

Objektorientierte Programmierung

Objektorientierte Konzepte mit der Programmiersprache C#

Alexander Stuckenholtz

Version 2022-05-04

Inhaltsverzeichnis

Vorwort	1
0.1 Dozent und Kommunikation	1
0.2 Lernziele	2
0.3 Vorlesung und Übung	2
0.4 Materialien	3
0.5 Modul und Prüfung	4
1 Einführung	7
1.1 Programmierparadigmen	7
1.2 Objektorientierung	8
1.3 Klassen und Objekte	10
1.4 Kernprinzipien der Objektorientierung	12
1.5 Zusammenfassung und Aufgaben	13
2 C# für C-Programmierer	15
2.1 Warum eine neue Programmiersprache	15
2.2 C# und .Net	16
2.3 Variablen und Ausdrücke	21
2.4 Datentypen	22
2.5 Typumwandlung	23
2.6 Kontrollstrukturen	26
2.7 Zusammenfassung und Aufgaben	32
3 Klassen und Objekte	33
3.1 Objektvariablen und Zugriffsmodifizierer	37
3.2 Geheimnisprinzip und Eigenschaften	38
3.3 Getter, Setter und Eigenschaftsmethoden	39
3.4 Weitere Methoden	43
3.5 Zusammenfassung und Aufgaben	46
4 Objekte zur Laufzeit	49
4.1 Objekterzeugung	49
4.2 Objektzerstörung	53
4.3 Laufzeitfehler	55
4.4 Objektidentität	59
4.5 Zusammenfassung und Aufgaben	63

5	Schnittstellen und Aufzählungen	65
5.1	Klassenmethoden und -attribute	65
5.2	Arrays und Datenstrukturen	66
5.3	Auflistungen und ArrayList	68
5.4	Generische Klassen	69
5.5	Schnittstellen	72
5.6	Zusammenfassung und Aufgaben	76
6	Objektbeziehungen	79
6.1	Vereinsverwaltung	79
6.2	Objektbeziehungen	83
6.3	Tic Tac Toe	84
6.4	Zusammenfassung und Aufgaben	89
7	Vererbung	91
7.1	Redundanzen im Programmcode	91
7.2	Vererbung in C#	93
7.3	Vererbungshierarchie und Typumwandlung	96
7.4	Konstruktoren und Vererbung	97
7.5	Objektbeziehungen und Vererbung	99
7.6	Methoden und Vererbung	101
7.7	Double Dispatch Problem	104
7.8	Zusammenfassung und Aufgaben	105
8	Polymorphie	107
8.1	Wurzelklasse Object und das base-Schlüsselwort	107
8.2	Abstrakte Klassen	108
8.3	Polymorphie	111
8.4	Polymorphie mit abstrakter Basisklasse	112
8.5	Polymorphie mit Schnittstelle	115
8.6	Zusammenfassung und Aufgaben	116
9	Analyse	119
9.1	Softwareentwicklung und Softwaretechnik	119
9.2	Analyse und Modellierung mit der UML	121
9.3	Klassendiagramm	122
9.4	Assoziationen und Multiplizitäten	122
9.5	Aggregation und Komposition	127
9.6	Modellbildung	130
9.7	Zusammenfassung und Aufgaben	133
10	Analysemuster	137
10.1	Analysemuster	137
10.2	Historie, Quantität, Intervall und Koordinator	138

10.3	Reflexive Assoziation und Kompositum	141
10.4	Exemplartyp und Vorrat	143
10.5	Rollen und wechselnde Rollen	145
10.6	Zusammenfassung und Aufgaben	147
11	Entwurf	149
11.1	Analyse vs Entwurf	149
11.2	SOLID	151
11.3	Prinzip der eindeutigen Verantwortlichkeit	152
11.4	Prinzip der Offen- und Verslossenheit	155
11.5	Liskovsches Substitutionsprinzip	156
11.6	Schnittstellen Segregationsprinzip	158
11.7	Abhängigkeits-Umkehrprinzip	160