2.7 Das Zeichen `=>` ist der Lambda-Operator. Er lässt sich als „wird zu“ lesen.
- 2.8 Der Operator `Distinct()` liefert Ihnen eine Liste ohne Duplikate.

Kapitel 3

- 3.1 Die Anweisung könnte so aussehen:

```
XElement xmlElement =
 XElement.Load("c:\\meinOrdner\\meineDatei.xml");
```

Der Bezeichner ist beliebig.

- 3.2 Die Abfrage könnte so aussehen:

```
var eintraege = from eintrag in
xmlElement.Elements(("meinEintrag"))
select eintrag;
```

Die Bezeichner sind dabei beliebig.

- 3.3 Die Abfrage und die Schleife könnten so aussehen:

```
var eintraege = from eintrag in
xmlElement.Elements(("eintrag"))
select new
{
    Vorname = eintrag.Element("vorname").Value,
    Nachname = eintrag.Element("nachname").Value,
};

foreach (var eintrag in eintraege)
    Console.WriteLine("Vorname: {0} Nachname: {1}",
        eintrag.Vorname, eintrag.Nachname);
```

Die Bezeichner können auch abweichen.

- 3.4 Zum Anhängen verwenden Sie die Methode `Add()`.
- 3.5 Sie können keine Knoten aus einer Abfrage löschen, die Sie gerade verwenden.

Kapitel 4

- 4.1 Um eine Entitätsklasse zu erstellen, fügen Sie über die Funktion **Projekt/Neues Element hinzufügen...** ein neues Element ein. Im Fenster **Neues Element hinzufügen** markieren Sie den Eintrag **LINQ to SQL-Klassen**.

Geben Sie einen Namen ein und klicken Sie auf **Hinzufügen**. Visual Studio öffnet den Object Relational Designer. Ziehen Sie hier die gewünschte Datenbanktabelle aus dem Server-Explorer in den linken Bereich des Object Relational Designer.

- 4.2 Um festzulegen, dass immer nur eine Zeile in einem DataGridView verwendet werden kann, verwenden Sie die Eigenschaft SelectionMode.
- 4.3 Der Datenkontext heißt `meineDatenDataContext`. Die Klasse wird automatisch beim Erstellen der Entität von Visual Studio angelegt.
- 4.4 Sie müssen mit der Methode `Single()` sicherstellen, dass nur ein Ergebnis geliefert wird. Andernfalls ist die Umwandlung in den Typ der Entitätsklasse nicht möglich.

B. Glossar

Abfragesyntax	Die Abfragesyntax ist eine Form, LINQ-Abfragen zu erstellen. Eine typische Abfrage in der Abfragesyntax sieht so aus: <code>var abfrage = from zeichen in zeichenkette where zeichen == suche select zeichen;</code>
ADO.NET	ADO.NET ist eine Schnittstelle für den Zugriff auf Daten in Datenbanken. Sie gehört zum .NET Framework.
Datenbank	Eine Datenbank ist eine strukturierte Sammlung von Daten in elektronisch verarbeitbarer Form.
Datenkontext	Der Datenkontext wird für den Zugriff auf das Objektmodell eingesetzt.
Datensatz	Ein Datensatz speichert logisch zusammengehörige Informationen in einer Datenbank beziehungsweise einer Datenbanktabelle.
Entität	Eine Entität beschreibt ein Objekt. Zu einer Entität gehören Attribute, die das Objekt beschreiben.
Entitätsklasse	Eine Entitätsklasse bildet eine Datenbanktabelle im Objektmodell ab.
Generic	Generics sind Methoden, Klassen, Strukturen und so weiter, die Platzhalter für die Typen benutzen, die von ihnen verarbeitet werden. Die konkreten Typen für die Platzhalter werden erst zur Laufzeit des Programms eingesetzt.
Generika	siehe Generic. Der Begriff wird auch für die Nachbildung von Medikamenten benutzt.
Knoten	Als Knoten wird ein Unterelement in einer XML-Datei bezeichnet. Die Bezeichnung gilt aber auch generell für Elemente in einer Struktur.
Lambda-Operator =>	Der Lambda-Operator => ist Teil eines Lambda-Ausdrucks. Er lässt sich als „wird zu“ lesen.
LINQ	LINQ ist eine universelle Abfragesprache, die fester Bestandteil des .NET Frameworks ist.
LINQ to SQL	LINQ to SQL ist ein Programmierschnittstelle, die die Verarbeitung von SQL-Datenbanktabellen ermöglicht.

LINQ to XML	LINQ to XML ist ein Programmierschnittstelle, die die Verarbeitung von XML-Daten ermöglicht. Die Daten werden dabei komplett im Arbeitsspeicher gehalten.
LINQ-Provider	LINQ-Provider ermöglichen den Zugriff auf Daten über LINQ. Zu den Providern gehören zum Beispiel LINQ to XML und LINQ to SQL.
Methodensyntax	Die Methodensyntax ist eine Form, LINQ-Abfragen zu erstellen. Eine typische Abfrage in der Methodensyntax sieht so aus: <pre>var abfrage = zeichenkette.Where(zeichen => zeichen == suche);</pre>
Object Relational Designer	Der Object Relational Designer ist ein Werkzeug für die Arbeit mit LINQ to SQL. Er ermöglicht unter anderem das Erstellen der Entitätsklassen.
Objektinitialisierer	Über einen Objektinitialisierer können Sie Instanzen einer Klasse erzeugen und direkt die Werte von Eigenschaften setzen. Ein Objektinitialisierer entspricht in der Wirkung einem Konstruktor.
Objektmodell	Das Objektmodell bildet bei LINQ to SQL das Datenmodell ab.
Primärschlüssel	Über den Primärschlüssel kann ein Datensatz in einer Datenbanktabelle eindeutig identifiziert werden.
Server-Explorer	Der Server-Explorer ist ein Teil der Oberfläche von Visual Studio. Über den Server-Explorer können Sie zum Beispiel Verbindungen zu einer Datenbank herstellen.
Tag (XML)	Ein XML-Tag dient zur Beschreibung und Strukturierung von Informationen in einer XML-Datei. Jedes Tag besteht in der Regel aus dem Start-Tag und dem End-Tag.
Typableitung	Siehe Typinferenz.
Typinferenz	Bei der Typinferenz kann der Datentyp einer Variablen aus der ersten Zuweisung abgeleitet werden.
XML	XML (<i>eXtensible Markup Language</i>) ist ein textbasiertes, strukturiertes Format für die Beschreibung von Daten.
Zahlenformatzeichenfolge	Über die Zahlenformatzeichenfolge können Sie dem Compiler mitteilen, in welcher Form ein numerischer Wert ausgegeben werden soll.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch.*

Spracheinführung, Objektorientierung, Programmiertechniken.
8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019.* Ideal für Programmieranfänger.
6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 2.1	Die drei Teile einer LINQ-Abfrage	7
Abb. 2.2	Das Programm im Einsatz.....	20
Abb. 2.3	Die Oberfläche für die Textanalyse.....	21
Abb. 3.1	Die fertige Anwendung	36
Abb. 4.1	Die Datenbankanwendung	39
Abb. 4.2	Das Daten- und Objektmodell	39
Abb. 4.3	Die aktualisierte Datenbank	41
Abb. 4.4	Der Bereich Codetools	42
Abb. 4.5	Der Eintrag LINQ to SQL-Klassen im Bereich Daten (in der Mitte der Abbildung am Mauszeiger)	43
Abb. 4.6	Der Object Relational Designer	44
Abb. 4.7	Die Entitätsklasse für die Kontakte	45
Abb. 4.8	Das Formular	46
Abb. 4.9	Einige Einträge in der Tabelle	48

E. Tabellenverzeichnis

Tab. 4.1 Die Spalten für die Datenbanktabelle 40

F. Codeverzeichnis

Code 1.1	Ein einfaches Beispiel	4
Code 1.2	Ein weiteres Beispiel	6
Code 2.1	Verzögerte Ausführung der Abfrage	9
Code 2.2	Eine ungültige Datenquelle (der Code lässt sich nicht übersetzen)	9
Code 2.3	Einsatz einer Methode in einem Filter	11
Code 2.4	Geänderter Typ	12
Code 2.5	Eine sehr einfache Klasse für Autos	14
Code 2.6	Der Zugriff auf die Autoliste	15
Code 2.7	Weitere Abfrageoperatoren im Einsatz	19
Code 2.8	Der Code zum Öffnen einer Datei	21
Code 2.9	Die Methode für das Anklicken der Startschaltfläche	24
Code 2.10	Eine generische Methode zur Addition	25
Code 2.11	Eine generische Liste über die Klasse List	26
Code 2.12	Eine lange Liste	27
Code 3.1	Das Lesen von XML-Daten über die Klasse XElement	30
Code 3.2	Trennen von Elementen	32
Code 3.3	Gezielter Zugriff auf einen Eintrag	33
Code 3.4	Das Erzeugen von neuen Strukturen und Elementen	34
Code 3.5	Das DataGridView für unsere Anwendung	36
Code 3.6	Die Methoden für die Schaltflächen	37
Code 4.1	Das DataGridView für die Anzeige	46
Code 4.2	Die Methode DatenLaden()	47
Code 4.3	Das Anlegen eines neuen Eintrags	47
Code 4.4	Die Methode zum Ändern von Daten	49
Code 4.5	Die Anweisungen für die Methode zum Löschen	50

G. Sachwortverzeichnis

A

Abfragesyntax 13

B

Bereichsvariable 10

D

Daten gruppieren 14

Datenkontext 46

Datenquelle 9

E

Entität 39

Entitätsklasse 39

G

Generics 24

K

Knoten 29

L

Lambda-Ausdruck 13

Lambda-Operator 13

LINQ 3

LINQ to DataSet 3

LINQ to Entities 3

LINQ to Objects 3

LINQ to XML 29

LINQ-Abfrage 4

LINQ-Anbieter 3

LINQ-Provider 3

M

Methodensyntax 13

O

Object Relational Designer 41

Objektinitialisierer 16

Objektmodell 39

S

Schnittstelle IENumerable 9, 9

T

Tag 29

Tools für LINQ to SQL 41

Typinferenz 5

Typparameter 25

U

Unterelement 29

W

Wurzelelement 29

X

XML 29

H. Einsendeaufgabe

Datenabfragen mit LINQ

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Code:
CSP23D-XX1-N01

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

- Wann wird die Abfrage im folgenden Codefragment ausgeführt?

```
int[] zahlen = {4, 99, 13, 3, 9, 12, 77, 43};

//die Abfrage
var abfrage =
    from zahl in zahlen
    where zahl % 3 == 0
    orderby zahl
    select zahl;

//die Ausgabe
foreach (int wert in abfrage)
    Console.WriteLine(wert + " ");
```

5 Pkt.

- Schreiben Sie die folgende Abfrage aus der Abfragesyntax in die Methodensyntax um:

```
var werte =
    from wert in liste
    where wert < 100 && wert > 10
    select wert;
```

5 Pkt.

- Erstellen Sie eine Abfrage, die aus der folgenden Datenquelle alle Einträge selektiert, bei denen der Bestellwert größer ist als 100. Lassen Sie die Ergebnisse gruppiert nach der Warengruppe ausgeben.

```
Lieferung[] lieferungen =
{
    new Lieferung {Warengruppe = "A", Name = "Papier",
    Bestellwert = 20},
    new Lieferung {Warengruppe = "A", Name = "Papier",
    Bestellwert = 250},
    new Lieferung {Warengruppe = "B", Name = "Hardware",
    Bestellwert = 30},
```

```
new Lieferung {Warengruppe = "C", Name = "Software",
Bestellwert = 600},
new Lieferung {Warengruppe = "C", Name = "Software",
Bestellwert = 650},
};
```

Notieren Sie dazu die nötigen Anweisungen.

10 Pkt.

4. Erstellen Sie ein Programm, das mindestens 20 zufällig erzeugte Zahlen im Bereich von 1 bis 100 in einem Listenfeld anzeigt. Beim Anklicken einer Schaltfläche sollen die Zahlen sortiert und alle Duplikate entfernt werden. Verwenden Sie zur Anzeige dieser Liste ein weiteres Listenfeld. Verwenden Sie zum Erstellen der Liste und zum Entfernen der Duplikate eine LINQ-Abfrage.

Ob Sie eine Windows Forms- oder eine WPF-Anwendung erstellen, bleibt Ihnen überlassen.

30 Pkt.

5. Erstellen Sie eine WPF-Anwendung, die Daten aus einer XML-Datei lesen, verarbeiten und speichern kann. Der Anwender soll dabei für eine Person den Vor- und Nachnamen sowie das Alter angeben können.

Lassen Sie die Daten in einem DataGrid anzeigen. Der Anwender soll dabei folgende Möglichkeiten haben:

- Erzeugen einer neuen Datei mit einem leeren Eintrag,
- Anhängen von neuen Daten an bereits vorhandene Daten,
- Überarbeitung vorhandener Daten,
- Löschen von Daten und
- Suchen nach Daten über den Nachnamen.

Den Pfad und den Namen der Datei können Sie fest im Programm hinterlegen.

Sorgen Sie dafür, dass das Programm betriebssicher ist – zum Beispiel, indem Sie vor dem Speichern prüfen, ob überhaupt eine Datei geladen wurde.

Wie Sie bei der Lösung vorgehen, bleibt Ihnen überlassen. Bitte dokumentieren Sie Ihren Ansatz aber.

Ein Hinweis zum Löschen:

Sie können sich den markierten Eintrag über die Eigenschaft `SelectedItem` beschaffen und ihn in den Typ `XElement` umbauen. Danach können Sie auf die einzelnen Werte zugreifen.

50 Pkt.

Gesamt: 100 Pkt.

Objektorientierte Software-Entwicklung mit C#

Universal Windows Platform Apps erstellen

Das Studienheft und seine Teile sind urheberrechtlich geschützt.
Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist
nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des
Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen
via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern
(und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

1219N01

CSHP24D

Objektorientierte Software-Entwicklung mit C#

Universal Windows Platform Apps erstellen

Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein. Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Universal Windows Platform Apps erstellen

Inhaltsverzeichnis

Einleitung	1
1 Grundlagen	3
1.1 Was sind Universal Windows Platform Apps?	3
1.2 Die Gestaltung von Universal Windows Platform Apps	4
1.3 Technische Besonderheiten von Universal Windows Platform Apps....	6
1.4 Die Entwicklung von Universal Windows Platform Apps	8
Zusammenfassung	8
2 Eine erste App	10
2.1 Das Projekt erstellen	10
2.2 Test auf unterschiedlichen Geräten.....	16
Zusammenfassung	19
3 Ein Taschenrechner als App	20
3.1 Die Grundfunktionen	20
3.2 Die Oberfläche des Taschenrechners	21
3.3 Die Logik	26
3.4 Reaktion auf das Beenden durch das System	31
3.5 Ein wenig Feinschliff	36
3.6 Veröffentlichen der App	38
Zusammenfassung	44
4 Ein Notizbuch als App	46
4.1 Die Grundfunktionen	46
4.2 Die Oberfläche	46
4.3 Die Logik	50
Zusammenfassung	55
Schlussbetrachtung	57

Anhang

A.	Lösungen der Aufgaben zur Selbstüberprüfung	58
B.	Glossar	60
C.	Literaturverzeichnis	62
D.	Abbildungsverzeichnis	63
E.	Codeverzeichnis	65
F.	Sachwortverzeichnis	66
G.	Einsendeaufgabe	67

Einleitung

In diesem Studienheft werden wir uns mit der Entwicklung von Universal Windows Platform Apps – auch universelle Windows Apps oder einfach nur Windows Apps genannt – beschäftigen. Dabei handelt es sich um spezielle Anwendungen, die auf verschiedenen Plattformen wie einem Desktop-Rechner, einem Tablet oder einem Smartphone unter Windows 10 eingesetzt werden können. Zuerst sehen wir uns an, was die Besonderheiten von Universal Windows Platform Apps sind und wie sie sich von „normalen“ Anwendungen für den Windows-Desktop unterscheiden. Danach erstellen wir eine ganz einfache erste Universal Windows Platform App. Wir programmieren aber auch zwei etwas anspruchsvollere Anwendungen – einen Taschenrechner und eine Art Zettelkasten.

Im Einzelnen lernen Sie in diesem Studienheft:

- was Universal Windows Platform Apps sind,
- wie sie sich von normalen Windows-Anwendungen unterscheiden,
- welche Besonderheiten Sie bei der Entwicklung berücksichtigen müssen,
- wie Sie eine Entwicklerlizenz erwerben,
- wie Sie eine erste einfache App erstellen,
- wie Sie die App im Simulator testen,
- wie Sie einen einfachen Taschenrechner als App umsetzen,
- wie Sie auf das Beenden einer App durch das System reagieren,
- wie Sie Daten in einer App in einer Datei speichern und
- wie Sie Daten in einer App aus einer Datei laden.

Ein wichtiger Hinweis:

Universal Windows Platform Apps laufen nur unter Windows 10 und können nur mit Windows 10 entwickelt werden. Wenn Sie dieses Betriebssystem nicht auf Ihrem Rechner installiert haben, können Sie eine virtuelle Maschine wie den VMware Workstation Player mit einer Testversion von Windows 10 einsetzen. Den VMware Workstation Player können Sie für den privaten Gebrauch kostenlos auf den Internet-Seiten von VMware herunterladen. Die Testversionen von Windows 10 finden Sie im Download-Bereich auf den Internet-Seiten von Microsoft.

Christoph Siebeck

1 Grundlagen

In diesem Kapitel beschäftigen wir uns mit Grundlagen der Universal Windows Platform Apps. Sie erfahren, was Universal Windows Platform Apps überhaupt sind, welche Besonderheiten es bei der Gestaltung gibt und worauf Sie bei der Entwicklung von Universal Windows Platform Apps achten müssen.

1.1 Was sind Universal Windows Platform Apps?

Universal Windows Platform Apps sind Anwendungen, die sich auf verschiedenen Plattformen einsetzen lassen – zum Beispiel einem Tablet Computer, einem Smartphone, aber auch auf einem normalen Rechner.

App steht als Abkürzung für *Application*.



Universal Windows Platform Apps laufen ausschließlich unter Windows 10.

Mit Windows 8 oder anderen älteren Windows-Versionen können Sie Universal Windows Platform Apps nicht einsetzen. Auch für die Entwicklung ist Windows 10 erforderlich.



Da mobile Geräte in der Regel über berührungsempfindliche Bildschirme bedient werden, arbeiten Universal Windows Platform Apps mit anderen Bedienkonzepten als klassische Anwendungen. So gibt es zum Beispiel Bedienelemente wie eine Symbolleiste oder ein Menüband in dieser Form nicht.

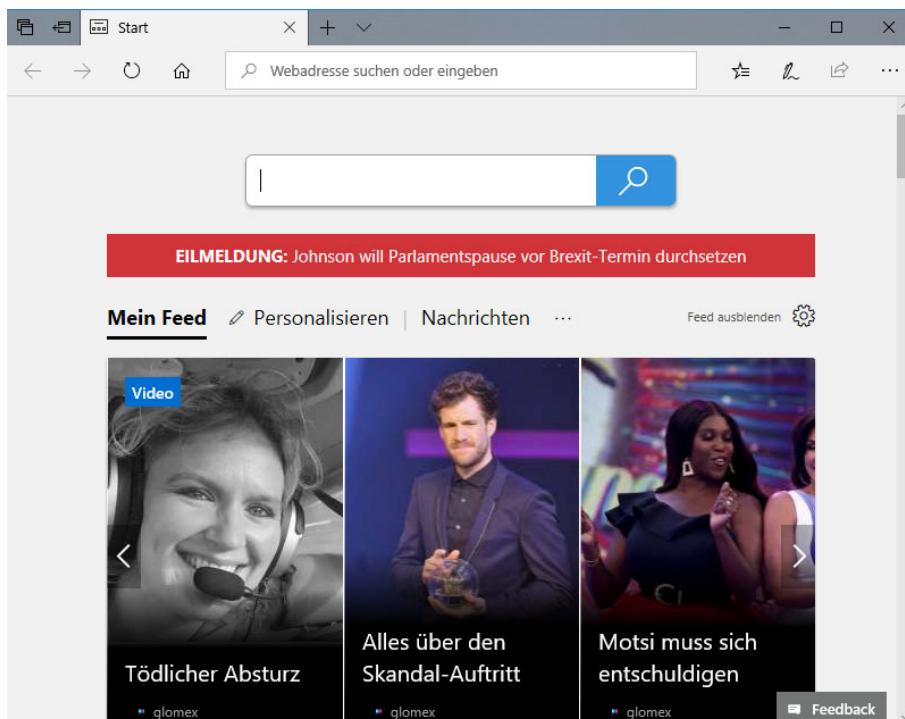


Abb. 1.1: Typische Universal Windows Platform App (hier: Edge von Microsoft)

Grundsätzlich lassen sich Universal Windows Platform Apps aber auch mit der Maus und der Tastatur bedienen. Das ist zum Beispiel dann wichtig, wenn Sie sie auf einem klassischen Computer ohne berührungssempfindlichen Bildschirm einsetzen.

Der Vertrieb der Universal Windows Platform Apps erfolgt über den Microsoft Store. Dabei handelt es sich um eine Art virtuellen Marktplatz im Internet, auf dem unterschiedlichste Apps angeboten werden. Der Microsoft Store kann direkt über Windows aufgerufen werden und steht damit grundsätzlich allen Windows-Anwendern zur Verfügung. Zum Teil sind die Apps auf dem Marktplatz auch kostenlos erhältlich.

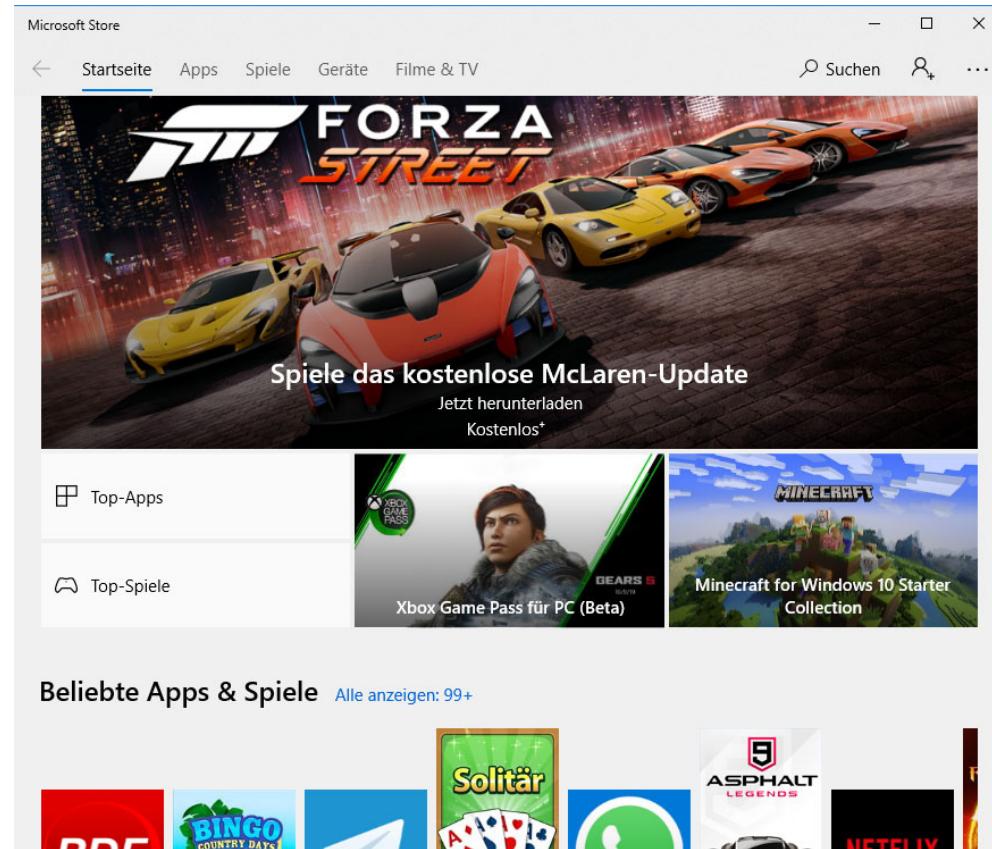


Abb. 1.2: Der Microsoft Store

1.2 Die Gestaltung von Universal Windows Platform Apps

Universal Windows Platform Apps weisen im Vergleich mit Windows-Forms-Anwendungen und auch WPF-Anwendungen einige Besonderheiten auf:

- Die Basis einer App bilden Seiten.

Die Darstellung der Inhalte erfolgt immer auf einer oder mehreren Seiten. Jede Seite übernimmt dabei eine eindeutige Aufgabe – zum Beispiel die Anzeige von vorhandenen Dateien und das Laden einer Datei. Zwischen den Seiten einer App kann der Anwender hin und her wechseln.

- Die App soll möglichst einfach und sachlich erscheinen.

Eine App soll so selbsterklärend wie eben möglich sein. Im Mittelpunkt steht der Inhalt – und nicht die Gestaltung. Dazu wird nicht nur die Oberfläche sehr sachlich gehalten, sondern auch Bedienelemente wie Symbolleiste oder Menüleisten sollen nicht verwendet werden. Auf grafische Spielereien soll komplett verzichtet werden. Microsoft wünscht aber ausdrücklich den Einsatz von Animationen zum Beispiel für Seitenübergänge.

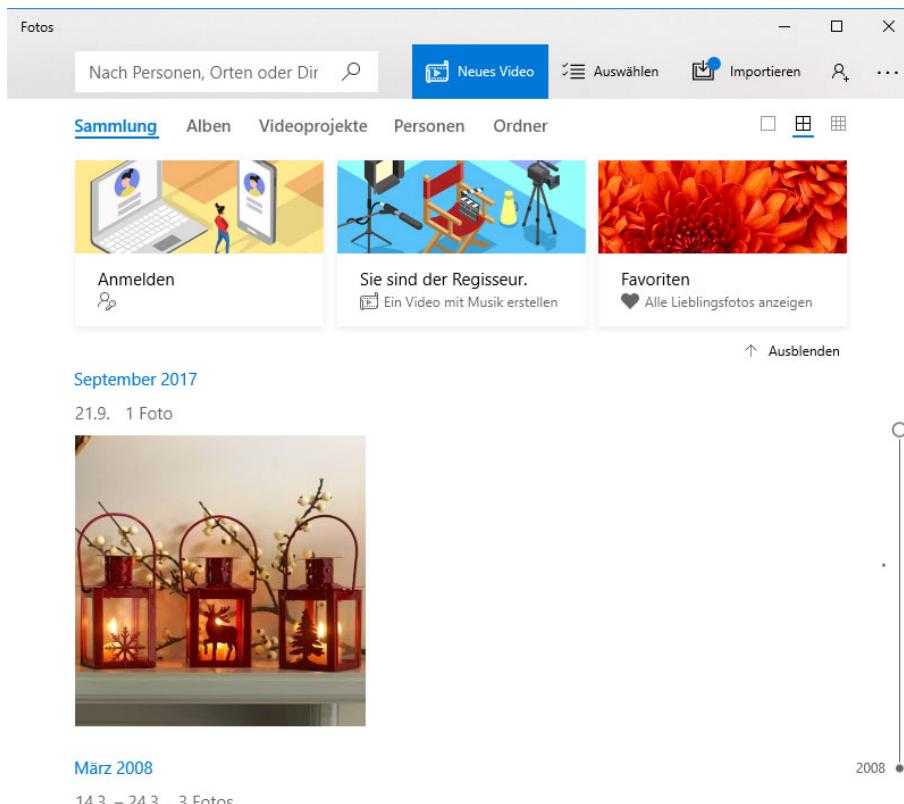


Abb. 1.3: Eine einfache und sachliche App (hier die Fotos von Windows 10)

- Die App soll einfach über Gesten bedient werden können.

Die Bedienelemente sollten daher nicht zu klein sein oder zu eng nebeneinanderliegen. Denn hier kann bei der Bedienung mit den Fingern unter Umständen nicht mehr exakt genug gezielt werden.

Die Bedienung soll nach Möglichkeit direkt über Elemente auf einer Seite erfolgen. Daher fallen auch viele klassische Menüleisten, Menübänder und Symbolleisten aus. Sie eignen sich für die Bedienung über Gesten nicht, weil die Elemente zu nah beieinanderliegen, oder sind in der Anwendung zu kompliziert. Als Ersatz kommen aber einige neue Elemente wie die Befehlsleiste zum Einsatz.

- Die App soll schnell und flüssig arbeiten.

Der Benutzer soll nie das Gefühl haben, auf die App warten zu müssen. Daher sollte der Anwender bei jeder Aktion, die er über die Oberfläche startet, nach Möglichkeit Feedback erhalten – zum Beispiel in Form einer Animation oder eines Seitenwechsels. Dadurch weiß der Anwender auch, dass die App seine Bedienkommandos angenommen hat.

1.3 Technische Besonderheiten von Universal Windows Platform Apps

Universal Windows Platform Apps weisen aber nicht nur in der Erscheinung einige typische Besonderheiten auf. Auch hinter den Kulissen gibt es zum Teil erhebliche Unterschiede zu Windows-Forms- und WPF-Anwendungen, die für Sie als Programmierer wichtig sind.

- Eine App muss verschiedene Darstellungen unterstützen.

Sie können nicht wissen, ob Ihre App auf einem normalen Rechner mit einem großen Monitor, einem Tablet oder einem Smartphone gestartet wird. Ihre App muss daher mit unterschiedlichsten Auflösungen arbeiten können. Zusätzlich müssen Sie für mobile Geräte auch noch Hoch- und Querformat unterstützen.

- Eine App hat nur eingeschränkten Zugriff auf das System.

Universal Windows Platform Apps arbeiten in einer Sandbox – einem streng abgeschirmten Bereich des Systems, der nur der Anwendung selbst zur Verfügung steht. Auf andere Bereiche des Systems hat die App nur dann Zugriff, wenn der Benutzer das ausdrücklich genehmigt. Als Anwender können Sie so ziemlich sicher sein, dass eine App nur solche Aktionen ausführen kann, die Sie erlaubt haben.

Als Programmierer haben Sie damit aber nicht mehr kompletten Zugriff auf die Systemressourcen. So können Sie zum Beispiel nicht mehr ohne Weiteres Daten aus dem Internet herunterladen oder die Kamera eines Smartphones nutzen.

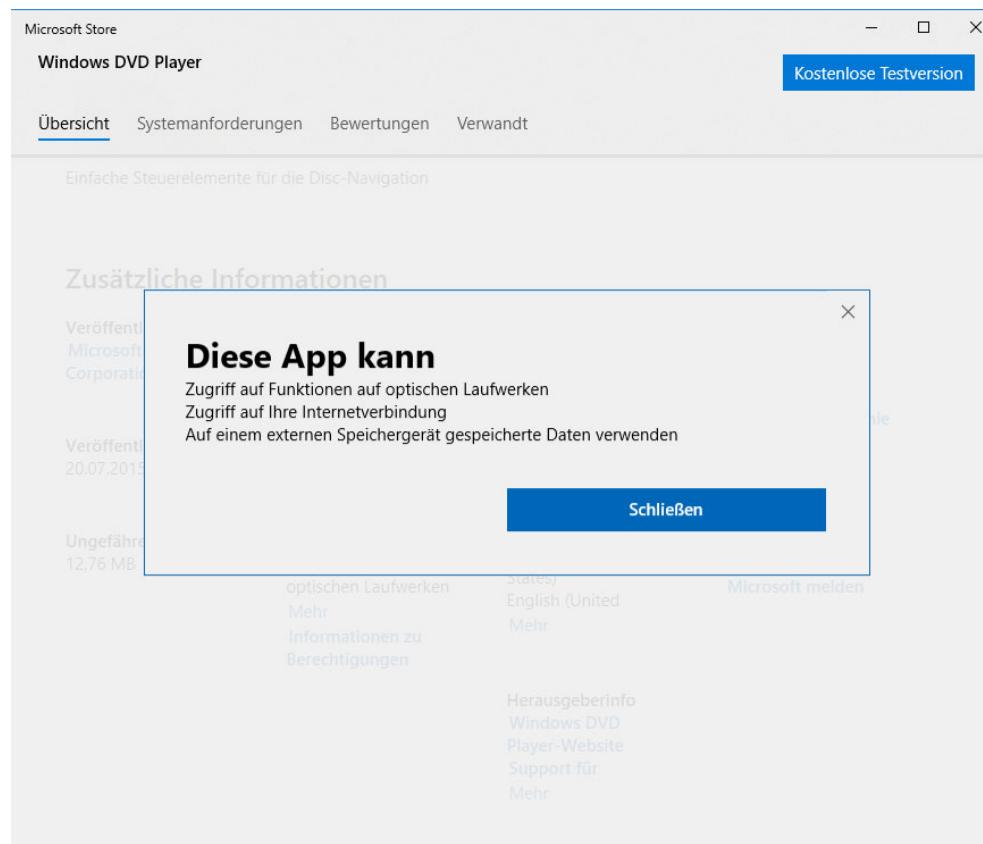


Abb. 1.4: Die Berechtigungen für eine App

Die Zugriffsregeln werden durch Capabilities^{a)} beschrieben.



- a) *Capability* bedeutet übersetzt „Fähigkeit“.

- Mehrere Apps können über Verträge zusammenarbeiten.

Sie können als Programmierer auf die Dienste bereits vorhandener Apps zurückgreifen oder aber die Dienste Ihrer App selber anderen Apps zur Verfügung stellen. Die Zusammenarbeit der Apps und der Austausch der Daten werden dabei über Verträge geregelt.

Die Verträge werden auch Contracts^{a)} genannt.



- a) *Contract* ist die englische Übersetzung für „Vertrag“.

- Jede App hat einen eigenen Lebenszyklus.

Eine Windows-Forms- oder WPF-Anwendung wird in der Regel gestartet und irgendwann wieder beendet. Vielleicht wird sie zwischendurch auch vom Anwender minimiert oder in den Hintergrund gestellt. Das spielt für Sie als Programmierer aber nur eine untergeordnete Rolle. Bei Universal Windows Platform Apps dagegen müssen Sie sich sehr viel intensiver mit dem Lebenszyklus der Anwendung beschäftigen. Denn sobald Ihre App in den Hintergrund gestellt wird, wird sie durch das Betriebssystem in eine Art Schlafzustand versetzt.

Der Schlafzustand wird auch Suspend-Modus genannt.



Dadurch wird sichergestellt, dass die jeweils aktive Anwendung die maximale Leistung des Systems nutzen kann und nicht unnötig Strom verbraucht wird. Im Extremfall kann das Betriebssystem eine App sogar komplett beenden – zum Beispiel, wenn der freie Arbeitsspeicher zu knapp wird.

Was sich auf den ersten Blick nicht besonders problematisch liest, hat aber eine besondere Tücke. Zwar werden die Daten einer Anwendung im Suspend-Modus automatisch gesichert, aber nicht beim Beenden der Anwendung. Darum müssen Sie sich als Programmierer kümmern. Außerdem sind Sie dafür zuständig, dass die Daten nach dem zwangsweisen Beenden beim nächsten Start auch wiederhergestellt werden.

Hinweis:

Da Universal Windows Platform Apps automatisch vom Betriebssystem beendet werden können, gibt es in der Regel in der App selber keine eigene Funktion zum Beenden. Sie müssen entweder eine entsprechende Geste verwenden oder aber die Tastenkombination **Alt + F4**.

1.4 Die Entwicklung von Universal Windows Platform Apps

Genau wie bei WPF-Anwendungen wird auch bei der Entwicklung von Universal Windows Platform Apps konsequent zwischen der Präsentation und der Logik getrennt. Für die Beschreibung der Oberfläche können Sie neben XAML zum Beispiel auch HTML verwenden. Die Logik – also das eigentliche Programmverhalten – lässt sich mit Programmiersprachen wie C++, JavaScript, Visual Basic oder eben C# abbilden.

Auch wenn einiges bei der Entwicklung von Universal Windows Platform Apps zu Beginn möglicherweise komplettes Neuland für Sie ist, wird Ihnen vieles bekannt vorkommen. So ähnelt der Designer für Universal Windows Platform Apps dem Designer für WPF-Anwendungen, den Sie ja bereits kennengelernt haben.

Hinweis:

Bei der Bezeichnung von Windows Apps herrscht leider ein wenig Durcheinander. Es gibt auch Windows Runtime Apps, Windows 8 Apps, universelle Windows 8 Apps und Windows Store Apps. Gemeint sind damit aber in der Regel Apps, die nicht speziell für Windows 10 erstellt wurden, sondern für Windows 8, Windows Phone oder Windows RT.

Zusammenfassung

Universal Windows Platform Apps sind Anwendungen, die auf unterschiedlichen Geräten eingesetzt werden. Sie setzen Windows 10 als Betriebssystem voraus.

Universal Windows Platform Apps weisen einige Besonderheiten in der Bedienung und auch bei der Entwicklung auf.

Für die Entwicklung können Sie unterschiedliche Programmiersprachen wie C++ und C# verwenden. Für die Beschreibung der Oberfläche kann HTML oder XAML verwendet werden.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Nennen Sie mindestens drei Besonderheiten in der Gestaltung und Bedienung von Universal Windows Platform Apps.

1.2 Was ist eine Sandbox?

1.3 Was regeln Capabilities?

2 Eine erste App

In diesem Kapitel erfahren Sie, wie Sie den Entwicklermodus aktivieren. Danach legen Sie Ihre erste einfache App an und testen sie.

2.1 Das Projekt erstellen

Für das Erstellen von Apps legen Sie wie gewohnt ein neues Projekt an.

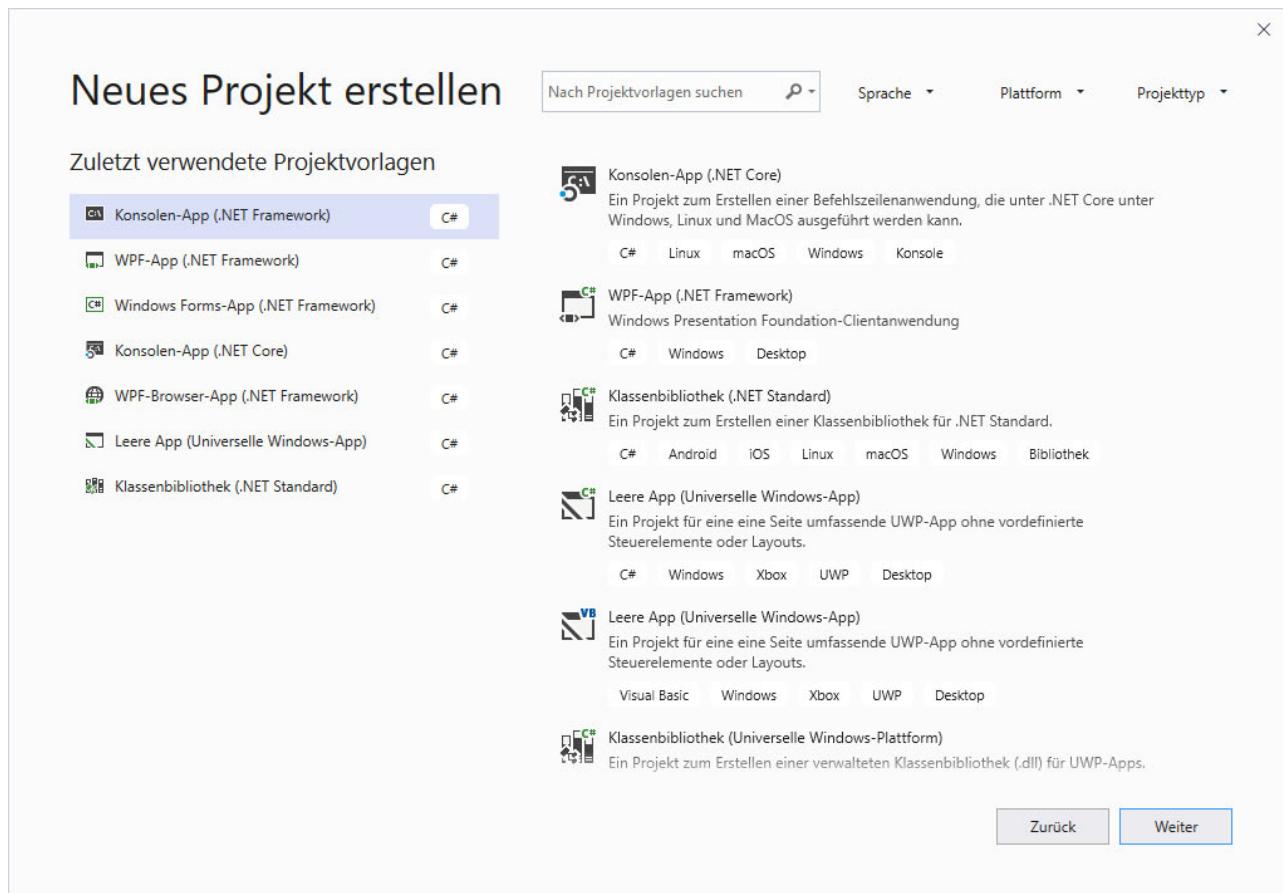


Abb. 2.1: Das Fenster Neues Projekt erstellen

Als Vorlage wählen Sie **Leere App (Universal Windows-App)** für die Programmiersprache C#.

Markieren Sie den entsprechenden Eintrag. Geben Sie anschließend im nächsten Schritt wie gewohnt einen Namen für das Projekt ein und wählen Sie einen Ordner aus. Wir verwenden in unserem Beispiel den Namen **HalloWeltApp**.

Klicken Sie dann auf **Erstellen**, um das Projekt anzulegen.

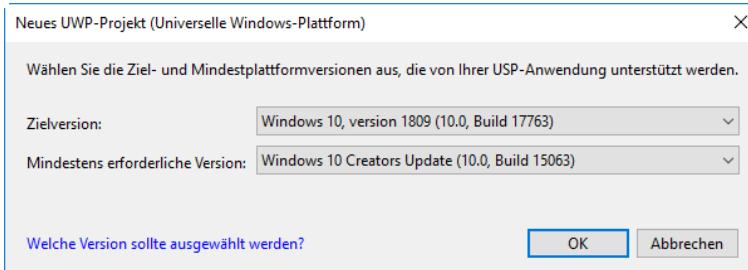


Abb. 2.2: Das Fenster Neues UWP-Projekt

Im folgenden Fenster legen Sie die Zielversion und die mindestens erforderliche Windows-Version für die Ausführung fest. In unserem Beispiel übernehmen wir die Vorschlagswerte. Klicken Sie also auf **OK**.

Im nächsten Schritt müssen Sie den Entwicklermodus für Windows 10 aktivieren. Dazu werden automatisch die Windows-Einstellungen geöffnet.

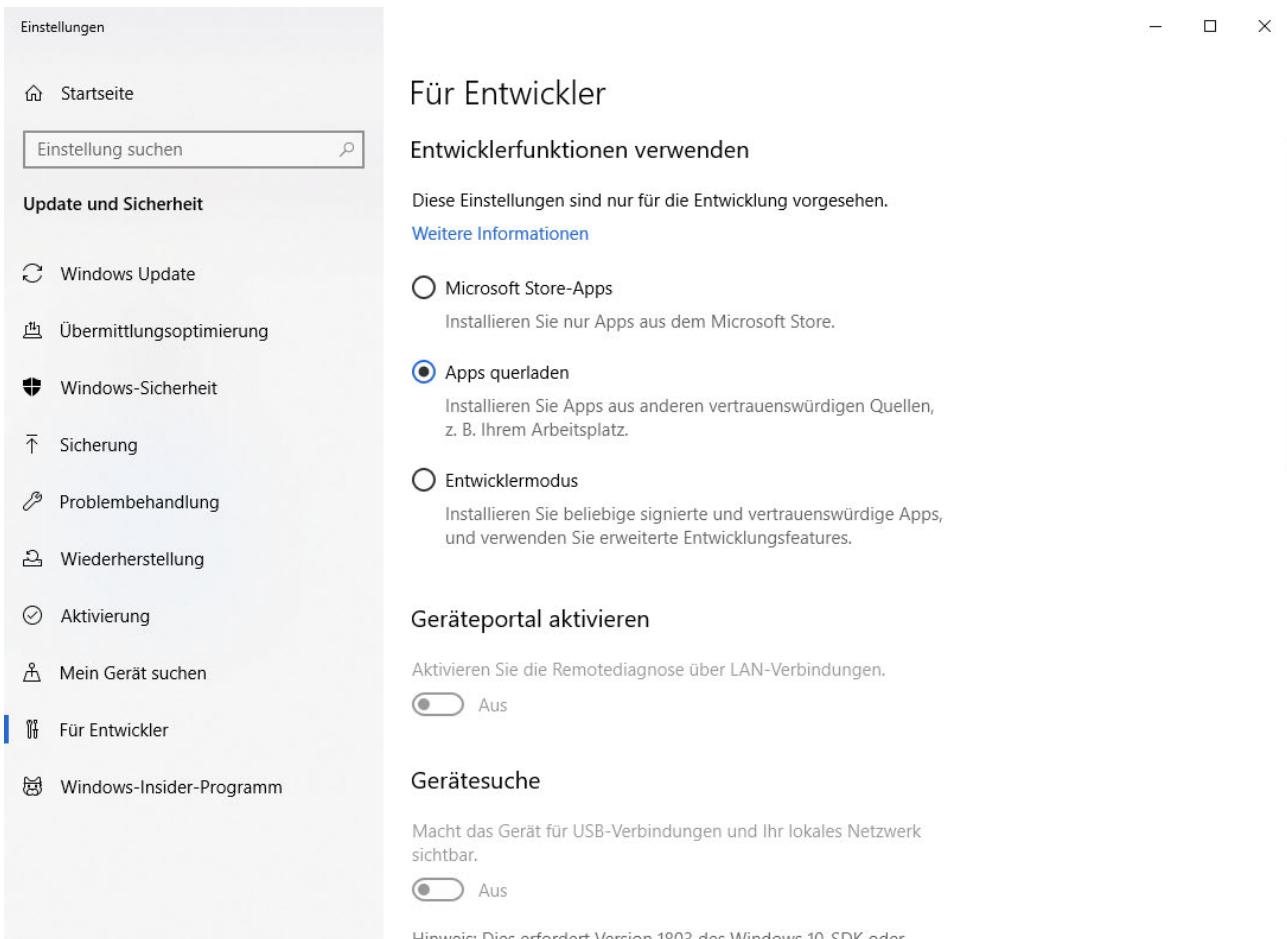


Abb. 2.3: Die Entwicklerfunktionen bei den Einstellungen

Aktivieren Sie hier den Eintrag **Entwicklermodus** rechts im Fenster. Bestätigen Sie anschließend die Sicherheitsabfrage mit einem Mausklick auf **Ja**. Schließen Sie danach das Fenster **Einstellungen** wieder.

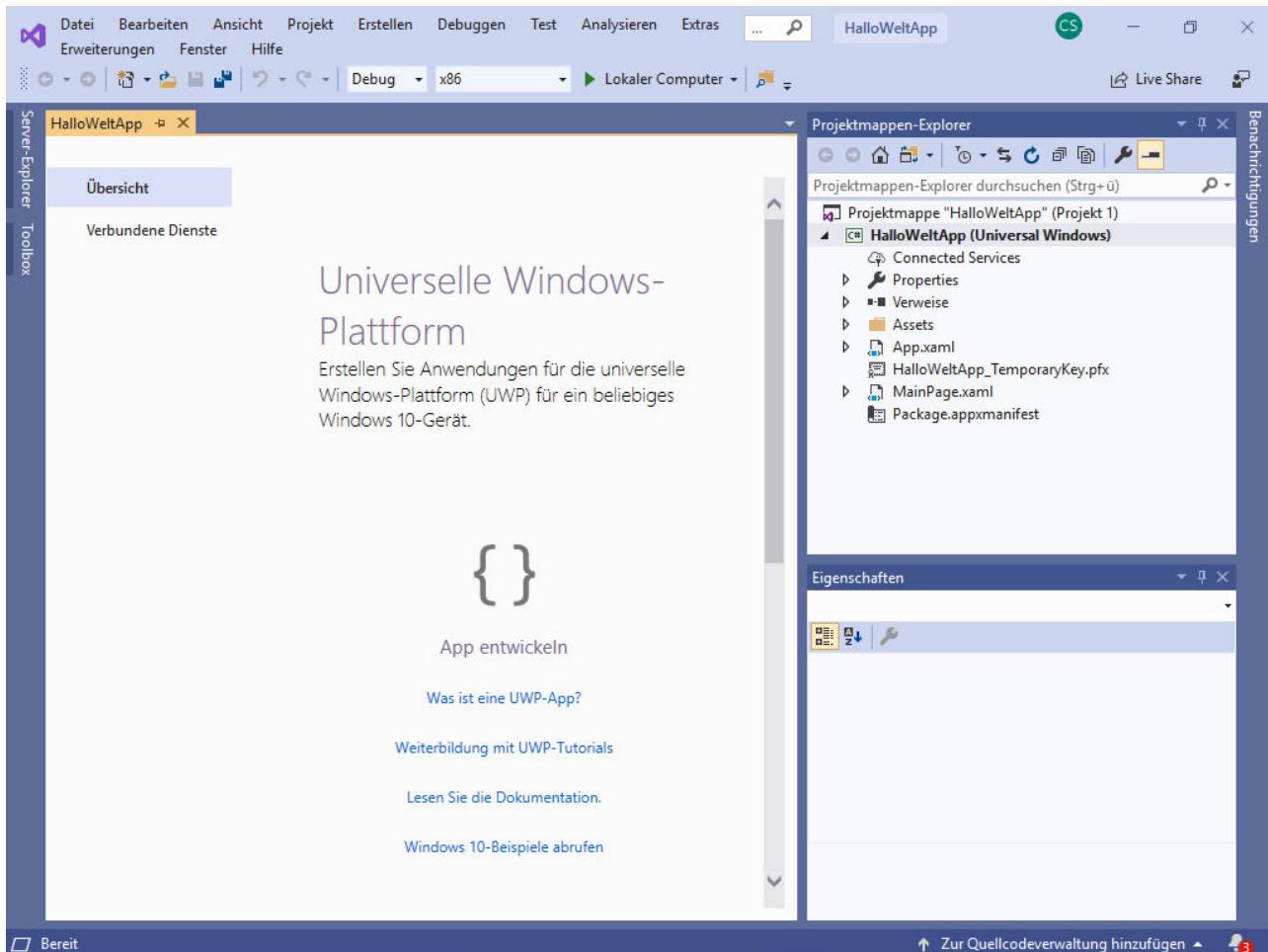


Abb. 2.4: Das neue Projekt

Das Projekt wird angelegt und erscheint in der Entwicklungsumgebung. Dabei wird zunächst eine Übersichtsseite angezeigt.

Im Projektmappen-Explorer rechts im Fenster von Visual Studio sehen Sie auch noch weitere Dateien und Ordner. Die Dateien in der Gruppe **MainPage.xaml** enthalten dabei die Startseite der App. Im Ordner **Assets** werden unter anderem Logos für die App abgelegt. In der Gruppe **App.xaml** befinden sich die Dateien der eigentlichen App.

Wechseln Sie bitte in die Startseite der App. Doppelklicken Sie dazu auf die Datei **MainPage.xaml** im Projektmappen-Explorer. Über die Toolbox oder auch direkt im XAML-Code können Sie jetzt wie gewohnt Steuerelemente einfügen. Probieren Sie das bitte aus.

Setzen Sie einen Textblock für die Anzeige von Text in die App. Das entsprechende Steuerelement finden Sie in der Toolbox unter dem Namen **TextBlock**.

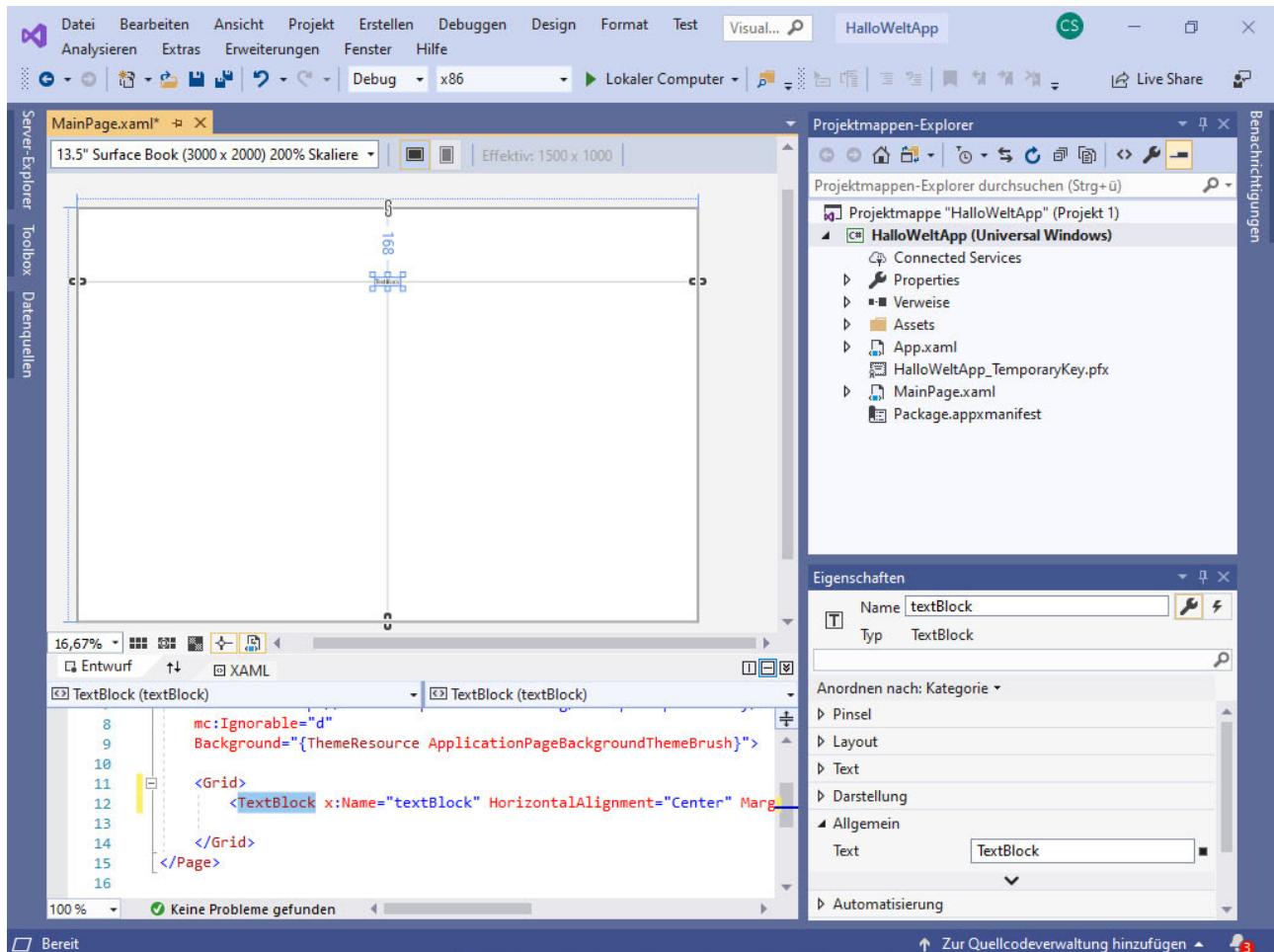


Abb. 2.5: Ein eingefügter Textblock

Ändern Sie den Text für den Textblock dann über die Eigenschaften – zum Beispiel in „Hallo Welt“. Damit der Text besser zu sehen ist, können Sie auch die Schriftgröße anpassen – zum Beispiel auf 24. Die entsprechenden Einstellungen finden Sie bei den Eigenschaften im Bereich **Text**.

Positionieren Sie danach das Steuerelement neu und lassen Sie die App ausführen. Dazu können Sie wie gewohnt die Funktion **Debuggen/Starten ohne Debugging** oder die Tastenkombination **Strg + F5** nutzen.

Nach einiger Zeit erscheint zuerst der Splash Screen – der Startbildschirm – der Anwendung. Da wir kein eigenes Logo angelegt haben, wird hier lediglich ein Viereck mit einem X angezeigt.

Kurze Zeit später wird aber auch die Oberfläche unserer ersten App angezeigt.

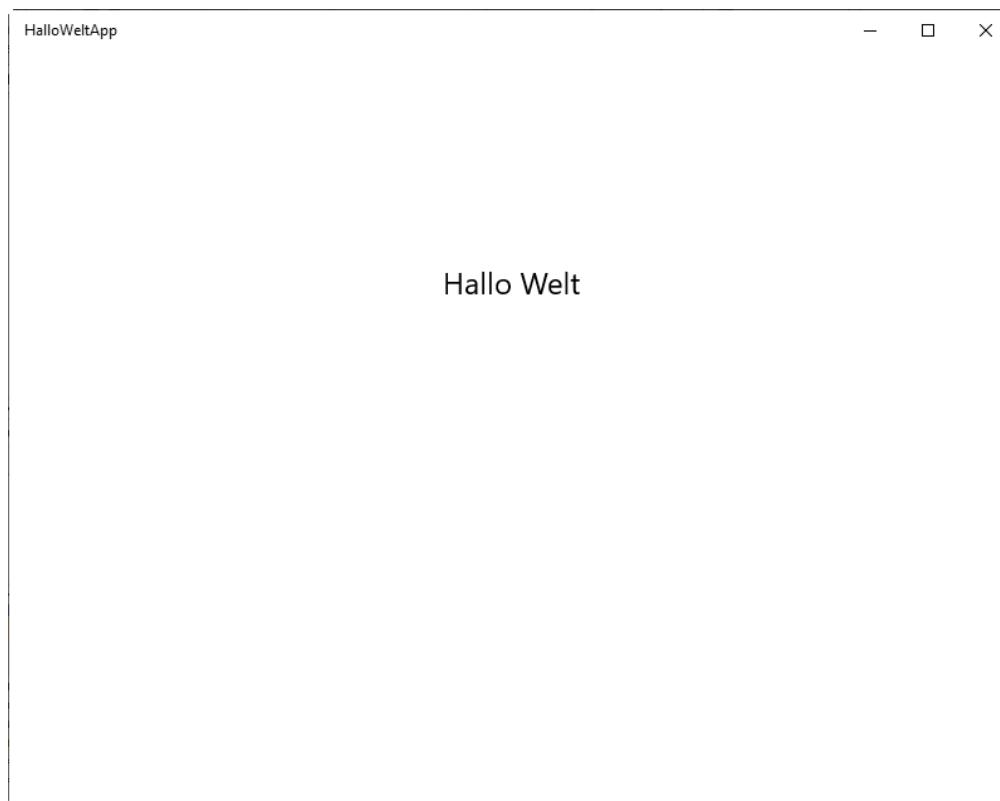


Abb. 2.6: Die erste App

Diese App wird aber nicht nur ausgeführt, sondern auch im Startmenü unter **Alle Apps** angezeigt. Das können Sie ganz einfach überprüfen, indem Sie das Startmenü öffnen und zu den Apps wechseln. Unter dem Buchstaben H sollte dann unsere App erscheinen.

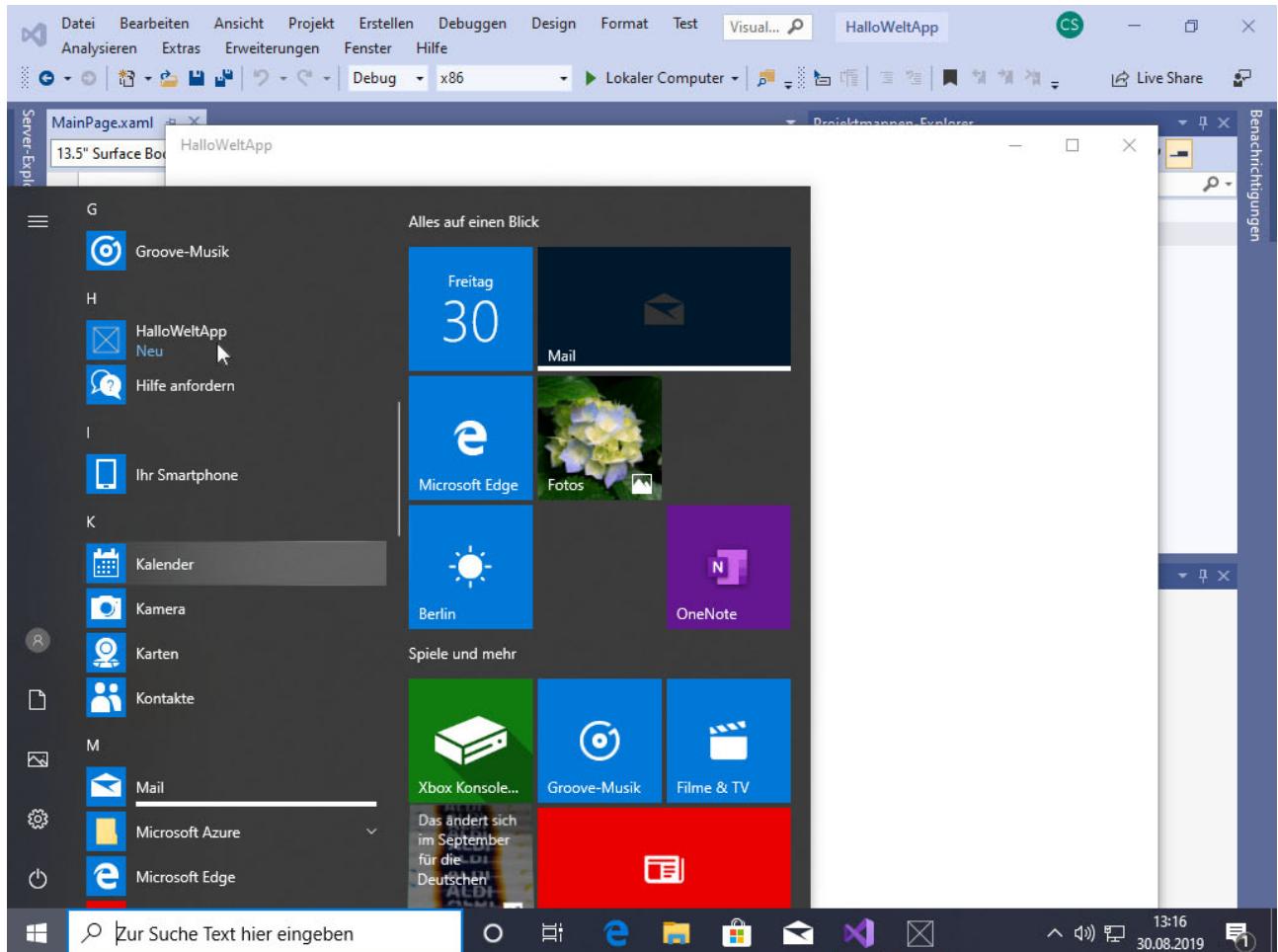


Abb. 2.7: Der Eintrag der App im Startmenü (links am Mauszeiger)

Hinweis:

Um den Eintrag der App wieder zu löschen, klicken Sie ihn mit der rechten Maustaste an. Wählen Sie dann im Kontext-Menü die Funktion **Deinstallieren** und bestätigen Sie die Deinstallation.

Über die Felder ganz oben im Designer können Sie verschiedene Geräte, Auflösungen und Ausrichtungen für die App einstellen.

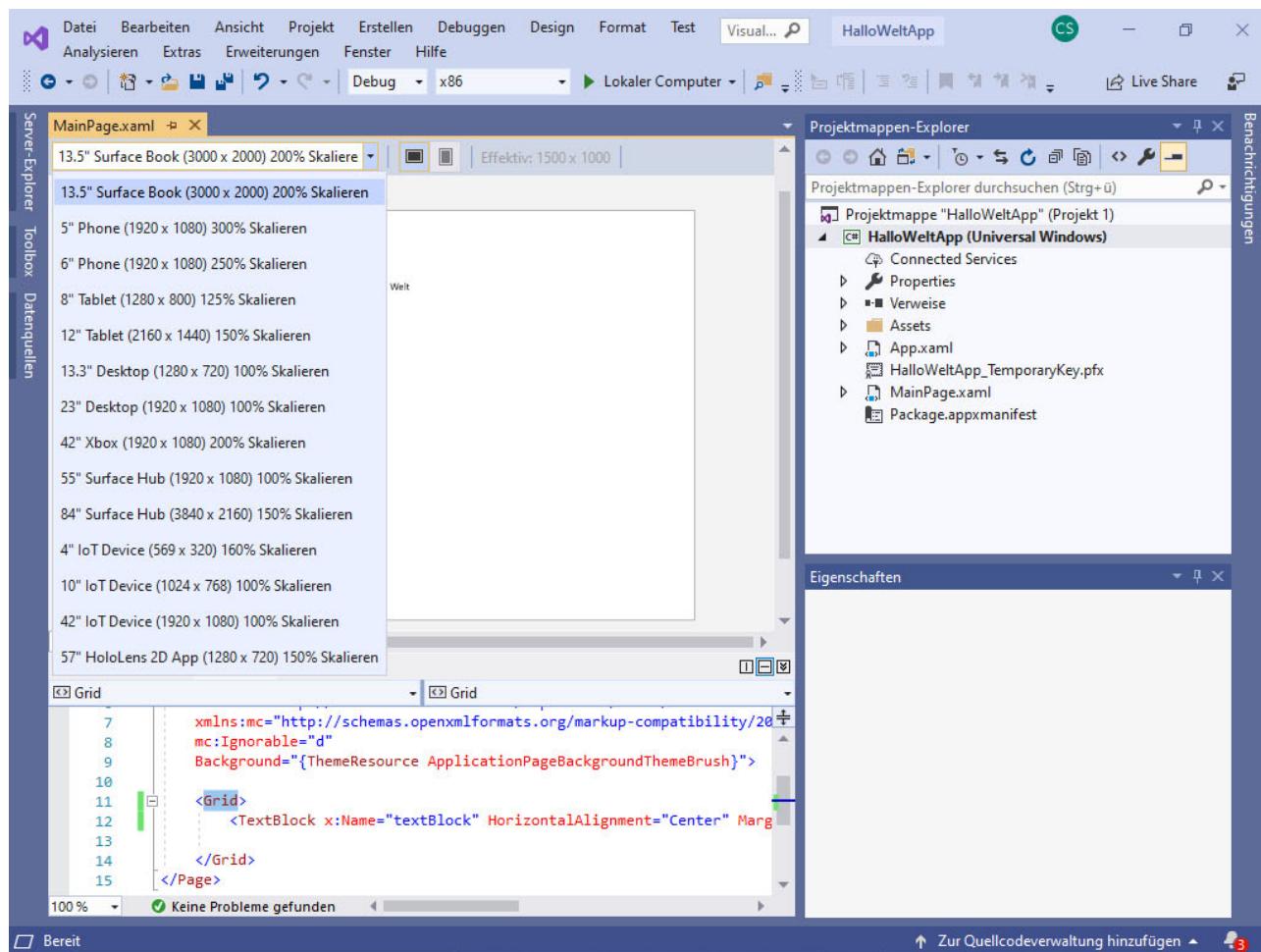


Abb. 2.8: Die Auswahl der Geräte und Auflösungen (links in der Abbildung)

Die Änderungen werden sofort im Designer sichtbar. So können Sie das Verhalten bei veränderten Auflösungen oder Seitenverhältnissen sehr gut testen und gegebenenfalls auch direkt Korrekturen vornehmen.

2.2 Test auf unterschiedlichen Geräten

Ausgeführt wird die App in der Standardeinstellung aber immer auf Ihrem Rechner. Damit wird sie auch so angezeigt, wie sie auf einem Desktop aussehen würde. Sie können Ihre App auch in einem Simulator testen. Damit lässt sich zum Beispiel das Verhalten auf einem Tablet simulieren.



Ein Simulator versucht, wesentliche Aspekte eines Systems nachzubilden.

Die entsprechenden Einstellungen nehmen Sie über das Kombinationsfeld in der Mitte der Symbolleiste vor. Ändern Sie hier einmal den Eintrag von **Lokaler Computer** in **Simulator**. Dadurch erfolgt ein Test auf einem simulierten Tablett mit Windows 10.

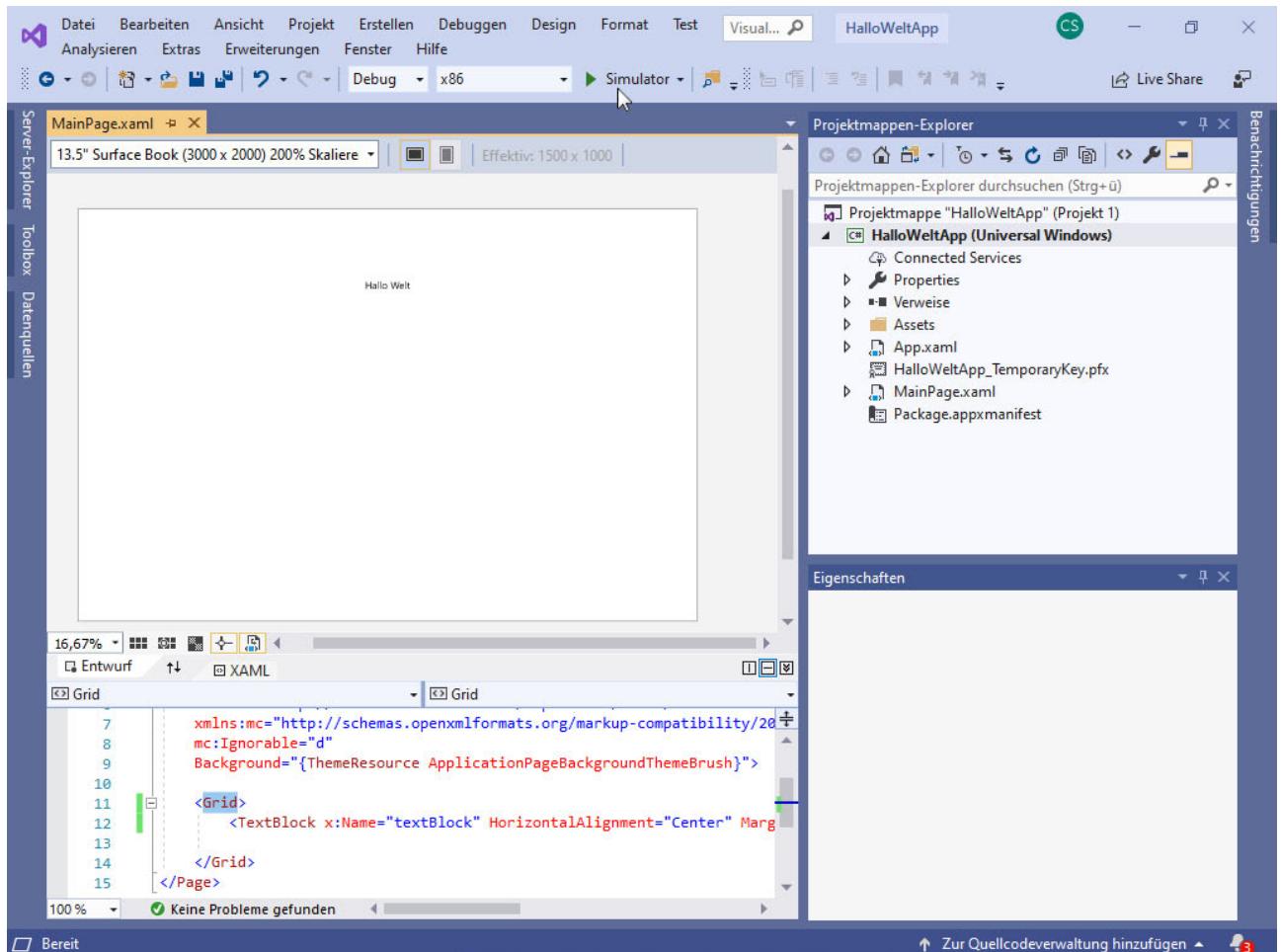


Abb. 2.9: Die Einstellungen für den Test im Simulator (in der Mitte der Symbolleiste am Mauszeiger; die Einstellungen sind bereits geändert)

Wenn Sie die App dann starten, wird sie im Simulator ausgeführt. Bitte beachten Sie aber, dass der Start des Simulators unter Umständen einige Zeit dauern kann.

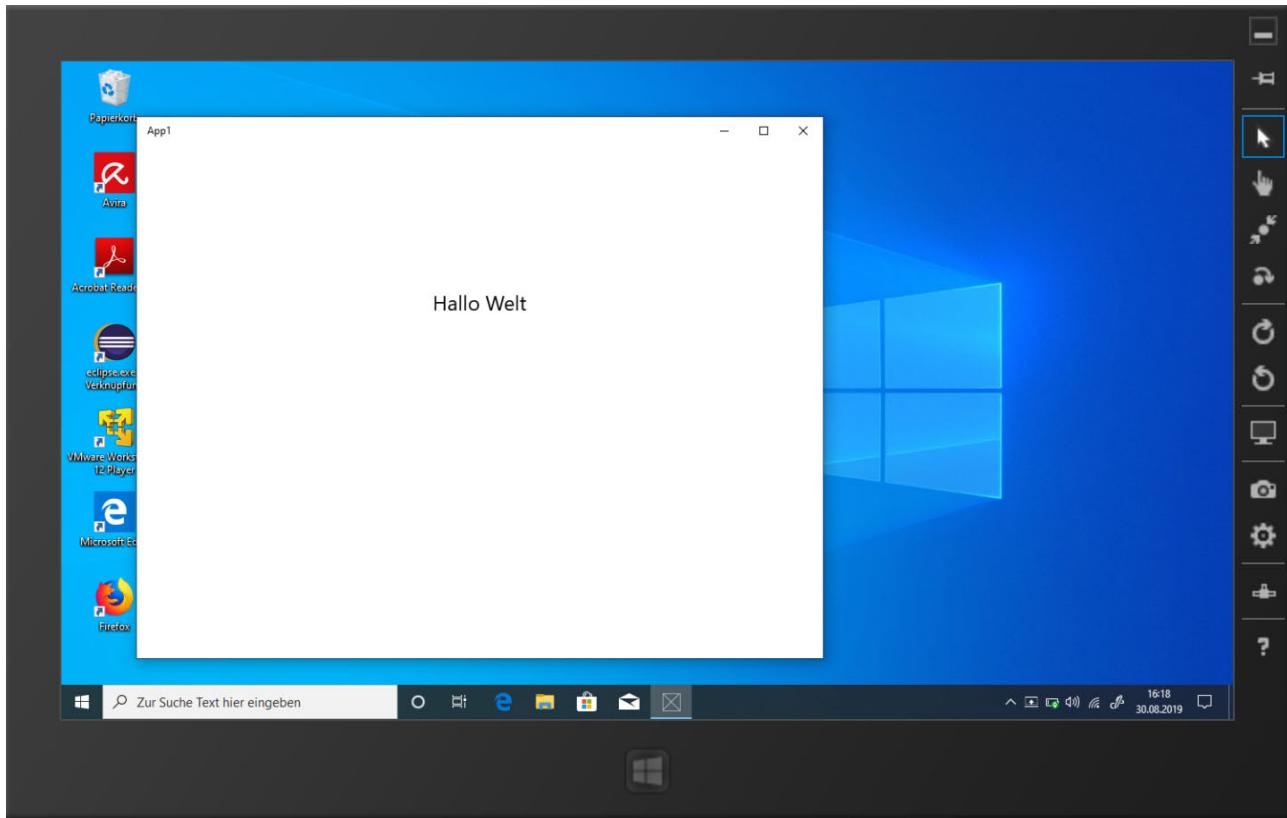


Abb. 2.10: Die App im Simulator

Über die Symbolleiste am rechten Rand des Simulators können Sie dann zum Beispiel die Bedienung mit dem Finger mit der Maus simulieren. Dazu verwenden Sie die Symbole im oberen Bereich der Symbolleiste. Sie können aber auch die Darstellung im Simulator drehen beziehungsweise ein Drehen des Tablets simulieren. Die entsprechenden Symbole finden Sie im mittleren Bereich der Symbolleiste.

Probieren Sie die verschiedenen Funktionen am besten selbst in Ruhe aus. Hilfe und weitere Informationen erhalten Sie, wenn Sie auf das Fragezeichen-Symbol ganz unten in der Symbolleiste klicken. Die Hilfeseite wird allerdings aus dem Internet geladen.

Um den Simulator zu beenden, drücken Sie die Tastenkombination **Strg + Alt + F4**. Sie können den Simulator aber auch mit der Funktion **Fenster schließen** im Kontext-Menü des Symbols in der Taskleiste beenden.

Hinweis:

Der Simulator stellt Ihren Rechner als Tablet nach. So finden Sie zum Beispiel auf dem Desktop des Simulators genau die Anwendungen, die Sie auch über den Desktop Ihres Rechners starten können. Allerdings werden die Anwendungen auch als Desktop-Anwendungen ausgeführt. Sie sehen also nicht anders aus als auf Ihrem Desktop.

Zusammenfassung

Für eine App verwenden Sie die Vorlage **Leere App (Universal Windows-App)** für die Programmiersprache C#.

Die Logik einer App wird wie bei WPF-Anwendungen in Code-behind-Dateien programmiert.

Steuerelemente fügen Sie über den Designer und die Toolbox ein. Sie können sie aber auch direkt im XAML-Code eintragen.

Nach dem Starten einer App erscheint zunächst der Startbildschirm.

Beim Ausführen wird auch automatisch ein Eintrag im Startmenü angelegt.

Sie können verschiedene Geräte, Auflösungen und Ausrichtungen für eine App vorgeben. Außerdem ist ein Test im Simulator möglich.

Aufgaben zur Selbstüberprüfung

- 2.1 Was enthält der Ordner **Assets** in einem Projekt für eine Universal Windows Platform App?

- 2.2 Beschreiben Sie, wie Sie eine App im Simulator testen.

- 2.3 Mit welcher Tastenkombination schließen Sie den Simulator für das Ausführen einer App wieder?

3 Ein Taschenrechner als App

In diesem Kapitel werden wir eine etwas komplexere App entwickeln – und zwar einen Taschenrechner für die vier Grundrechenarten. Dabei erfahren Sie unter anderem, wie Sie auf das Beenden der App durch das System reagieren.

3.1 Die Grundfunktionen

Die Grundfunktionen unserer Taschenrechner-App orientieren sich an dem einfachen Taschenrechner, den wir bereits einmal als Windows Forms-Anwendung erstellt haben. Wir bieten dem Anwender zwei Eingabefelder für die Zahlen an und lassen das Ergebnis als Text ausgeben. Die Eingabe der Zahlen und auch die Auswahl der Rechenoperation sollen aber vor allem über Schaltflächen erfolgen. Damit unterstützen wir gezielt die Bedienung über einen Touchscreen.

Damit die Bedienung möglichst einfach ist, rufen wir die Rechenoperation direkt nach dem Antippen beziehungsweise Anklicken einer Schaltfläche für die Rechenoperation auf.

Die fertige App soll ungefähr so aussehen:

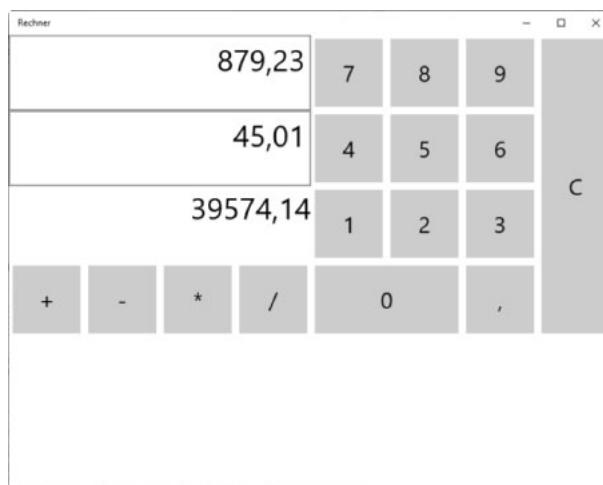


Abb. 3.1: Die Taschenrechner-App

Hinweis:

Wir halten den Rechner hier ganz bewusst einfach und konzentrieren uns vor allem auf den grundsätzlichen Ablauf beim Erstellen.

Legen Sie jetzt bitte ein neues Projekt für eine Universal Windows Platform App mit Visual C# an. Als Vorlage benutzen Sie wieder eine leere App. Als Namen können Sie zum Beispiel **Rechner** verwenden.

3.2 Die Oberfläche des Taschenrechners

Beginnen wir mit der Entwicklung der Oberfläche. Hier verwenden wir ein Grid als Container. In diesem Container ordnen wir zwei Eingabefelder, ein Ausgabefeld und die Schaltflächen an.

Hinweis:

Die Container von Apps basieren wie bei der WPF auf XAML. Wir stellen sie Ihnen daher hier nicht noch einmal vor.

Die Tabelle soll fünf Zeilen und sieben Spalten enthalten. Die ersten vier Zeilen sollen dabei 100 Pixel hoch sein und die letzte Zeile den verbleibenden Rest des Bildschirms einnehmen. Die Breite der Spalten machen wir von der Auflösung des Bildschirms abhängig. Alle Spalten sollen dabei gleich breit sein. Das lässt sich am einfachsten durch eine proportionale Breite erledigen.

Die XAML-Anweisungen für die Tabelle sehen so aus:

```
<Grid.RowDefinitions>
    <RowDefinition Height="100"/>
    <RowDefinition Height="100"/>
    <RowDefinition Height="100"/>
    <RowDefinition Height="100"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
```

Code 3.1: Das Grid

Geben Sie die Anweisungen aus dem vorigen Code jetzt im XAML-Editor unterhalb der Anweisung

```
<Grid>
    ...
</Grid>
```

ein.

Achten Sie dabei bitte darauf, dass Sie das Ende-Tag für das Grid nicht überschreiben und dass Sie die Tags korrekt schließen. Nach der Eingabe der Anweisungen sollte die Seite so aussehen:

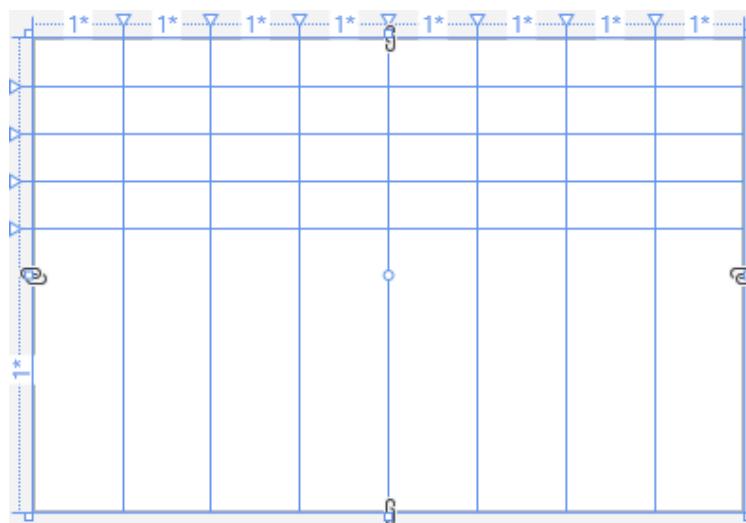


Abb. 3.2: Das Grid in der Seite

Fügen Sie dann ein Eingabefeld vom Typ **TextBox** ein. Setzen Sie die Eigenschaften **Row** und **Column** im Bereich **Layout** des Eigenschaftenfensters jeweils auf 0, damit das Eingabefeld in der Zelle ganz links oben angezeigt wird. Sie können das Eingabefeld aber auch mit der Maus direkt in dieser Zelle ablegen beziehungsweise es nach dem Einfügen mit der Maus in diese Zelle ziehen.



Denken Sie bitte daran:

Die Nummerierung der Spalten und Zeilen beginnt bei 0 und nicht bei 1.

Lassen Sie das Feld über vier Spalten anzeigen. Ändern Sie dazu die Eigenschaft **ColumnSpan** in 4. Setzen Sie dann noch die Ränder bei der Eigenschaft **Margin** jeweils auf 0 und ändern Sie die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** auf **Stretch** beziehungsweise löschen Sie diese Eigenschaften für die Textbox im XAML-Code.

Damit wir das Eingabefeld gleich eindeutig identifizieren können, vergeben Sie bitte über das Feld **Name**: im Eigenschaftenfenster noch einen Bezeichner – zum Beispiel `eingabe1`. Löschen Sie abschließend den Text in dem Feld. Setzen Sie dazu die Eigenschaft **Text** auf einen leeren Wert oder löschen Sie die Eigenschaft im XAML-Code.

Die Oberfläche sollte nun so aussehen:

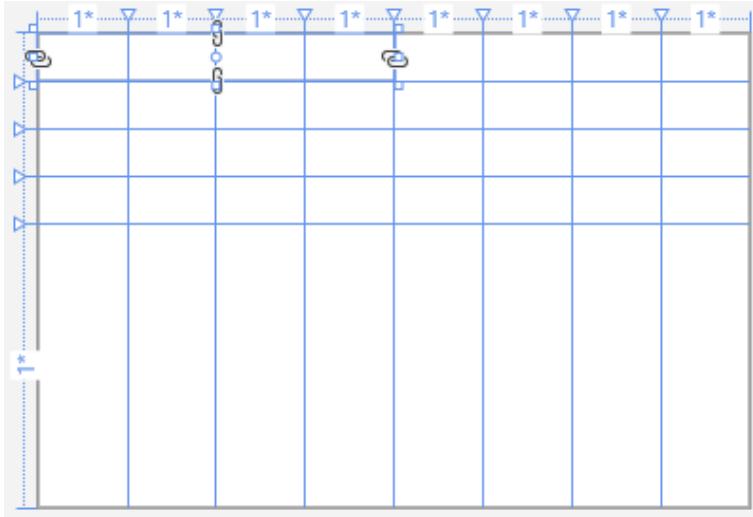


Abb. 3.3: Das Eingabefeld

Legen Sie dann das zweite Eingabefeld an. Das geht am schnellsten, wenn Sie den XAML-Code des ersten Feldes kopieren und anpassen. Er muss so aussehen:

```
<TextBox x:Name="eingabe2" Grid.ColumnSpan="4" Margin="0,0,0,0"
TextWrapping="Wrap" Grid.Row="1"/>
```

Code 3.2: Das zweite Eingabefeld

Durch die Angabe `Grid.Row="1"` sorgen wir dafür, dass das Eingabefeld in der zweiten Zeile der Tabelle angezeigt wird.

Fügen Sie anschließend einen Textblock für die Ausgabe in die dritte Zeile ein. Das entsprechende Steuerelement finden Sie in der Toolbox als **TextBlock**.

Ändern Sie die Eigenschaften des Textblocks so, dass er ebenfalls vier Spalten umfasst und die Spalten sowie die Zeile komplett ausfüllt. Vergeben Sie auch an den Textblock einen sprechenden Namen wie `ausgabe` und löschen Sie seinen Inhalt. Die XAML-Anweisung für den Textblock sollte danach ungefähr so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<TextBlock x:Name="ausgabe" Grid.ColumnSpan="4"
Margin="0,0,0,0" Grid.Row="2" TextWrapping="Wrap"/>
```

Code 3.3: Der Textblock

Hinweis:

Die Grenzen zwischen den beiden Eingabefeldern und dem Ausgabefeld sind nicht zu erkennen, da wir den Rand auf 0 gesetzt haben. Wenn Sie das stört, können Sie den oberen und den unteren Rand ja wieder erhöhen – zum Beispiel auf 5. Dann sind die Steuerelemente auch sichtbar getrennt.

Jetzt fehlen noch die Schaltflächen für die Eingabe der Zahlen. Sie sollen in den Spalten 5, 6 und 7 und den Zeilen 1, 2 und 3 angezeigt werden. Wie bei einem echten Taschenrechner sollen in der ersten Zeile die Ziffern 7, 8 und 9 stehen. Darunter folgen 4, 5 und 6 sowie 1, 2 und 3. Die Ziffer 0 stellen wir nach links in die vierte Zeile. Die Schaltfläche soll dabei zwei Spalten umfassen.

Da die Schaltflächen bis auf die Beschriftung und die Position weitgehend identisch sind, sollten Sie sich eine Schaltfläche als Vorlage anlegen und mehrfach kopieren. Fügen Sie dazu im ersten Schritt ein Steuerelement **Button** aus der Toolbox in die vierte Spalte in der ersten Reihe ein. Setzen Sie die Eigenschaft **Content** der Schaltfläche auf 7 und sorgen Sie dafür, dass die Schaltfläche die gesamte Zelle ausfüllt. Ändern Sie dazu die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** jeweils in **Stretch** oder löschen Sie sie im XAML-Code. Die Ränder setzen Sie bitte jeweils auf 5. Passen Sie dann noch die Schriftgröße an – zum Beispiel auf 30.

Da wir gleich die Reaktion auf das Anklicken beziehungsweise Antippen für alle Schaltflächen mit den Ziffern in einem Ereignis behandeln, legen Sie bitte auch dieses Ereignis jetzt schon an. Das erfolgt bei einer App genauso wie bei einer WPF-Anwendung über die Ereignisse im Eigenschaftenfenster. Als Namen für das Ereignis können Sie zum Beispiel `Button_Click_Ziffer` verwenden. Der XAML-Code für die Schaltfläche sollte dann so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<Button Content="7" Grid.Column="4"
HorizontalAlignment="Stretch" Margin="5"
VerticalAlignment="Stretch" FontSize="30"
Click="Button_Click_Ziffer"/>
```

Code 3.4: Die erste Schaltfläche

Erstellen Sie von diesem Code insgesamt neun Kopien. Passen Sie den Code so an, dass die Schaltflächen korrekt in den jeweiligen Zellen dargestellt werden. Dazu müssen Sie bei den Schaltflächen in der zweiten, dritten und vierten Zeile auch noch die Position für die Zeile ergänzen. Die Schaltfläche für die 0 lassen Sie bitte über zwei Spalten anzeigen.

Die Oberfläche sollte danach ungefähr so aussehen:

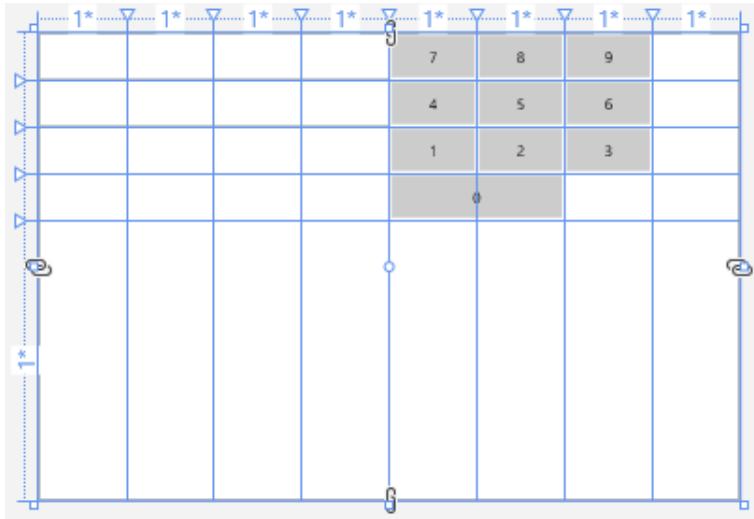


Abb. 3.4: Die Schaltflächen für die Ziffern

Ergänzen Sie dann noch vier Schaltflächen in der vierten Zeile für die Auswahl der Rechenart. Der XAML-Code für diese Schaltflächen sieht so aus:

```
<!-- bitte jeweils in einer Zeile eingeben -->
<Button Content="+" Grid.Row ="3" HorizontalAlignment="Stretch"
Margin="5" VerticalAlignment="Stretch" FontSize="30" />
<Button Content="-" Grid.Column="1" Grid.Row ="3"
HorizontalAlignment="Stretch" Margin="5"
VerticalAlignment="Stretch" FontSize="30" />
<Button Content="*" Grid.Column="2" Grid.Row ="3"
HorizontalAlignment="Stretch" Margin="5"
VerticalAlignment="Stretch" FontSize="30" />
<Button Content="/" Grid.Column="3" Grid.Row ="3"
HorizontalAlignment="Stretch" Margin="5"
VerticalAlignment="Stretch" FontSize="30" />
```

Code 3.5: Die Schaltflächen für die Rechenarten

Jetzt fehlen nur noch eine Schaltfläche für ein Komma und eine Schaltfläche, um alle Eingaben zurückzusetzen. Die Schaltfläche für das Komma können Sie rechts neben die Schaltfläche mit der Ziffer 0 setzen. Sie soll ebenfalls beim Anklicken die Methode aufrufen, die auch die anderen Schaltflächen für die Ziffern ausführen. Die Schaltfläche zum Löschen können Sie in die letzte Spalte setzen und über die ersten vier Zeilen laufen lassen.

Die beiden XAML-Anweisungen sehen entsprechend so aus:

```
<!-- bitte in einer Zeile eingeben -->
<Button Content="," Grid.Column="6" Grid.Row ="3"
HorizontalAlignment="Stretch" Margin="5"
VerticalAlignment="Stretch" FontSize="30"
Click="Button_Click_Ziffer"/>
<Button Content="C" Grid.Column="7" Grid.RowSpan="4"
HorizontalAlignment="Stretch" Margin="5"
VerticalAlignment="Stretch" FontSize="30"/>
```

Code 3.6: Die Schaltfläche für die Ziffer 0 und die Schaltfläche zum Löschen

Damit ist die Oberfläche fertiggestellt. Speichern Sie die Änderungen und sehen Sie sich die Oberfläche in verschiedenen Ansichten und Auflösungen an. Probieren Sie dazu verschiedene Einträge im Kombinationsfeld links oben im Editor aus. Die Darstellung sollte bei allen Geräten brauchbar sein. Lediglich bei sehr niedrigen Auflösungen werden im Querformat die unteren Schaltflächen abgeschnitten. Das liegt an der festen Höhe der ersten vier Zeilen.

Im nächsten Schritt können wir uns um die Logik der Anwendung kümmern.

3.3 Die Logik

Grundsätzlich muss unsere App „wissen“, in welches Eingabefeld gerade Daten eingegeben werden sollen. Dazu verwenden wir ein Feld mit dem Namen `aktiveEingabe`, das je nach aktivem Eingabefeld entweder den Wert 1 oder den Wert 2 erhält. Gesetzt wird dieser Wert im Ereignis **GotFocus** für die beiden Eingabefelder.

Vereinbaren Sie bitte zunächst in der Klasse für die Anwendung ein Feld mit dem Namen `aktiveEingabe`. Als Typ können Sie `int` verwenden.



Zur Erinnerung:

Die Klasse für die Anwendung befindet sich in der Datei mit dem Namen `MainPage.xaml.cs`.

Markieren Sie anschließend das erste Eingabefeld in Ihrer Oberfläche. Wechseln Sie dann im Eigenschaftenfenster zu den Ereignissen und doppelklicken Sie in das Feld hinter dem Eintrag **GotFocus**. Setzen Sie in der Methode für das Ereignis den Wert des Feldes `aktiveEingabe` auf 1. Im Ereignis **GotFocus** für das zweite Eingabefeld setzen Sie den Wert des Feldes `aktiveEingabe` entsprechend auf 2.

Kommen wir nun zur Methode für die Schaltflächen für die Ziffern. Diese Methode haben Sie ja bereits beim Anlegen der Schaltflächen zugewiesen. Sie soll eine weitere Methode aufrufen, die den Text von der Schaltfläche an die bereits vorhandene Eingabe in einem Feld anhängt. Die Methode für das Anklicken könnte zum Beispiel so aussehen:

```
private void Button_Click_Ziffer(object sender,  
RoutedEventArgs e)  
{  
    Anzeigen((sender as Button).Content.ToString());  
}
```

Code 3.7: Die Methode für das Anklicken der Ziffern

Aufgerufen wird hier eine Methode `Anzeigen()`, die den Text auf der Schaltfläche als Zeichenkette erhält. Damit wir die Methode für alle Schaltflächen mit Ziffern und auch für die Schaltfläche mit dem Komma verwenden können, greifen wir nicht direkt auf die Taste zu, die das Ereignis auslöst, sondern beschaffen uns den Auslöser über den Ausdruck `(sender as Button)`. Er formt das Objekt, das als `sender` übergeben wird, in den Typ `Button` um und ermöglicht uns so den Zugriff auf die Eigenschaft `Content`. Da diese Eigenschaft ebenfalls den sehr allgemeinen Typ `object` liefert, müssen wir das Ergebnis noch mit der Methode `ToString()` in eine Zeichenkette umbauen.

Die Methode `Anzeigen()` selbst ist dann wieder sehr einfach. Sie prüft, welches Eingabefeld aktiv ist, und hängt die übergebene Zeichenkette an den Inhalt im Eingabefeld an. Die komplette Methode sieht so aus:

```
private void Anzeigen(string eingabe)  
{  
    if (aktiveEingabe == 1)  
        eingabe1.Text = eingabe1.Text + eingabe;  
    else  
        eingabe2.Text = eingabe2.Text + eingabe;  
}
```

Code 3.8: Die Methode `Anzeigen()`

Übernehmen Sie jetzt die beiden Methoden. Die Methode für das Click-Ereignis können Sie sehr einfach aufrufen, indem Sie in der XAML-Ansicht im Kontextmenü für das Click-Ereignis die Funktion **Gehe zu Definition** aufrufen. Alternativ können Sie aber auch auf das entsprechende Ereignis im Eigenschaftenfenster doppelklicken.

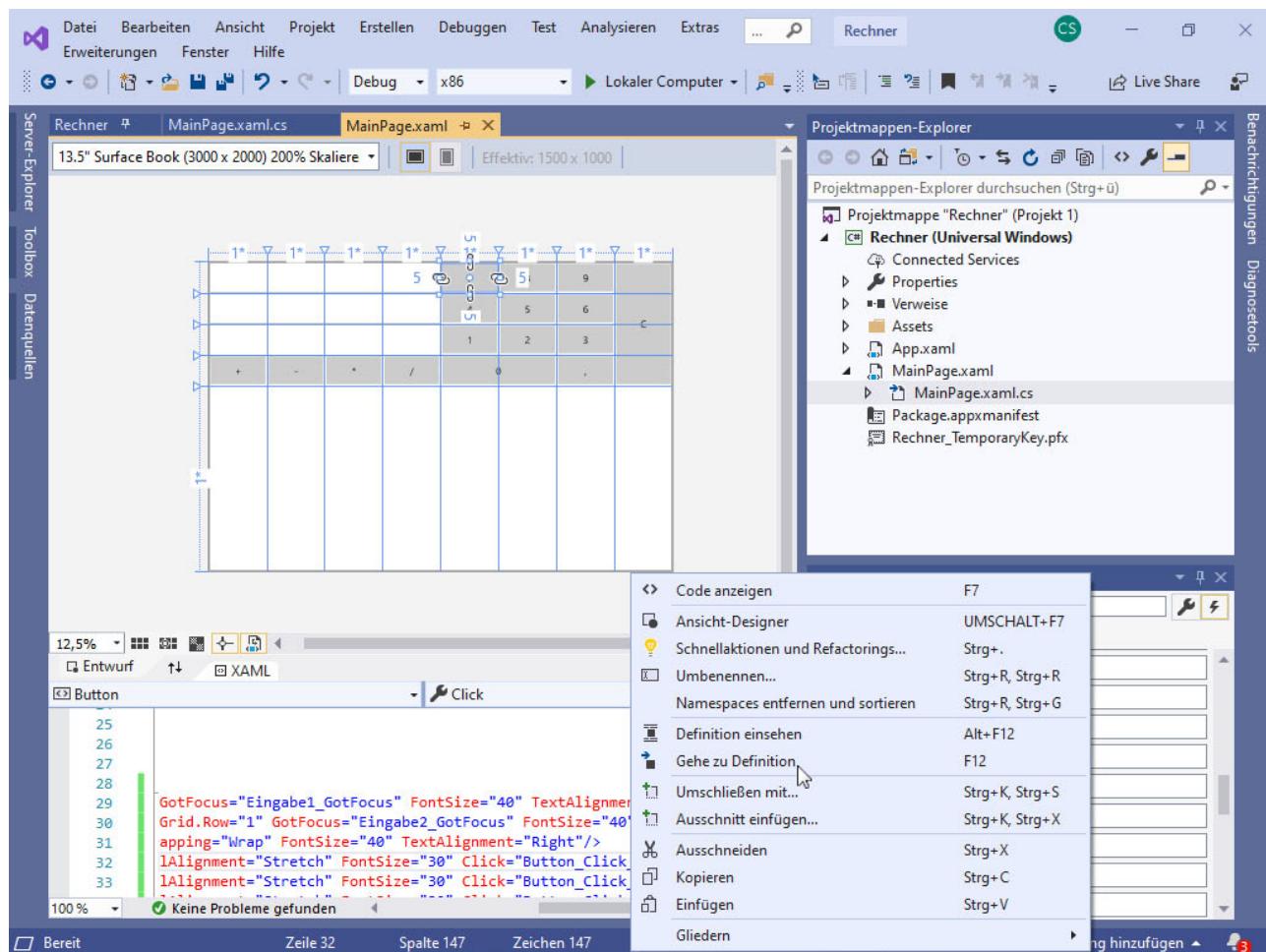


Abb. 3.5: Die Funktion Gehe zu Definition im Kontextmenü

Speichern Sie danach die Änderungen und testen Sie die App. Sie sollten jetzt Zahlen und Kommas in die beiden Eingabefelder eingeben können.

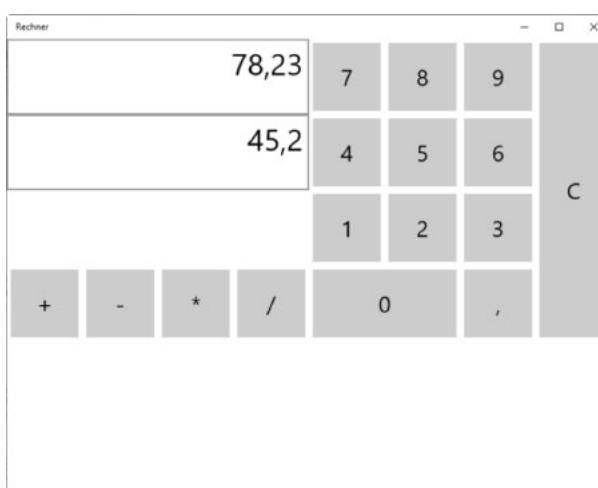


Abb. 3.6: Ein erster Test der App

Jetzt fehlen noch die Rechnungen. Hier rufen wir beim Anklicken einer Schaltfläche für die Rechenoperationen eine eigene Methode auf, die die Berechnungen durchführt und das Ergebnis im Textblock ausgibt. Welche Rechnung ausgeführt wird, machen wir von der angeklickten Schaltfläche abhängig.

Die Methode für das Click-Ereignis der vier Schaltflächen für die Rechenoperationen könnte zum Beispiel so aussehen:

```
private void Button_Click_Rechnung(object sender,  
RoutedEventArgs e)  
{  
    Rechnen((sender as Button).Content.ToString());  
}
```

Code 3.9: Die Methode für die Schaltflächen mit den Rechenoperationen

Die Methode für das Berechnen wertet dann den übergegebenen Rechenoperator aus, führt die Berechnung durch und gibt das Ergebnis aus. Sie sieht so aus:

```
private void Rechnen(string eingabe)  
{  
    float zahl1, zahl2, ergebnis = 0;  
    zahl1 = Convert.ToSingle(eingabe1.Text);  
    zahl2 = Convert.ToSingle(eingabe2.Text);  
    switch (eingabe)  
    {  
        case "+":  
            ergebnis = zahl1 + zahl2;  
            break;  
        case "-":  
            ergebnis = zahl1 - zahl2;  
            break;  
        case "*":  
            ergebnis = zahl1 * zahl2;  
            break;  
        case "/":  
            ergebnis = zahl1 / zahl2;  
            break;  
    }  
    ausgabe.Text = ergebnis.ToString();  
}
```

Code 3.10: Die Methoden für die Rechenoperationen

Übernehmen Sie jetzt auch diese beiden Methoden. Denken Sie bitte daran, dass Sie die Methode für das Click-Ereignis allen vier Schaltflächen für die Rechenoperationen zuweisen müssen. Speichern Sie danach die Änderungen und testen Sie das Programm noch einmal. Nun sollten auch die Berechnungen durchgeführt werden.

Provozieren Sie bei Ihren Tests auch einmal einen Fehler. Geben Sie zum Beispiel einen Wert in ein Eingabefeld ein, der nicht konvertiert werden kann, oder lassen Sie eines der beiden Eingabefelder leer. Wenn Sie dann eine Berechnung ausführen lassen, werden Sie eine Überraschung erleben. Es erscheint nämlich nicht wie bei einer Windows Forms- oder eine WPF-Anwendung eine Meldung, sondern die App wird einfach beendet.

Durch diesen Mechanismus soll verhindert werden, dass eine nicht mehr funktionierende App Ressourcen des Systems in Anspruch nimmt und so möglicherweise andere Apps blockiert.

Damit der Anwender weiß, dass etwas schiefgegangen ist, stattet wir unsere App noch mit einer Fehlermeldung aus. Dazu benutzen wir die Klasse `Windows.UI.Popups.MessageDialog`. Sie erzeugt einen einfachen Dialog. Da dieser Dialog asynchron arbeitet, kann die eigentliche Anwendung weiterarbeiten.



Bei Universal Windows Platform Apps sollten alle Prozesse, die etwas zeitintensiver sind, asynchron arbeiten. Dadurch können Sie verhindern, dass die laufende App behindert wird. Die asynchrone Verarbeitung spielt auch beim Laden und Speichern von Daten eine besondere Rolle. Damit werden wir uns gleich noch intensiver beschäftigen.

Für die asynchrone Verarbeitung sind zwei Schlüsselwörter wichtig: `async` und `await`. Das Schlüsselwort `async` verwenden Sie für Methoden, die asynchrone Verarbeitungen durchführen. Mit dem Schlüsselwort `await` können Sie auf das Ergebnis der asynchronen Verarbeitung warten und es auswerten. Eine Methode `FehlerMeldung()`, die den Dialog für die neue Oberfläche aufruft, könnte dann so aussehen:

```
private async void FehlerMeldung()
{
    //bitte in einer Zeile eingeben
    MessageDialog meinDialog = new MessageDialog("Es ist ein
    Problem aufgetreten");
    await meinDialog.ShowAsync();
}
```

Code 3.11: Die Methode FehlerMeldung()

Über das Schlüsselwort `async` wird dem Compiler mitgeteilt, dass die Methode eine asynchrone Verarbeitung durchführt. Diese asynchrone Verarbeitung erfolgt in unserem Beispiel über die Methode `ShowAsync()` der Klasse `MessageDialog`. Das Schlüsselwort `await` beim Aufruf der asynchronen Methode sorgt dafür, dass auf das Ergebnis der Verarbeitung gewartet wird.

Hinweis:

Sie können das Schlüsselwort `await` auch weglassen. Dann wird allerdings nicht auf das Ergebnis gewartet, sondern die Verarbeitung der Methode fortgesetzt. Da das Ergebnis dabei in der Regel verloren geht, zeigt Ihnen der Compiler eine entsprechende Warnung an.

Der Aufruf der Methode `FehlerMeldung()` kann in einer `try-catch`-Konstruktion beim Berechnen erfolgen. Da es dabei keine Besonderheiten gibt, drucken wir den Quelltext hier nicht ab. Sie finden ihn im Projekt im Download-Bereich Ihrer Online-Lernplattform.

Hinweis:

Die Klasse `MessageDialog` liegt im Namensraum `Windows.UI.Popups`. Sie müssen diesen Namensraum erst bekannt machen beziehungsweise vor den Bezeichner der Klasse stellen.

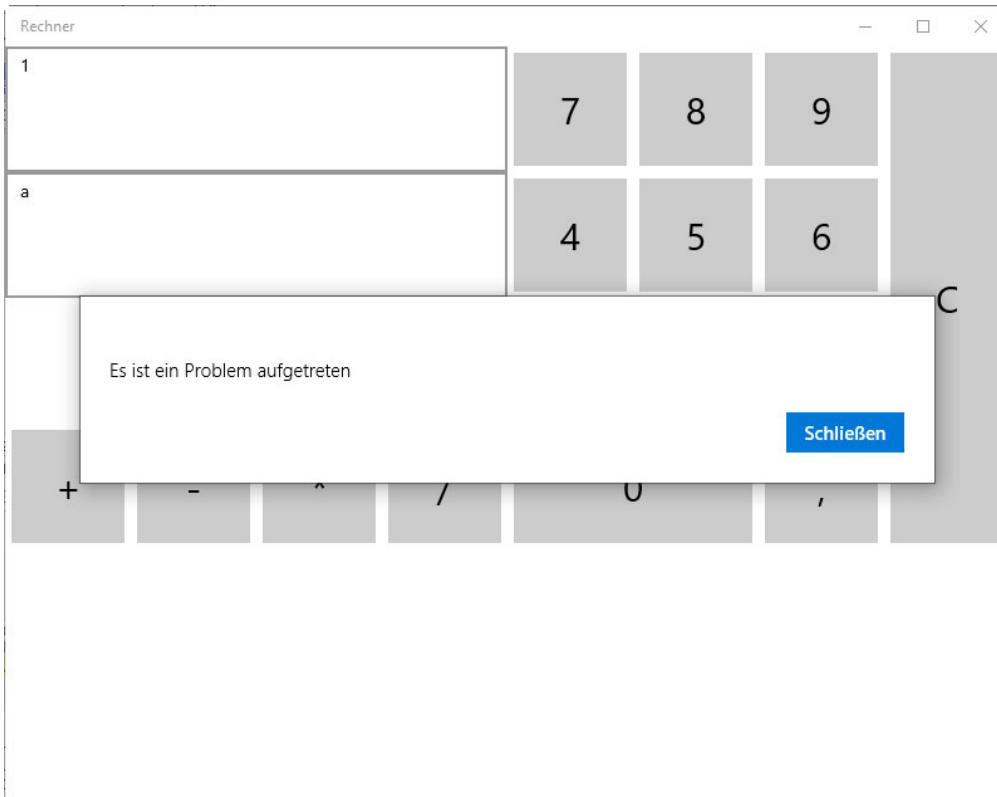


Abb. 3.7: Der Dialog

Für die Grundfunktionalität fehlt jetzt nur noch die Methode beim Anklicken der Schaltfläche C. Hier löschen wir den Text in den Eingabefeldern und auch im Textblock. Dabei gibt es ebenfalls keine Besonderheiten. Sie finden den Quelltext daher auch nur im Projekt im Download-Bereich Ihrer Online-Lernplattform.

3.4 Reaktion auf das Beenden durch das System

Nachdem die Grundfunktionalität der Anwendung steht, werden wir uns um eine typische Anforderung von Universal Windows Platform Apps kümmern – die Reaktion auf das Beenden der App durch das System. Dieses zwangsweise Beenden erfolgt zum Beispiel dann, wenn die App nicht mehr reagiert oder wenn sie sich im Hintergrund befindet und die Systemressourcen knapp werden. Damit der Anwender beim nächsten Start seine App so wieder vorfindet, wie er sie verlassen hat, müssen Sie die Daten der App beim Beenden speichern und beim Start wieder laden.

Sie können sich nicht darauf verlassen, dass eine App immer durch den Anwender beendet wird. Auch das System selber kann eine App beenden – und zwar, ohne dass der Anwender davon etwas mitbekommt. Sie sollten daher in jedem Fall dafür sorgen, dass die Daten gesichert und wiederhergestellt werden.



Das Speichern und Wiederherstellen der App-Daten ist nicht allzu kompliziert, da ein entsprechendes Codegerüst bereits in der Datei `App.xaml.cs` vorhanden ist. Hier finden Sie zum einen in der Methode `OnLaunched()`, die nach dem Starten der App ausgeführt wird, eine Abfrage, ob die App beim letzten Mal vom System beendet wurde. Dazu wird die Eigenschaft `args.PreviousExecutionState` mit dem Wert `ApplicationExecutionState.Terminated` verglichen. Er markiert, dass die App durch das System beendet wurde.

Die komplette Abfrage finden Sie im folgenden Code. Damit Sie sie in der Datei `App.xaml.cs` schneller wiederfinden, haben wir auch noch einige Anweisungen vor und hinter der Abfrage mit abgedruckt. Die eigentliche Abfrage ist im Code fett markiert.

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    ...
    Frame rootFrame = Window.Current.Content as Frame;
    ...
    if (rootFrame == null)
    {
        rootFrame = new Frame();
        rootFrame.NavigationFailed += OnNavigationFailed;
        if (e.PreviousExecutionState ==
            ApplicationExecutionState.Terminated)
        {
            //Zustand von zuvor angehaltener Anwendung laden
        }

        // Den Frame im aktuellen Fenster platzieren
        Window.Current.Content = rootFrame;
    }
    ...
}
```

Code 3.12: Die Methode OnLaunched()

Für das Laden und Speichern der Daten benutzen wir die Klasse `Windows.Storage.ApplicationData`. Sie stellt uns eine Eigenschaft `Current.LocalSettings` vom Typ `ApplicationDataContainer` zur Verfügung. Diese Eigenschaft steht für den Datenspeicher der App. Über ihn können wir sehr einfach Daten ablegen und auch wieder beschaffen. Dazu geben Sie bei der Eigenschaft `Values` in eckigen Klammern einen Schlüssel als Zeichenkette an und weisen dann einen Wert zu. Die Anweisungen

```
ApplicationDataContainer lokaleDaten =
    ApplicationData.Current.LocalSettings;
lokaleDaten.Values["eingabel"] = eingabel.Text;
```

erzeugen zuerst eine Instanz `lokaleDaten` der Klasse `ApplicationDataContainer` und weisen ihr über den Ausdruck `ApplicationData.Current.LocalSettings` den Datenspeicher der App zu. Danach erzeugen wir einen Eintrag mit dem Schlüssel `eingabel` und legen in diesem Schlüssel den Text aus dem ersten Eingabefeld ab.

Der lesende Zugriff auf die Daten erfolgt dann ganz ähnlich. Sie erzeugen zunächst wieder eine Instanz der Klasse `ApplicationDataContainer` und weisen ihr den Datenspeicher der App zu. Danach können Sie über die Eigenschaft `Values` auf die Schlüssel und die Werte zugreifen.

Die beiden Methoden im folgenden Code setzen das Speichern und Laden für unsere App praktisch um.

```
public void Speichern ()
{
    //den Container erzeugen
    //bitte in einer Zeile eingeben
    ApplicationDataContainer lokaleDaten =
    ApplicationData.Current.LocalSettings;
    //die Daten schreiben
    lokaleDaten.Values["eingabe1"] = eingabe1.Text;
    lokaleDaten.Values["eingabe2"] = eingabe2.Text;
    lokaleDaten.Values["ausgabe"] = ausgabe.Text;
}

public void Laden()
{
    //den Container erzeugen
    //bitte in einer Zeile eingeben
    ApplicationDataContainer lokaleDaten =
    ApplicationData.Current.LocalSettings;
    //gibt es Daten?
    //wenn ja, dann laden
    if (lokaleDaten.Values["eingabe1"] != null)
        eingabe1.Text = lokaleDaten.Values["eingabe1"].ToString();
    if (lokaleDaten.Values["eingabe2"] != null)
        eingabe2.Text = lokaleDaten.Values["eingabe2"].ToString();
    if (lokaleDaten.Values["ausgabe"] != null)
        ausgabe.Text = lokaleDaten.Values["ausgabe"].ToString();
}
```

Code 3.13: Das Laden und Speichern der Daten

Hinweis:

Denken Sie bitte an das Einbinden des Namensraums `Windows.Storage` über eine entsprechende `using`-Anweisung.

In der Methode zum Laden überprüfen wir zur Sicherheit zusätzlich noch, ob unter dem jeweiligen Schlüssel überhaupt Daten abgelegt sind.

Übernehmen Sie die beiden Methoden jetzt bitte in die Klasse für die Hauptseite – also in die Datei `MainPage.xaml.cs`.

Der Zugriff auf die beiden Methoden erfolgt dann in der Klasse der App – also in der Datei `App.xaml.cs`. Dazu müssen wir uns im ersten Schritt die Instanz mit der Hauptseite der App beschaffen. Vereinbaren Sie dazu ein Feld vom Typ `MainPage`. Wir ver-

wenden in unserem Beispiel den Bezeichner `meineSeite`. Weisen Sie diesem Feld in der Methode `OnLaunched()` den Inhalt des Basisframes der Anwendung zu. Das erfolgt über die Anweisung

```
meineSeite = rootFrame.Content as MainPage;
```



Ein Frame^{a)} dient als Container für die Navigation zwischen Seiten einer App.

a) *Frame* bedeutet übersetzt „Rahmen“.

Ergänzen Sie diese Anweisung bitte hinter der Anweisung

```
rootFrame.Navigate(typeof(MainPage), e.Arguments);
```

in der Abfrage `if (rootFrame.Content == null)`

Vorher liefert der Ausdruck `rootFrame.Content` immer `null` und wir können nicht auf die Hauptseite der App zugreifen.

Ergänzen Sie hinter der Anweisung noch eine Abfrage, ob die App beim letzten Beenden vom System beendet wurde. Die entsprechende Anweisung können Sie aus der Abfrage `if (rootFrame == null)` kopieren. Wenn das System die App beendet hat, rufen Sie die Methode zum Laden der Daten auf. Die vollständige Methode `OnLaunched()` sollte nach diesen Erweiterungen so aussehen wie im folgenden Code. Neue Anweisungen haben wir zur besseren Übersicht wieder fett markiert.

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;

    // App-Initialisierung nicht wiederholen, wenn das Fenster
    // bereits Inhalte enthält.
    // Nur sicherstellen, dass das Fenster aktiv ist.
    if (rootFrame == null)
    {
        // Frame erstellen, der als Navigationskontext fungiert, und
        // zum Parameter der ersten Seite navigieren
        rootFrame = new Frame();
        rootFrame.NavigationFailed += OnNavigationFailed;

        if (e.PreviousExecutionState ==
            ApplicationExecutionState.Terminated)
        {
            // Zustand von zuvor angehaltener Anwendung laden
        }

        // Den Frame im aktuellen Fenster platzieren
        Window.Current.Content = rootFrame;
    }
}
```

```
if (rootFrame.Content == null)
{
    // Wenn der Navigationsstapel nicht wiederhergestellt
    // wird, zur ersten Seite navigieren und die neue Seite
    // konfigurieren, indem die erforderlichen Informationen
    // als Navigationsparameter übergeben werden
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
    //Frame als Hauptseite beschaffen
    meineSeite = rootFrame.Content as MainPage;
    //bitte in einer Zeile eingeben
    if (e.PreviousExecutionState ==
        ApplicationExecutionState.Terminated)
    {
        meineSeite.Laden();
    }
}
// Sicherstellen, dass das aktuelle Fenster aktiv ist
Window.Current.Activate();
}
```

Code 3.14: Die Methode OnLaunched()

Das Speichern ist dann recht einfach. Hier ergänzen wir einfach in der Methode `OnSuspending()` die Anweisung zum Speichern zwischen den beiden bereits vorhandenen Anweisungen. Die Methode sieht dann so aus:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //die Daten speichern
    //das erfolgt über eine Methode in der Hauptseite
    meineSeite.Speichern();
    deferral.Complete();
}
```

Code 3.15: Die Methode OnSuspending()**Hinweis:**

Die Methode `OnSuspending()` wird bei jedem Wechsel in den Suspend-Modus ausgeführt – also nicht nur beim zwangseisigen Herunterfahren der App durch das System. Allerdings gibt es kein eigenes Ereignis, das beim Herunterfahren durch das System ausgelöst wird. Daher nehmen wir in Kauf, dass die Daten häufiger gespeichert werden, als es eigentlich erforderlich ist.

Übernehmen Sie jetzt bitte die Anweisungen aus den vorigen Codes. Führen Sie dann einen Test durch. Das geht am einfachsten, wenn Sie die App im Simulator über den Debugger starten und dann im Kombinationsfeld **Ereignisse des Lebenszyklus** die Funktion **Anhalten und herunterfahren** wählen.

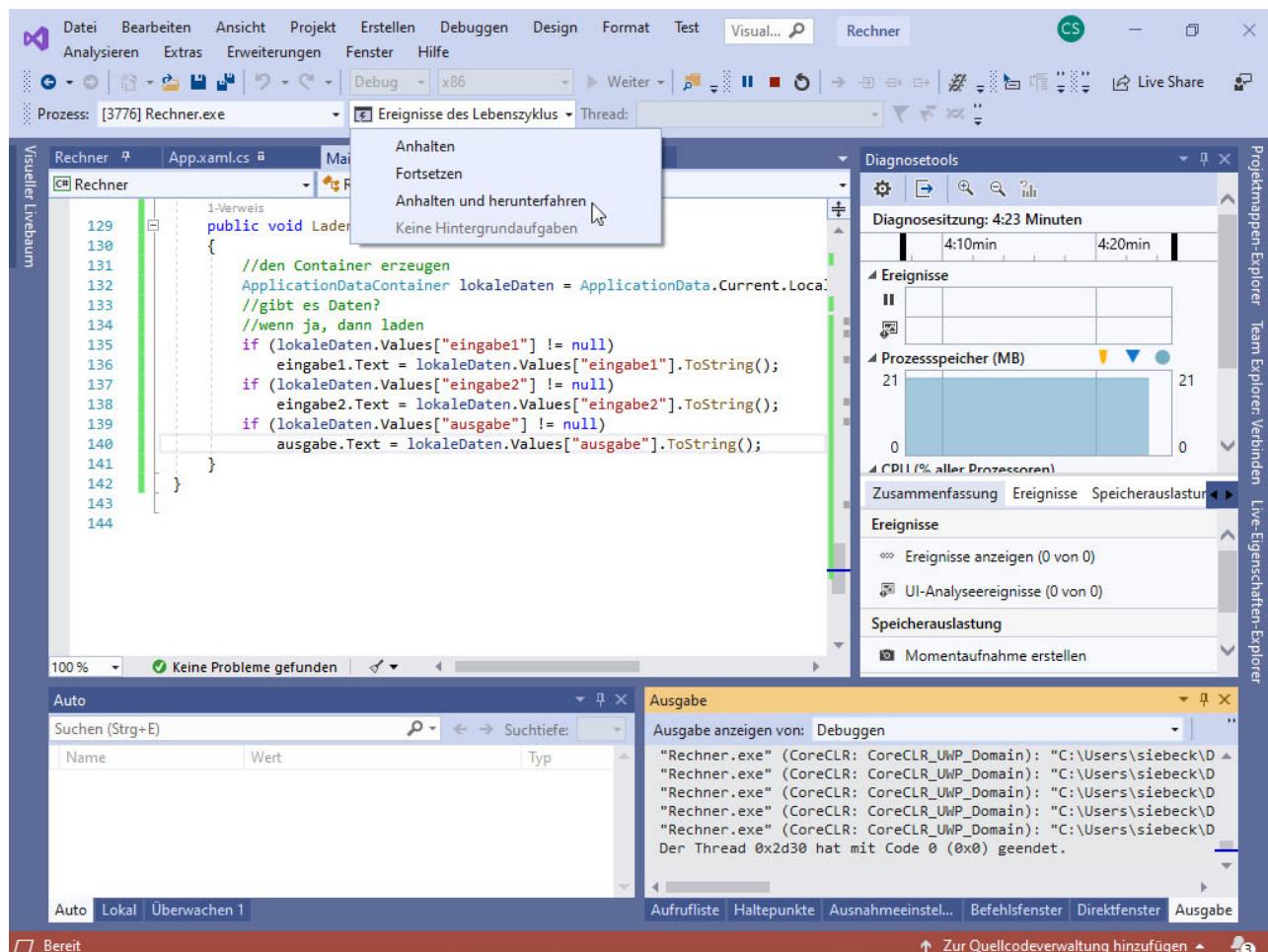


Abb. 3.8: Die Funktion Anhalten und herunterfahren

Wenn Sie die App danach im Simulator neu starten, sollten genau die Daten erscheinen, die auch beim Herunterfahren in den Feldern beziehungsweise dem Textblock standen.



Das Wechseln einer App in den Suspend-Modus darf nicht länger als fünf Sekunden dauern. Sonst wird die App automatisch von Windows beendet. Erledigen Sie Aktionen, die möglicherweise länger dauern, daher nicht beim Wechseln in den Suspend-Modus, sondern bereits vorher.

3.5 Ein wenig Feinschliff

Bevor wir unsere App veröffentlichen, wollen wir noch ein wenig Feinschliff vornehmen.

Im ersten Schritt können Sie die Zeichen in den Eingabefeldern und dem Ausgabefeld größer anzeigen lassen. Das entsprechende Feld finden Sie im Bereich **Text** bei den Eigenschaften.

Wenn Sie wollen, können Sie die Texte außerdem noch rechtsbündig anzeigen lassen. Dazu wechseln Sie im Bereich **Text** im Eigenschaftenfenster in das Register **Absatz**. Über das Feld rechts unten in der zweiten Zeile können Sie dann die Absatzausrichtung festlegen.

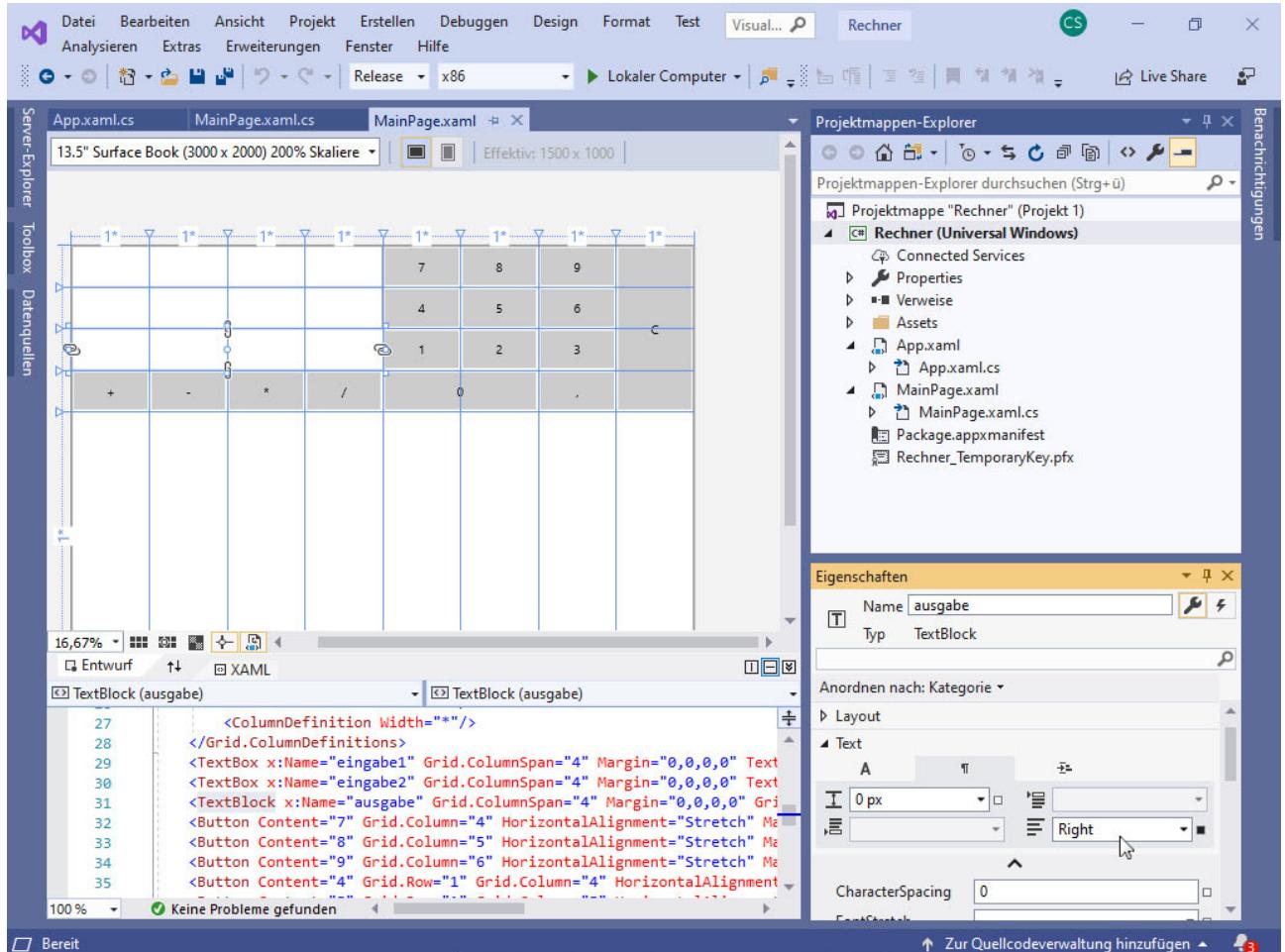


Abb. 3.9: Das Festlegen der Absatzausrichtung (der Mauszeiger steht am Feld für die Ausrichtung)

Ein weiteres Problem lässt sich nicht ganz so einfach ändern. Bisher kann ein Anwender beliebig viele Kommas eingeben. Das führt beim Konvertieren der Zahlen zu einem Fehler. Um das zu verhindern, sorgen wir dafür, dass maximal ein Komma für jedes Eingabefeld verwendet werden kann. Dazu prüfen wir bei der Eingabe eines Kommas mit der Methode `Contains()`, ob sich in dem Feld bereits ein Komma befindet. Die entsprechenden Änderungen an der Methode `Anzeigen()` finden Sie im folgenden Code:

```
//zum Zwischenspeichern der Eingabe
string tempText;
if (aktiveEingabe == 1)
    tempText = eingabe1.Text;
else
    tempText = eingabe2.Text;
```

```
//wurde das Komma angeklickt?
//dann prüfen wir, ob es bereits ein Komma gibt
if (eingabe == "," && tempText.Contains(","))
    return;
else
    tempText = tempText + eingabe;
//und den geänderten Text wieder zurückkopieren
if (aktiveEingabe == 1)
    eingabel.Text = tempText;
else
    eingabe2.Text = tempText;
```

Code 3.16: Die Überprüfung auf ein Komma

Außerdem können Sie auch noch die Logos der Anwendung ändern. Dazu ersetzen Sie die Dateien im Ordner **Assets** durch eigene Bilder. Achten Sie dabei aber unbedingt darauf, dass die Logos eine vorgeschriebene Größe haben müssen. Am besten orientieren Sie sich an den Dimensionen der Vorlagen.



Wenn Sie die Dateien nicht bearbeiten, scheitert später die Zertifizierung der App für die Veröffentlichung.

Auch wenn unsere App an einer oder anderen Stelle noch verbessert werden könnte, wollen wir die Arbeit an dieser Stelle abschließen. Weitere Anpassungen wie zum Beispiel das Verhindern einer Division durch 0 oder detailliertere Fehlermeldungen können Sie ja selbst durchführen.

3.6 Veröffentlichen der App

Abschließend wollen wir unsere App veröffentlichen. Da Sie für die Veröffentlichung im Windows Store ein kostenpflichtiges Entwicklerkonto benötigen, „veröffentlichen“ wir unsere App aber nur lokal.



Das lokale Verteilen einer App wird von Microsoft **Sideload**ing genannt.

Dabei können Sie aber ebenfalls die Prüfungen durchführen, die auch bei einer Veröffentlichung im Windows Store erfolgen. Bei diesen Prüfungen – der Zertifizierung – wird zum Beispiel getestet, ob die App Fehler enthält und einen echten Nutzen für den Anwender bietet.

Hinweis:

Eine detaillierte Übersicht, welche Prüfungen bei der Zertifizierung durchgeführt werden, finden Sie im Internet unter <https://docs.microsoft.com/de-de/windows/uwp/debug-test-perf/windows-app-certification-kit-tests>.

Lassen Sie im ersten Schritt bitte eine Release-Version des Projekts erstellen. Ändern Sie dazu im Kombinationsfeld in der Mitte der Symbolleiste den Eintrag von **Debug** in **Release**. Erstellen Sie das Projekt dann mit der Funktion **Erstellen/Projektmappe neu erstellen** neu. Rufen Sie anschließend die Funktion **Projekt/Store/App-Pakete erstellen...** auf.

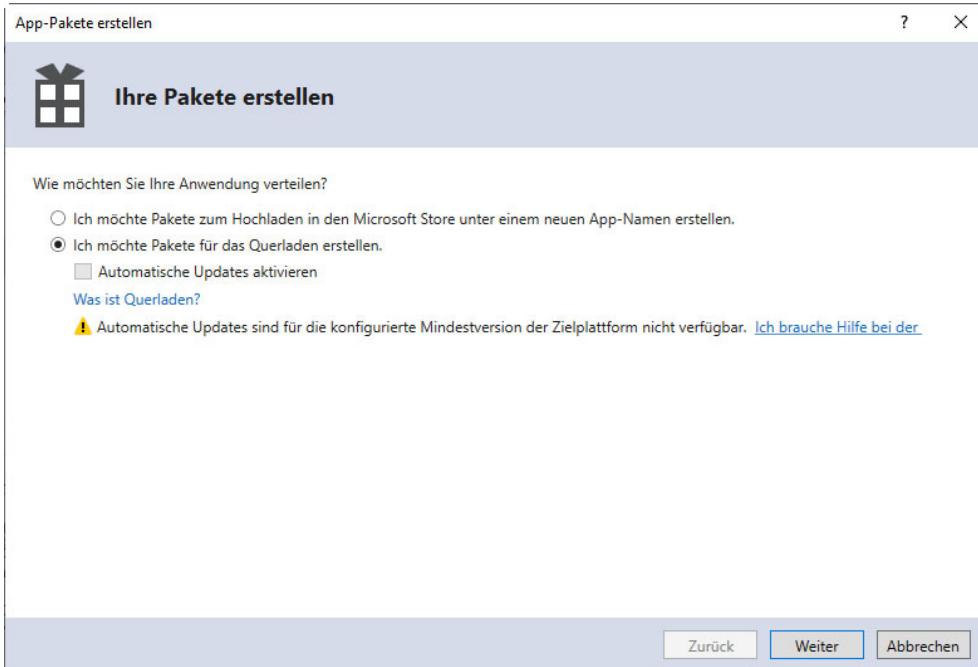


Abb. 3.10: Der Dialog App-Pakete erstellen

Im Dialog App-Pakete erstellen wählen Sie zuerst aus, ob Sie Pakete für den Windows Store oder für das Querladen erstellen möchten. In unserem Fall markieren Sie bitte die Option Ich möchte Pakete für das Querladen erstellen. Klicken Sie dann auf Weiter.

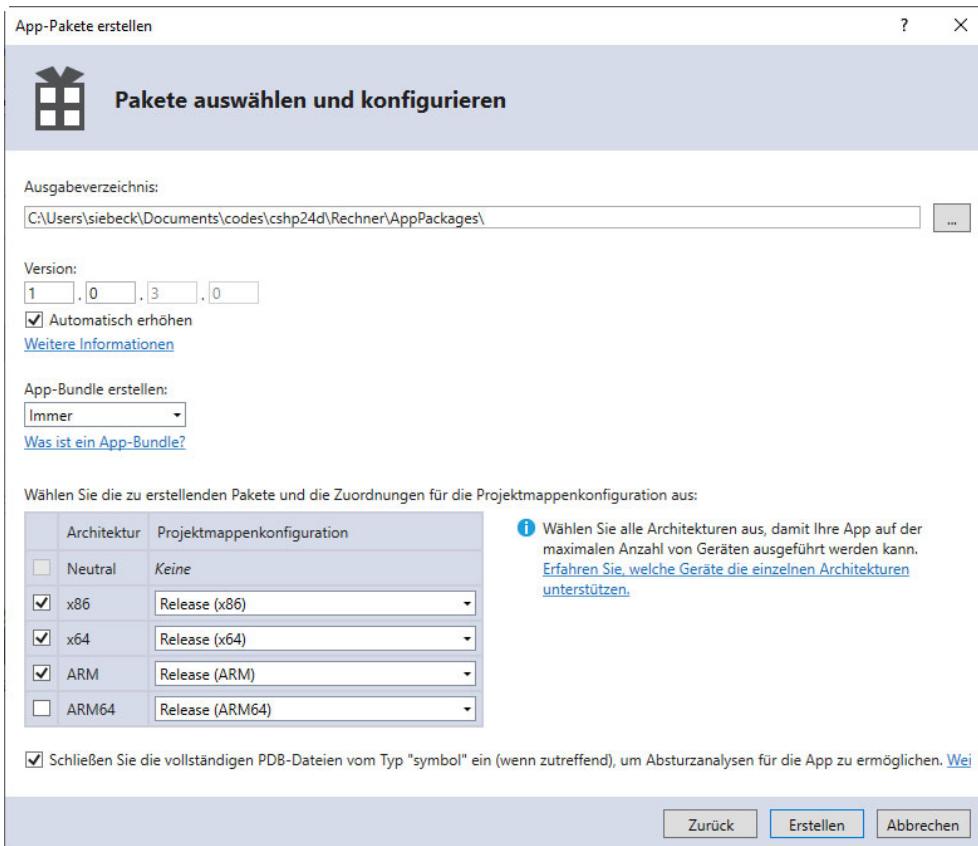


Abb. 3.11: Der zweite Schritt des Assistenten

Im zweiten Schritt des Assistenten legen Sie fest, für welche Umgebungen Pakete erzeugt werden sollen und wo diese Pakete abgelegt werden sollen. Klicken Sie dann auf **Erstellen**.

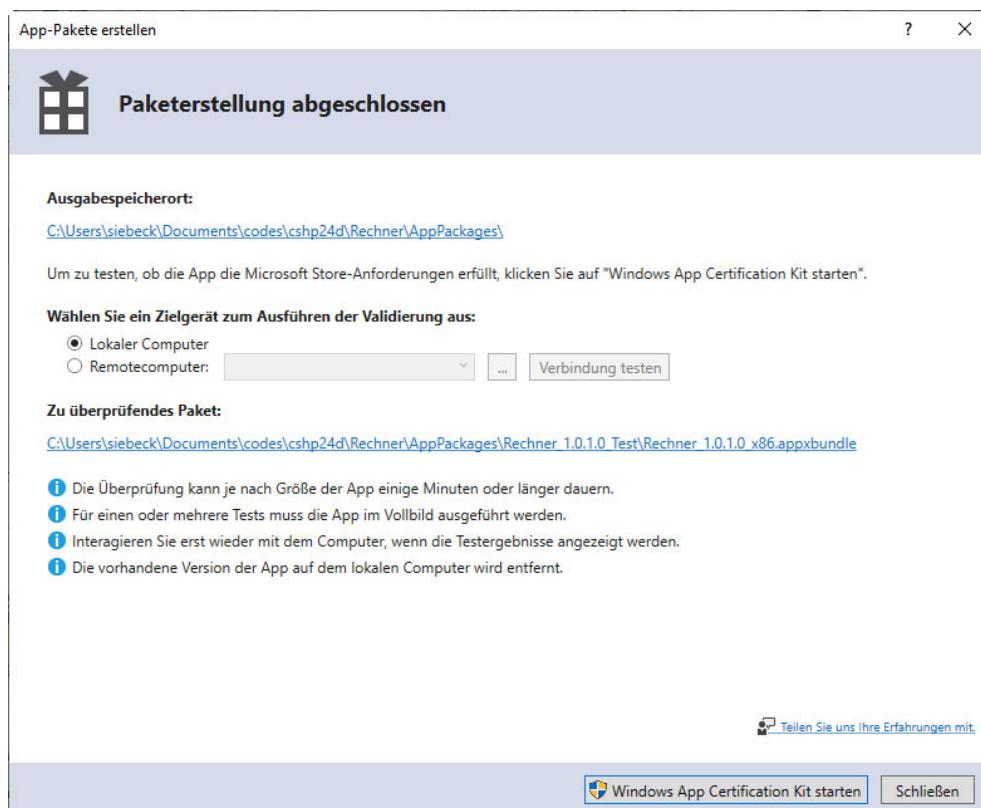


Abb. 3.12: Die Pakete wurden erzeugt

Wenn die Pakete erzeugt wurden, erscheint eine entsprechende Meldung. Über die Schaltfläche **Windows App Certification Kit starten** unten rechts im Fenster können Sie jetzt auch nach einer Abfrage durch die Benutzerkontensteuerung die Prüfung durch das Windows App Certification Kit durchführen lassen.



Wenn Sie eine App ausschließlich lokal installieren wollen, können Sie auf die gesamte Zertifizierung auch verzichten.

Ein Taschenrechner als App

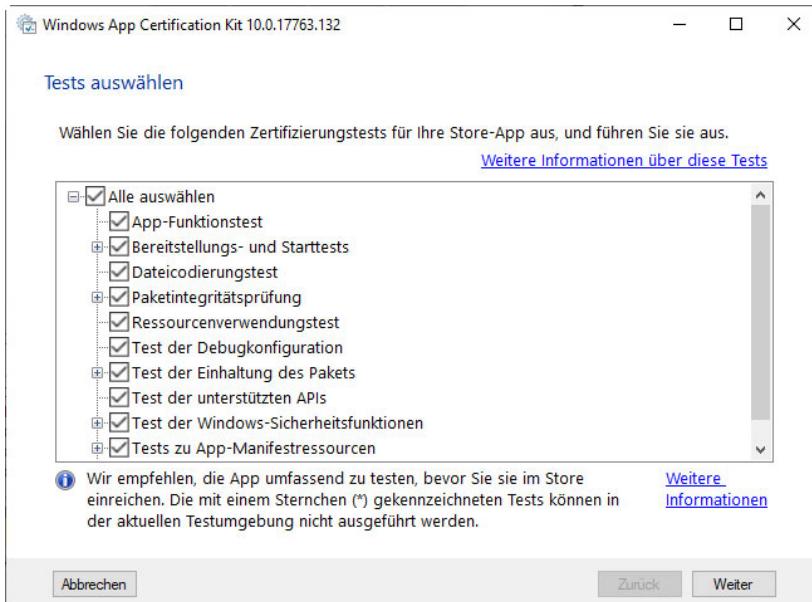


Abb. 3.13: Die Auswahl der Tests

Im nächsten Schritt wählen Sie zunächst die Tests aus, die durchgeführt werden sollen. Wenn Sie sichergehen wollen, sollten Sie immer alle Tests ausführen. Mit einem Klick auf **Weiter** starten Sie dann die eigentliche Prüfung.

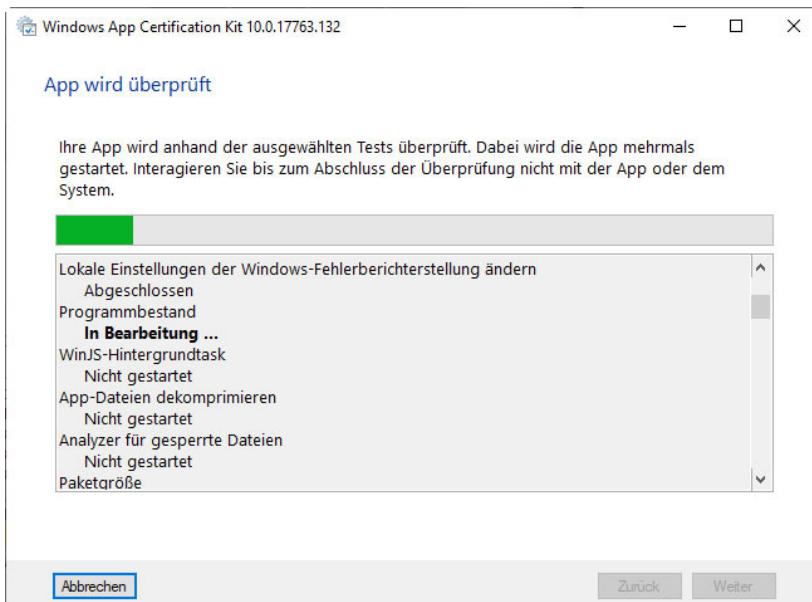


Abb. 3.14: Die Zertifizierung

Diese Prüfung kann durchaus einige Zeit dauern. Dabei wird die App mehrere Mal automatisch gestartet und auch wieder beendet. Bitte greifen Sie in diesen automatischen Ablauf nicht ein, sondern warten Sie geduldig ab.

Hinweis:

Sie können das Windows App Certification Kit auch über das Startmenü im Zweig **Windows Kits** aufrufen. Sie müssen dann allerdings die App, die Sie zertifizieren lassen wollen, selbst im Dialog auswählen.

Am Ende der Prozedur erscheint eine Meldung, ob Ihre App die Prüfungen bestanden hat.

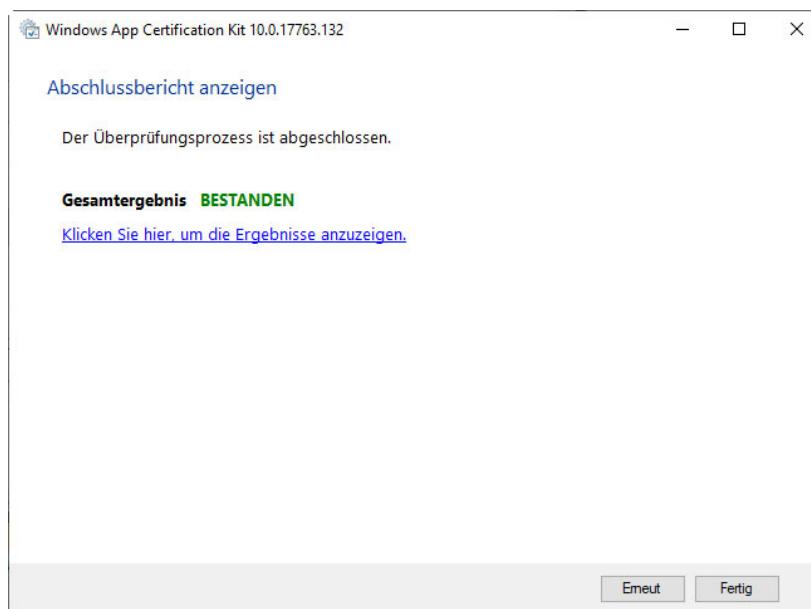


Abb. 3.15: Die Rückmeldung zur Prüfung

Über die Schaltfläche **Fertig** beenden Sie das Erstellen der Pakete. Im Ordner, den Sie als Ziel beim Erstellen der Pakete angegeben haben, finden Sie dann einen Unterordner mit dem Namen der App, der Version und dem Paket.

Ein Taschenrechner als App

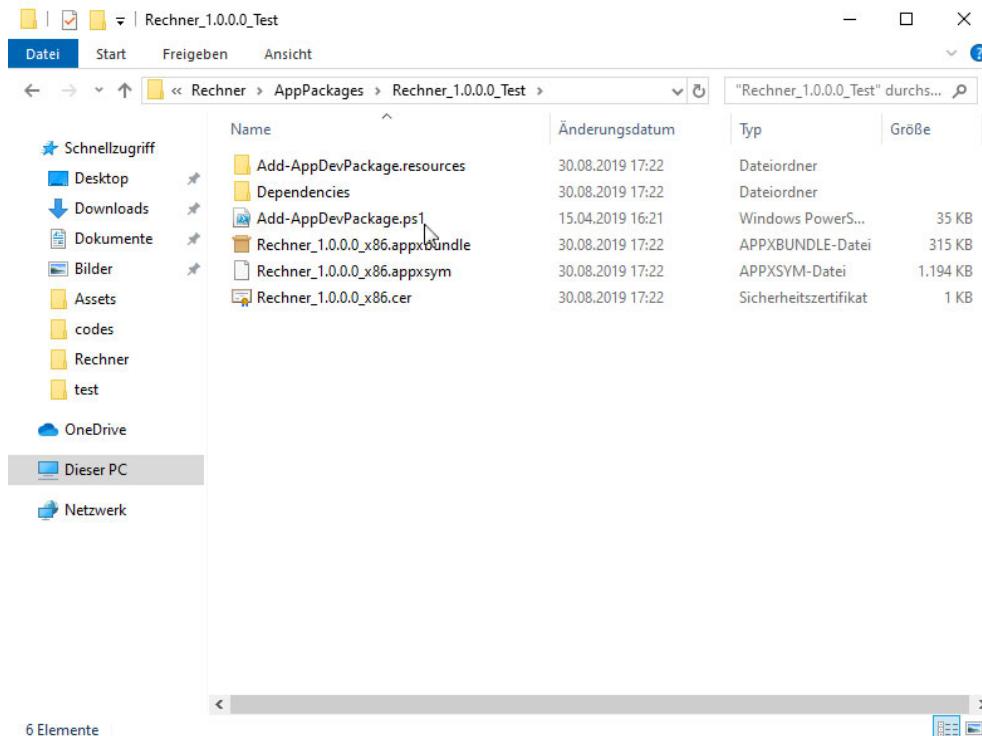


Abb. 3.16: Der Ordner mit dem Paket

Über diesen Ordner können Sie die lokale Installation starten. Dazu verwenden Sie das PowerShell-Installationsskript in dem Ordner. Sie erkennen es an der Dateierweiterung .ps1 und dem Symbol . Um das Skript auszuführen, wählen Sie im Kontextmenü der Datei die Funktion **Mit PowerShell ausführen**.

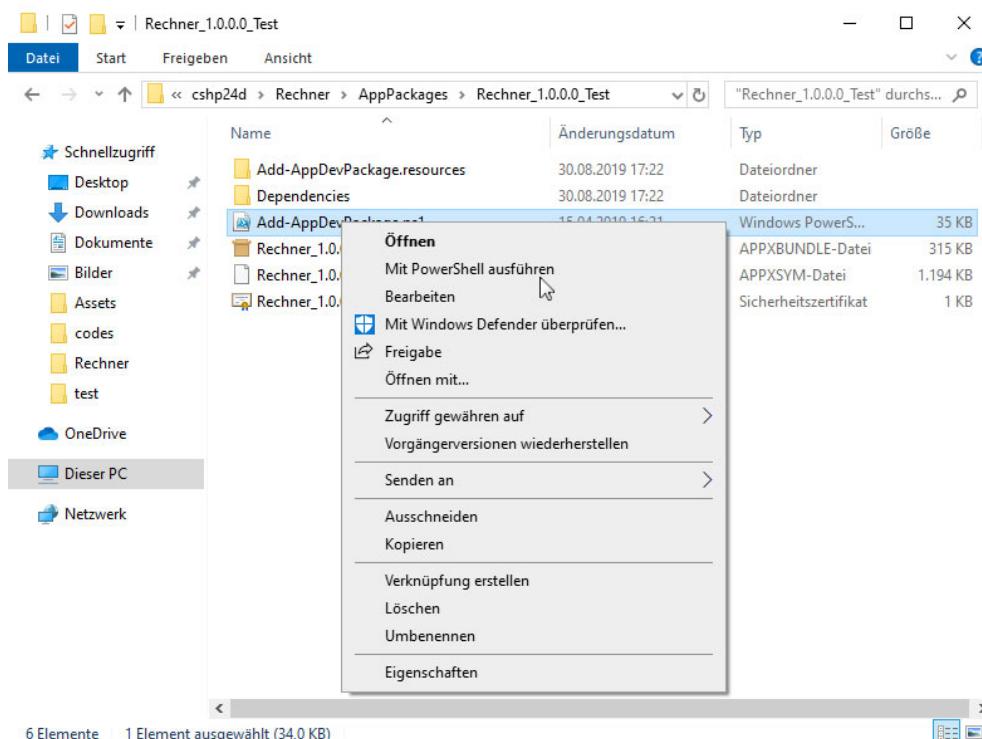


Abb. 3.17: Die Funktion **Mit PowerShell ausführen**

Bei der Installation wird dann nicht nur die App an sich installiert, sondern auch das Zertifikat und gegebenenfalls die Entwicklerlizenz.

Hinweis:

Ein Entwicklerkonto für den Windows Store können Sie mit der Funktion **Projekt/Store/Entwicklerkonto öffnen...** auch direkt aus Visual Studio anlegen. Dabei wird aber lediglich eine Seite im Internet geöffnet.

Bitte beachten Sie aber, dass das Anlegen eines Entwicklerkontos mit Kosten verbunden ist und Sie eine Kreditkarte benötigen.

So viel zu einer ersten etwas anspruchsvoller App. Im nächsten Kapitel werden wir eine kleine App zum Verwalten von Notizen programmieren.

Zusammenfassung

Die Container einer Universal Windows Store App basieren wie bei einer WPF-Anwendung auf XAML.

Mit der Klasse `Windows.UI.Popups.MessageDialog` erzeugen Sie einen Dialog, der sich als eine Art Band über die Oberfläche legt.

Alle zeitintensiven Prozesse sollten bei Universal Windows Apps asynchron ausgeführt werden.

Sie sollten sicherstellen, dass die App nach einem zwangsweisen Beenden durch das System beim nächsten Laden genauso wiederhergestellt wird, wie sie beim Beenden verlassen wurde. Die entsprechenden Abfragen sind bei einer Standardseite bereits vorhanden.

Das Laden und Speichern der Daten erfolgt über die Klasse `ApplicationData`. Hier können Sie auf den Datenspeicher der App zugreifen.

Das eigentliche Speichern und Laden erfolgt in den Methoden `OnSuspending()` und `OnLaunched()` der App-Klasse.

Um eine App im Windows Store zu veröffentlichen, müssen Sie sie zertifizieren lassen. Sie können die Zertifizierung aber auch erst lokal mit dem Windows App Certification Kit durchführen.

Apps, die Sie lokal verteilen, müssen Sie nicht zertifizieren. Hier können Sie direkt ein Paket für die Verteilung erstellen.

Aufgaben zur Selbstüberprüfung

- 3.1 Wie wird die lokale Verteilung einer App bei Microsoft genannt?

- 3.2 Welche beiden Schlüsselwörter sind für die asynchrone Verarbeitung wichtig?
Welche Bedeutung haben die beiden Schlüsselwörter?

- 3.3 Wie können Sie am einfachsten testen, ob Ihre Anweisungen für das Wiederherstellen nach dem zwangsweisen Herunterfahren durch das System korrekt arbeiten?

- 3.4 Wie lange darf der Wechsel in den Suspend-Modus maximal dauern? Was geschieht, wenn die maximale Dauer überschritten wird?

- 3.5 Sie haben für die lokale Verteilung ein Paket erstellt. Wie wird über dieses Paket die Installation durchgeführt?

4 Ein Notizbuch als App

In diesem Kapitel erstellen wir ein kleines Notizbuch als App. Es soll kurze Texte laden und speichern können. Sie werden unter anderem lernen, wie Sie eine Liste mit Daten füllen. Außerdem erfahren Sie, wie Sie mit einer App Textdateien verarbeiten.

4.1 Die Grundfunktionen

Die Grundfunktionen unserer App sind recht einfach. Wir erstellen ein Listenfeld, in dem die vorhandenen Notizen in Kurzform angezeigt werden. In einem Textfeld daneben kann der Anwender eine Notiz, die er im Listenfeld auswählt, komplett anzeigen und bearbeiten. Außerdem sollen neue Notizen angelegt werden können.

Damit der Anwender die Notizen nicht immer wieder neu erstellen muss, kann er sie lokal speichern. Beim Starten der App werden die Daten dann automatisch geladen und angezeigt.

Legen Sie ein neues Projekt für eine leere Universal Windows Platform App mit Visual C# an. Als Namen können Sie zum Beispiel **Notizen** verwenden.

4.2 Die Oberfläche

Für die Oberfläche verwenden wir wieder ein Grid als Container. In dem Grid ordnen wir das Listenfeld, ein Eingabefeld und einige Symbole an.

Legen Sie in der Vorlage eine Tabelle mit vier Spalten und vier Zeilen an. Die ersten drei Zeilen sollen dabei 100 Pixel hoch sein und die letzte Zeile den verbleibenden Rest des Bildschirms einnehmen. Bei den Spalten soll die erste Spalte 150 Pixel breit sein, die zweite und die dritte je 100 Pixel und die letzte Spalte soll den verbleibenden Rest einnehmen.

Die XAML-Anweisungen für die Tabelle sehen entsprechend so aus:

```
<Grid.RowDefinitions>
    <RowDefinition Height="100"/>
    <RowDefinition Height="100"/>
    <RowDefinition Height="100"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="150"/>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
```

Code 4.1: Das Grid

Fügen Sie dann ein Listenfeld ein. Sie finden es in der Toolbox unter dem Eintrag **ListView**. Positionieren Sie das Listenfeld in der Zelle ganz oben links und sorgen Sie dafür, dass es drei Zeilen abdeckt. Setzen Sie außerdem die Ränder auf 0 und ändern Sie die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** auf **Stretch** bezie-

hungsweise löschen Sie diese Eigenschaften im XAML-Code. Geben Sie dem Listenfeld dann noch einen sprechenden Namen. Wir verwenden in unserem Beispiel den Namen `liste`.

Stellen Sie anschließend zum Test ein paar Einträge in die Liste. Dazu verwenden Sie die Eigenschaft `Items` im Bereich **Allgemein**. Legen Sie danach im Auflistungs-Editor ein paar Elemente vom Typ `ListViewItem` an. Den Text setzen Sie dabei jeweils über die Eigenschaft `Content` im Bereich **Allgemein**.

Sie können die Einträge auch sehr schnell über den XAML-Code erstellen. Setzen Sie dazu ein Tag `ListBoxItem` zwischen die Tags `ListView`. Den Text legen Sie über die Eigenschaft `Content` fest. Der XAML-Code für unsere Liste mit fünf Einträgen könnte dann so aussehen:

```
<ListView x:Name="liste" Margin="0,0,0,0" Grid.RowSpan="3">
    <ListViewItem Content="Eintrag 1"/>
    <ListViewItem Content="Eintrag 2"/>
    <ListViewItem Content="Eintrag 3"/>
    <ListViewItem Content="Eintrag 4"/>
    <ListViewItem Content="Eintrag 5"/>
</ListView>
```

Code 4.2: Der XAML-Code für die Liste mit fünf Einträgen

Fügen Sie anschließend in die zweite Spalte der Tabelle oben ein Eingabefeld ein. Es soll zwei Spalten und zwei Zeilen komplett ausfüllen und ein wenig Abstand zum Listenfeld links daneben haben. Geben Sie dem Feld ebenfalls einen eindeutigen Namen – zum Beispiel `anzeige`. Der XAML-Code könnte so aussehen:

```
<!-- bitte in einer Zeile eingeben -->
<TextBox x:Name="anzeige" Grid.Column="1" Grid.Row="0"
Margin="5,0,0,0" TextWrapping="Wrap" Text="" Grid.RowSpan="2"
Grid.ColumnSpan="2"/>
```

Code 4.3: Der XAML-Code für das Eingabefeld

Fügen Sie dann noch je eine Schaltfläche in die zweite und die dritte Spalte der dritten Zeile ein. Positionieren Sie die Schaltflächen so, dass sie ein wenig Abstand zu den Rändern der Zelle haben. Der XAML-Code könnte so aussehen:

```
<!-- bitte jeweils in einer Zeile eingeben -->
<Button Grid.Column="1" Grid.Row="2" Margin="10"
HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
FontSize="36"/>
<Button Grid.Column="2" Grid.Row="2" Margin="10"
HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
FontSize="36"/>
```

Code 4.4: Der XAML-Code für die Schaltflächen

Für die Symbole auf den Schaltflächen verwenden wir die Schriftart **Segoe UI Symbol**. Sie enthält zahlreiche Zeichen, die sehr gut zu einer App unter Windows 10 passen. Die Auswahl ist allerdings ein wenig umständlich, da Sie den Zeichencode zuweisen müssen. Diesen Code finden Sie am einfachsten über die Zeichentabelle von Windows heraus. Sie finden sie im Startmenü im Zweig **Windows-Zubehör**.

Nach dem Start wählen Sie über das Kombinationsfeld oben die Schriftart **Segoe UI Symbol** aus. Ganz unten in der Liste der Zeichen finden Sie dann auch zahlreiche Symbole.

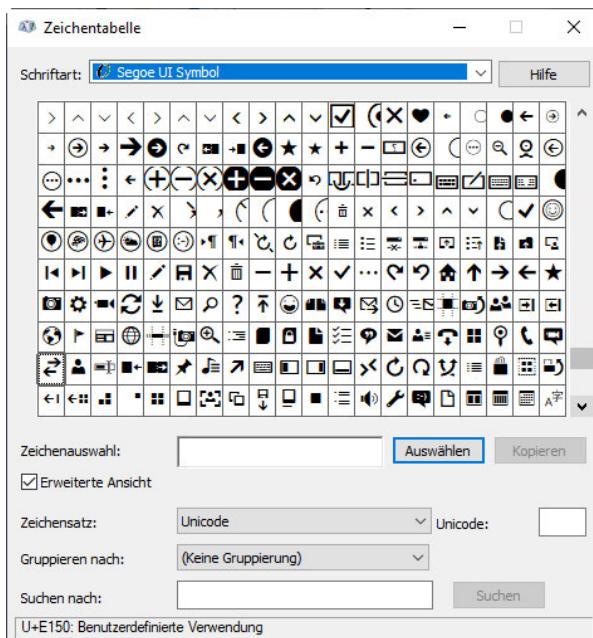


Abb. 4.1: Die Schriftart Segoe UI Symbol

In unserer App benötigen wir ein Symbol zum Speichern und ein Symbol zum Neuerstellen. Dazu können Sie zum Beispiel die Symbole mit den Codes **E105** und **E160** verwenden.

Sie könnten jetzt die Symbole aus der Zeichentabelle kopieren und über das Eigenschaftenfenster zuweisen. Wenn Sie dann die Schriftart entsprechend setzen, sollte das passende Symbol auf der Schaltfläche angezeigt werden. Schneller geht es aber, wenn Sie die Änderung im XAML-Code vornehmen. Dazu weisen Sie der Eigenschaft `FontFamily` die Schriftart `Segoe UI Symbol` zu und setzen die Eigenschaft `Content` auf das gewünschte Zeichen. Damit der Zeichencode benutzt wird und nicht die Zahl, stellen Sie vor den Code die Zeichen `&#x`. Außerdem müssen Sie die Eingabe mit einem Semikolon abschließen.

Vergeben Sie danach noch sprechende Namen an die Schaltflächen – zum Beispiel `neu` und `speichern`. Der vollständige XAML-Code für die Schaltflächen mit den Symbolen sieht so aus:

```
<!-- bitte jeweils in einer Zeile eingeben -->
<Button x:Name="neu" Grid.Column="1" Grid.Row="2" Margin="10"
HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
FontSize="36" FontFamily="Segoe UI Symbol" Content=""/>
<Button x:Name="speichern" Grid.Column="2" Grid.Row="2"
Margin="10" HorizontalAlignment="Stretch"
VerticalAlignment="Stretch" FontSize="36" FontFamily="Segoe UI
Symbol" Content=""/>
```

Code 4.5: Der XAML-Code für die Schaltflächen mit Symbolen

Die Oberfläche sollte nach diesen Zuweisungen ungefähr so aussehen:

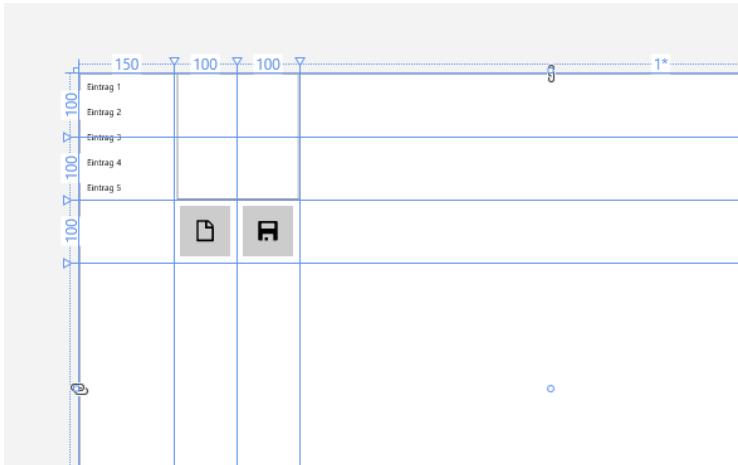


Abb. 4.2: Die Schaltflächen mit den Symbolen

Testen Sie die App auch einmal.

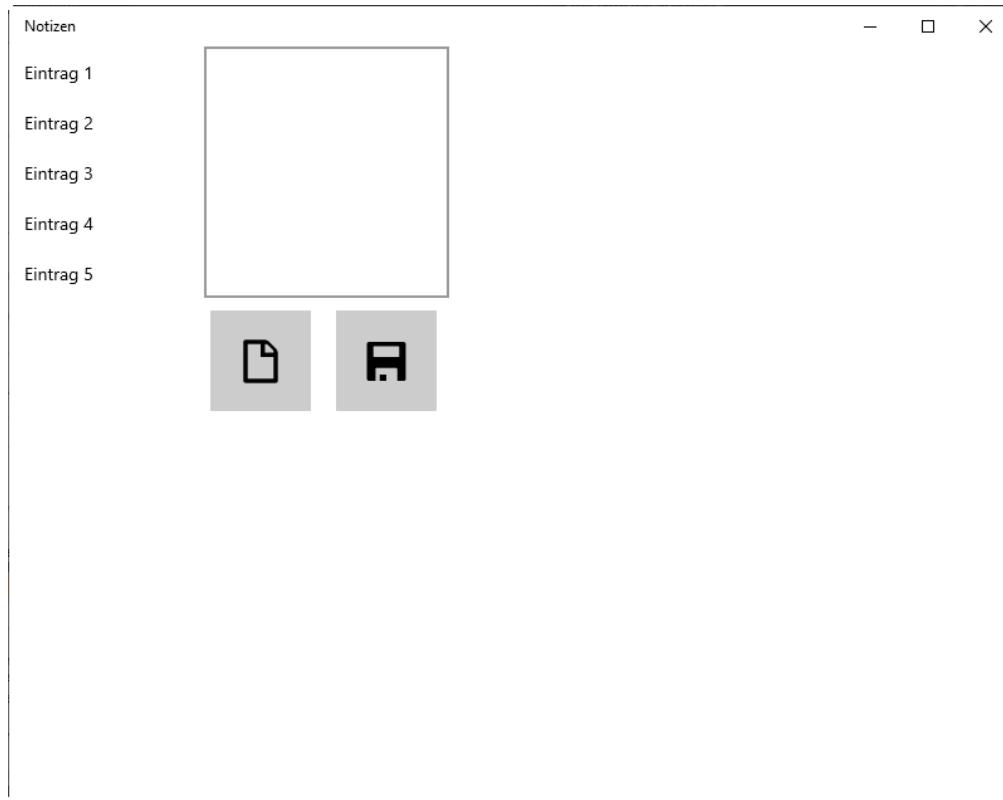


Abb. 4.3: Ein erster Test der App

4.3 Die Logik

Kommen wir jetzt zur Logik für die Anwendung. Die ersten Schritte sind nicht sonderlich schwierig. Beim Auswählen eines Elements in der Liste kopieren wir den Text des Elements in das Eingabefeld. Dazu verwenden wir das Ereignis **SelectionChanged** für das Listenfeld. Damit Änderungen an dem Text möglich werden, kopieren wir den Text wieder zurück, sobald es Änderungen im Eingabefeld gegeben hat. Das soll im Ereignis **TextChanged** für das Eingabefeld erfolgen.

Die entsprechenden Methoden sehen so aus:

```
private void Liste_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    //den Eintrag aus dem markierten Listenelement kopieren
    if (liste.SelectedIndex != -1)
        //bitte in einer Zeile eingeben
        anzeigen.Text = ((ListViewItem)(liste.SelectedItem)).Content.ToString();
}
```

```
private void Anzeige_TextChanged(object sender,
TextChangedEventArgs e)
{
    //den Eintrag aus dem Textfeld in das Listenelement kopieren
    if (liste.SelectedIndex != -1)
        ((ListViewItem)(liste.SelectedItem)).Content = anzeige.Text;
}
```

Code 4.6: Die Methoden zum Anzeigen der Einträge

In beiden Methoden prüfen wir zunächst, ob überhaupt ein Element in der Liste markiert ist. Anschließend kopieren wir den Text aus dem Listenelement in das Eingabefeld beziehungsweise aus dem Eingabefeld in das Listenelement. Da die Eigenschaft `SelectedItem` den sehr allgemeinen Typ `object` liefert, müssen wir das ausgewählte Element vor dem Zugriff noch in den Typ `ListViewItem` konvertieren. Danach können wir uns dann den Text über `Content.ToString()` beschaffen beziehungsweise direkt auf die Eigenschaft `Content` zugreifen.

Auch das Neuanlegen eines Eintrags ist nicht sonderlich schwierig. Wir hängen hier beim Anklicken der Schaltfläche einen leeren Eintrag mit dem Text `<leer>` an das Ende der Liste an und setzen die Markierung auf diesen Eintrag. Dadurch wird dann auch der Text in das Eingabefeld gesetzt.

Die entsprechende Methode sieht so aus:

```
private void Neu_Click(object sender, RoutedEventArgs e)
{
    //einen neuen Eintrag erzeugen und anhängen
    ListViewItem eintrag = new ListViewItem();
    eintrag.Content = "<leer>";
    liste.Items.Add(eintrag);
    //den letzten Eintrag markieren
    liste.SelectedIndex = liste.Items.Count - 1;
}
```

Code 4.7: Das Erzeugen eines neuen Eintrags

Damit der Ausdruck für das neue Element nicht zu kompliziert wird, gehen wir hier einen kleinen Umweg über eine lokale Variable vom Typ `ListViewItem`. Sie könnten das neue Element aber auch direkt an die Methode `Add()` übergeben.

Jetzt kommt der komplizierteste Teil der Programmierung – das Speichern und Laden der Daten. Denn hier gibt es einige Besonderheiten. So kann eine App durch das Sandbox-Prinzip nicht ohne Weiteres auf beliebige Dateien beziehungsweise Ordner zugreifen. Ihr stehen zunächst einmal nur einige vordefinierte Ordner, wie zum Beispiel das Installationsverzeichnis oder ein lokales Datenverzeichnis, zur Verfügung. Außerdem müssen alle Dateioperationen asynchron erfolgen, um die Ausführung der App nicht zu unterbrechen. Daher müssen Sie für das Lesen und Schreiben auch andere Klassen wie zum Beispiel `Windows.Storage.FileIO` verwenden.

Schauen wir uns zuerst das Speichern an. Hier erzeugen wir über die Klassen eine neue Textdatei im lokalen Datenverzeichnis der App, schreiben die Inhalte des Listenfelds in eine Liste vom Typ `string` und speichern diese Liste dann zeilenweise über die

Methode `FileIO.WriteLineAsync()`. Die komplette Methode für das Anklicken der Schaltfläche zum Speichern finden Sie im folgenden Code. Die Besonderheiten erklären wir Ihnen im Anschluss.

```
private async void Speichern_Click(object sender,
RoutedEventArgs e)
{
    //eine neue Datei anlegen
    //den lokalen Ordner beschaffen
    //bitte in einer Zeile eingeben
    StorageFolder lokalerOrdner =
        ApplicationData.Current.LocalFolder;
    //die Datei erzeugen und ggf. überschreiben
    //bitte in einer Zeile eingeben
    StorageFile datei = await lokalerOrdner.CreateFileAsync(
        "notizen.txt", CreationCollisionOption.ReplaceExisting);
    //konnte die Datei angelegt werden?
    if (datei != null)
    {
        //dann die Einträge anhängen
        //dazu benutzen wir eine Listenstruktur
        List<string> tempListe = new List<string>();
        foreach (ListViewItem eintrag in liste.Items)
            tempListe.Add(eintrag.Content.ToString());
        //die Zeilen in die Datei schreiben
        await FileIO.WriteLineAsync(datei, tempListe);
    }
}
```

Code 4.8: Das Speichern der Einträge

Hinweis:

Damit die Klassen `StorageFolder`, `StorageFile` und `FileIO` bekannt sind, müssen Sie den Namensraum `Windows.Storage` einbinden.

Im Kopf der Methode müssen Sie das Schlüsselwort `async` ergänzen. Sonst kann die Anweisung mit `await` nicht übersetzt werden.

Mit den beiden Anweisungen

```
StorageFolder lokalerOrdner =
    ApplicationData.Current.LocalFolder;
StorageFile datei = await
    lokalerOrdner.CreateFileAsync("notizen.txt",
    CreationCollisionOption.ReplaceExisting);
```

beschaffen wir uns über den Ausdruck `ApplicationData.Current.LocalFolder` den lokalen Ordner für die App und speichern ihn in einer Instanz `lokalerOrdner` der Klasse `StorageFolder`. In dem Ordner erzeugen wir mit der Methode `CreateFileAsync()` eine neue Datei mit dem Namen **notizen.txt** und speichern sie in einer Instanz `datei` der Klasse `StorageFile`. Falls die Datei bereits vorhanden ist, soll sie überschrieben werden. Daher geben wir als zweiten Parameter `CreationCollisionOption.ReplaceExisting` an.

Wenn die Datei angelegt werden konnte – `datei` also ungleich `null` ist –, erzeugen wir mit der Anweisung

```
List<string> tempListe = new List<string>();
```

eine Liste für den Typ `string` und schreiben mit der Schleife

```
foreach (ListViewItem eintrag in liste.Items)
    tempListe.Add(eintrag.Content.ToString());
```

die Texte aller Einträge aus dem Listenfeld in die Liste.

Zur Erinnerung:

Mit dem Typ `List` können Sie sehr einfach Listen für nahezu beliebige Datentypen erstellen.



Danach werden alle Einträge der Liste mit der Anweisung

```
await FileIO.WriteLineAsync(datei, tempListe);
```

in einem Rutsch zeilenweise in die Datei geschrieben. Die Methode `WriteLinesAsync()` erwartet dabei zuerst die Datei, in die geschrieben werden soll, und danach die Liste, die verarbeitet werden soll.

Das Zurücklesen der gespeicherten Daten erfolgt mit einer ähnlichen Technik. Wir erstellen zunächst eine Liste vom Typ `string` und laden über die Methode `FileIO.ReadLinesAsync()` die Daten zeilenweise aus der Datei in diese Liste. Falls die Datei nicht vorhanden ist oder andere Probleme aufgetreten sind, stellen wir über unsere Methode für das Anklicken der Schaltfläche für einen neuen Eintrag einen leeren Eintrag in die Liste. Dazu prüfen wir in einer `try-catch`-Konstruktion, ob das Laden der Datei eine Ausnahme ausgelöst hat.

Damit das Laden automatisch geschieht, lassen wir es im Ereignis **Loaded** für die Seite unserer App ausführen. Die komplette Methode zum Zurücklesen sieht dann so aus:

```
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        //den lokalen Ordner beschaffen
        //bitte jeweils in einer Zeile eingeben
        StorageFolder lokalerOrdner =
            ApplicationData.Current.LocalFolder;
        //die Datei beschaffen
        StorageFile datei =
            await lokalerOrdner.GetFileAsync("notizen.txt");
        if (datei != null)
        {
            //die Daten zurücklesen
            IList<string> tempListe = new List<string>();
            tempListe = await FileIO.ReadLinesAsync(datei);
```

```

foreach (string eintragText in tempListe)
{
    //einen neuen Eintrag erzeugen und anhängen
    ListViewItem eintrag = new ListViewItem();
    eintrag.Content = eintragText;
    liste.Items.Add(eintrag);
}
//den letzten Eintrag markieren
liste.SelectedIndex = liste.Items.Count - 1;
}
//wenn es Probleme gegeben hat, hängen wir einen leeren
//Eintrag an die Liste an
catch
{
    neu_Click(sender, e);
}
}

```

Code 4.9: Das Zurücklesen

Hinweis:

Denken Sie auch hier an das Schlüsselwort `async` im Kopf der Methode.

Im `try`-Zweig beschaffen wir uns wieder den Ordner und die Datei. Dabei gibt es keine Besonderheiten.

Danach erzeugen wir die Liste vom Typ `string`. Da die Methode `FileIO.ReadLinesAsync()` den Typ `IList` zurückliefert, verwenden wir diesmal aber nicht `List`, sondern `IList`.

Hinweis:

`IList` ist eine Schnittstelle, die eine Sammlung von Objekten darstellt. Die Elemente in der Sammlung können wie bei einem Array über den Index angesprochen werden.

In der `foreach`-Schleife erzeugen wir dann für jeden Eintrag in der Liste ein neues Element für das Listenfeld und hängen es mit der Methode `Add()` an. Abschließend setzen wir die Markierung in der Liste auf das letzte Element.

Wenn es Probleme gegeben hat, erzeugen wir im `catch`-Zweig einen neuen leeren Eintrag über unsere Methode `Neu_Click()`.

Übernehmen Sie jetzt die Änderungen aus den vorigen Codes und testen Sie das Programm. Sie sollten die Daten aus der Liste jetzt speichern und auch wieder laden können. Allerdings werden dabei die Daten immer an die Einträge angehängt, die wir über die Eigenschaft `Items` gesetzt haben. Sie sollten diese Einträge also löschen.

Damit wollen wir die Arbeit an unserer App beenden – auch wenn zum Beispiel die Reaktion auf das Beenden durch das System noch komplett fehlt. Entsprechende Erweiterungen können Sie ja selbst durchführen. Denken Sie dabei aber bitte daran, dass nach den Vorgaben von Microsoft nicht mehr als fünf Sekunden beim Wechsel in den Suspend-Modus vergehen dürfen. Sie sollten sich daher überlegen, ob Sie das Speichern der

Daten direkt nach jeder Änderung automatisch durchführen – und nicht erst beim Wechsel in den Suspend-Modus. Das ist zwar viel Aufwand, stellt aber sicher, dass der Suspend-Modus ohne Verzögerung erreicht werden kann.

Hinweis:

Wir haben in unserem Beispiel auf eine strikte Trennung der Daten und der Oberfläche verzichtet, da es sich nur um eine sehr einfache Anwendung handelt. Für komplexere Anwendungen sollten Sie überlegen, ob Sie die Verwaltung der Daten in der Liste nicht komplett in einer eigenen Klasse durchführen und die Liste nur zur Anzeige verwenden. Eine andere Möglichkeit besteht darin, die Liste über die Klasse `ObservableCollection` zu verwalten. Hier wird die Listendarstellung automatisch bei Änderungen der zugrunde liegenden Liste aktualisiert. Mehr zur Klasse `ObservableCollection` finden Sie ebenfalls in der Hilfe des .NET Frameworks.

Zusammenfassung

Mit dem Steuerelement `ListView` können Sie Listen erstellen.

Für die Symbole auf den Schaltflächen in einer Universal Windows Platform App können Sie zum Beispiel die Schriftart **Segoe UI Symbol** verwenden.

Für das Laden und Speichern in Dateien können Sie die Klasse `Windows.Storage.FileIO` verwenden.

Über die Klassen `StorageFolder` und `StorageFile` können Sie auf Ordner zugreifen und Dateien anlegen.

Mit den Methoden `WriteLinesAsync()` und `ReadLinesAsync()` der Klasse `FileIO` können Sie Zeilen in einer Liste komplett in eine Datei schreiben beziehungsweise komplett aus einer Datei lesen.

Aufgaben zur Selbstüberprüfung

- 4.1 Beschreiben Sie kurz, wie Sie ein Symbol über den Zeichencode einer Schaltfläche zuweisen können.

- 4.2 Mit welchem Ausdruck erhalten Sie den lokalen Ordner einer App?

- 4.3 Notieren Sie die Anweisungen, um eine Datei **test.txt** im lokalen Ordner einer App anzulegen. Wenn die Datei bereits vorhanden ist, soll sie überschrieben werden.

Schlussbetrachtung

In diesem Studienheft haben Sie sich mit der Entwicklung von Universal Windows Platform Apps beschäftigt. Sie kennen jetzt die wichtigsten Unterschiede zu Windows-Forms- und WPF-Anwendungen und wissen, welche Grundprinzipien Sie beim Entwickeln von Universal Windows Platform Apps berücksichtigen müssen. Zum Teil haben Sie diese Grundprinzipien auch praktisch in den Apps in diesem Studienheft umgesetzt – zum Beispiel bei der asynchronen Verarbeitung von Daten und bei der asynchronen Anzeige von Informationen.

In diesem Studienheft haben wir viele Dinge nur kurz gestreift und einige Aspekte überhaupt nicht besprochen. Wenn Sie sich weiter mit dem Thema Universal Windows Platform Apps beschäftigen wollen, finden Sie zahlreiche Informationen in dem Buch von Cordts und Nasutta bei den Literaturempfehlungen. Auch im Internet gibt es zahlreiche Quellen für weitere Informationen – zum Beispiel im Developer Center für Windows Apps von Microsoft. Sie finden es auf den Seiten <https://docs.microsoft.com/de-de/windows/uwp/>.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

- 1.1 Universal Windows Platform Apps weisen folgende Besonderheiten auf:
 - Die Basis einer App bilden Seiten.
 - Die App soll möglichst einfach und sachlich erscheinen.
 - Die App soll einfach über Gesten bedient werden können.
 - Die App soll schnell und flüssig arbeiten und dem Anwender eine Rückmeldung über seine Aktivitäten geben.

Für die richtige Lösung reicht es aus, wenn Sie die drei Punkte genannt haben.

- 1.2 Eine Sandbox ist ein streng abgeschirmter Bereich, auf den nur die App Zugriff hat. Außerhalb der Sandbox kann die App keine Aktionen ausführen, wenn der Anwender nicht ausdrücklich zustimmt.
- 1.3 Capabilities beschreiben Zugriffsregeln für Apps.

Kapitel 2

- 2.1 Im Ordner **Assets** finden Sie unter anderem Logos für die App.
- 2.2 Um den Simulator zu benutzen, ändern Sie die Einstellung im Kombinationsfeld in der Mitte der Symbolleiste von **Lokaler Computer** in **Simulator**. Nach dem Start der App wird sie automatisch im Simulator ausgeführt.
- 2.3 Um den Simulator für das Ausführen einer App zu beenden, drücken Sie die Tastenkombination **Strg + Alt + F4**.

Kapitel 3

- 3.1 Die lokale Verteilung einer App wird bei Microsoft Sideloadung genannt.
- 3.2 Für die asynchrone Verarbeitung sind zwei Schlüsselwörter wichtig: `async` und `await`. Das Schlüsselwort `async` verwenden Sie für Methoden, die asynchrone Verarbeitungen durchführen. Mit dem Schlüsselwort `await` können Sie auf das Ergebnis der asynchronen Verarbeitung warten und es auswerten.
- 3.3 Um die Anweisungen zu testen, benutzen Sie im Debugger des Simulators die Funktion **Anhalten und Herunterfahren**.
- 3.4 Das Wechseln einer App in den Suspend-Modus darf nicht länger als fünf Sekunden dauern. Sonst wird die App automatisch von Windows beendet.
- 3.5 Um die Installation durchzuführen, lassen Sie das PowerShell-Skript in dem Ordner mit dem Paket ausführen. Dazu benutzen Sie die Funktion **Mit PowerShell ausführen** im Kontextmenü des Skripts.

Kapitel 4

- 4.1 Entweder kopieren Sie das Symbol aus der Zeichentabelle von Windows oder Sie weisen den Zeichencode direkt zu. Damit der Code benutzt wird, müssen Sie vor den numerischen Wert noch die Zeichen &#x stellen.
- 4.2 Den lokalen Ordner einer App erhalten Sie über den Ausdruck ApplicationData.Current.LocalFolder.
- 4.3 Die Anweisungen könnten so aussehen:

```
StorageFolder lokalerOrdner =  
ApplicationData.Current.LocalFolder;  
StorageFile datei = await  
lokalerOrdner.CreateFileAsync("test.txt",  
CreationCollisionOption.ReplaceExisting);
```

B. Glossar

Capabilities	<i>Capabilities</i> beschreiben die Zugriffsregeln einer Universal Windows Platform App.
Code-Behind-Dateien	Die <i>Code-Behind</i> -Dateien sind die Quelltextdateien für die Programmlogik einer WPF-Anwendung oder einer Universal Windows Platform App.
Container	Ein Container nimmt weitere Steuerelemente in einer WPF-Anwendung oder einer Universal Windows Platform App auf.
Contracts	<i>Contracts</i> werden die Verträge für die Zusammenarbeit mehrerer Apps genannt.
Frame	Ein Frame dient als Container für die Navigation zwischen Seiten einer Universal Windows Platform App.
HTML	HTML steht für <i>Hypertext Markup Language</i> . HTML ist eine Auszeichnungssprache, die im Internet eingesetzt wird.
Hypertext Markup Language	Siehe HTML.
Microsoft Store	Beim Microsoft Store handelt es sich um eine Art virtuellen Marktplatz im Internet, auf dem unterschiedlichste Apps angeboten werden.
Sandbox	Eine Sandbox ist ein streng abgeschirmter Bereich eines Systems, der nur einer ganz bestimmten Anwendung zur Verfügung steht. Außerhalb der Sandbox kann die Anwendung keine Aktionen ausführen, wenn der Benutzer nicht ausdrücklich zustimmt.
Sideloadung	Sideloadung wird das lokale Verteilen einer Universal Windows Platform App genannt.
Simulator	Ein Simulator versucht, wesentliche Aspekte eines Systems nachzubilden.
Splash Screen	Ein Splash Screen ist ein Startbildschirm.
Suspend-Modus	Suspend-Modus wird der Schlafzustand einer App genannt.

Universal Windows Platform App

Eine Universal Windows Platform App ist eine Anwendung, die auf mobilen Geräten wie einem Tablet Computer oder einem Smartphone, aber auch auf einem normalen Rechner eingesetzt wird. Universal Windows Platform Apps können nur unter Windows 10 eingesetzt und entwickelt werden.

Windows App Certification Kit

Über das Windows App Certification Kit können Sie die Zertifizierung einer Universal Windows Platform App durchführen.

Windows Phone

Windows Phone ist eine spezielle Version von Windows für Smartphones.

Windows RT

Windows RT ist eine Windows Version für mobile Geräte.

C. Literaturverzeichnis

Empfohlene Literatur

- Cordts, S. & Nasutta, M. (2016).
Apps für Windows 10 in C#.
3. überarbeitete und erweiterte Aufl., Heide: Mana-Buch.
- Kühnel, A. (2019).
C# 8 mit Visual Studio: Das umfassende Handbuch.
Spracheinführung, Objektorientierung, Programmiertechniken.
8. Aufl., Bonn: Rheinwerk.
- Theis, T. (2019).
Einstieg in C# mit Visual Studio 2019.
Ideal für Programmieranfänger.
6. Aufl., Bonn: Rheinwerk.

Internet-Links

Microsoft Developer Center für Apps
<https://docs.microsoft.com/de-de/windows/uwp/>

D. Abbildungsverzeichnis

Abb. 1.1	Typische Universal Windows Platform App (hier: Edge von Microsoft)	3
Abb. 1.2	Der Microsoft Store	4
Abb. 1.3	Eine einfache und sachliche App (hier die Fotos von Windows 10)	5
Abb. 1.4	Die Berechtigungen für eine App	6
Abb. 2.1	Das Fenster Neues Projekt erstellen	10
Abb. 2.2	Das Fenster Neues UWP-Projekt	11
Abb. 2.3	Die Entwicklerfunktionen bei den Einstellungen	11
Abb. 2.4	Das neue Projekt	12
Abb. 2.5	Ein eingefügter Textblock	13
Abb. 2.6	Die erste App	14
Abb. 2.7	Der Eintrag der App im Startmenü (links am Mauszeiger)	15
Abb. 2.8	Die Auswahl der Geräte und Auflösungen (links in der Abbildung)	16
Abb. 2.9	Die Einstellungen für den Test im Simulator (in der Mitte der Symbolleiste am Mauszeiger; die Einstellungen sind bereits geändert)	17
Abb. 2.10	Die App im Simulator	18
Abb. 3.1	Die Taschenrechner-App	20
Abb. 3.2	Das Grid in der Seite	22
Abb. 3.3	Das Eingabefeld	23
Abb. 3.4	Die Schaltflächen für die Ziffern	25
Abb. 3.5	Die Funktion Gehe zu Definition im Kontextmenü	28
Abb. 3.6	Ein erster Test der App	28
Abb. 3.7	Der Dialog	31
Abb. 3.8	Die Funktion Anhalten und herunterfahren	36
Abb. 3.9	Das Festlegen der Absatzausrichtung (der Mauszeiger steht am Feld für die Ausrichtung)	37
Abb. 3.10	Der Dialog App-Pakete erstellen	39
Abb. 3.11	Der zweite Schritt des Assistenten	39
Abb. 3.12	Die Pakete wurden erzeugt	40
Abb. 3.13	Die Auswahl der Tests	41
Abb. 3.14	Die Zertifizierung	41
Abb. 3.15	Die Rückmeldung zur Prüfung	42
Abb. 3.16	Der Ordner mit dem Paket	43
Abb. 3.17	Die Funktion Mit PowerShell ausführen	43

Abb. 4.1	Die Schriftart Segoe UI Symbol	48
Abb. 4.2	Die Schaltflächen mit den Symbolen	49
Abb. 4.3	Ein erster Test der App.....	50

E. Codeverzeichnis

Code 3.1	Das Grid	21
Code 3.2	Das zweite Eingabefeld	23
Code 3.3	Der Textblock	23
Code 3.4	Die erste Schaltfläche	24
Code 3.5	Die Schaltflächen für die Rechenarten	25
Code 3.6	Die Schaltfläche für die Ziffer 0 und die Schaltfläche zum Löschen	26
Code 3.7	Die Methode für das Anklicken der Ziffern	27
Code 3.8	Die Methode Anzeigen()	27
Code 3.9	Die Methode für die Schaltflächen mit den Rechenoperationen	29
Code 3.10	Die Methoden für die Rechenoperationen.....	29
Code 3.11	Die Methode FehlerMeldung()	30
Code 3.12	Die Methode OnLaunched()	32
Code 3.13	Das Laden und Speichern der Daten.....	33
Code 3.14	Die Methode OnLaunched().....	35
Code 3.15	Die Methode OnSuspending()	35
Code 3.16	Die Überprüfung auf ein Komma	38
Code 4.1	Das Grid	46
Code 4.2	Der XAML-Code für die Liste mit fünf Einträgen	47
Code 4.3	Der XAML-Code für das Eingabefeld	47
Code 4.4	Der XAML-Code für die Schaltflächen	47
Code 4.5	Der XAML-Code für die Schaltflächen mit Symbolen	49
Code 4.6	Die Methoden zum Anzeigen der Einträge	51
Code 4.7	Das Erzeugen eines neuen Eintrags	51
Code 4.8	Das Speichern der Einträge	52
Code 4.9	Das Zurücklesen	54

F. Sachwortverzeichnis

A	
App	
veröffentlichen	38
B	
Besonderheiten	
technische, von Apps.....	6
von Apps	4
Breite	
proportionale	21
C	
Capability	7
Contract	7
D	
Datenverzeichnis	
lokales	51
Dialog	
einfacher	30
F	
Frame	34
L	
Lebenszyklus	7
M	
Microsoft Store	4
P	
PowerShell-Installationsskript	43
S	
Sandbox	6
Schlüsselwort	
async	30
await	30
Sideloadung	38
Simulator	16
Speichern	
der App-Daten	32
Splash Screen	14
Startbildschirm	14
Suspend-Modus	7
U	
Universal Windows Platform App	3
V	
Verarbeitung	
asynchrone	30
W	
Wiederherstellen	
der App-Daten	32
Windows App Certification Kit	40
Z	
Zeichentabelle	48
Zertifizierung	38

G. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:
CSPH24D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Erstellen Sie eine Universal Windows Platform App, in der ein Anwender einen beliebigen Text eingeben kann. Die Schriftgröße dieses Textes soll im Bereich 10 bis 40 verändert werden können. Für die Veränderung können Sie zum Beispiel zwei Schaltflächen verwenden. Die eine vergrößert den Text und die andere verkleinert ihn.

- Behandeln Sie in der App das erzwungene Beenden durch das System.
- Erstellen Sie für die Anwendung ein Paket für die lokale Installation und führen Sie eine Zertifizierung durch.
- Um Übertragungszeit und -kosten zu sparen, können Sie das Projekt mit einem geeigneten Programm packen – zum Beispiel mit WinZip oder direkt über Windows.
- Beschreiben Sie außerdem, welche grundsätzlichen Schritte für die Lösung erforderlich sind – also zum Beispiel, welche Steuerelemente Sie einfügen und wie Sie die Eigenschaften dieser Elemente setzen.

Für das Erstellen der App: 75 Pkt.

Für das Erstellen der Pakete und die Zertifizierung: 25 Pkt.

Gesamt: 100 Pkt.

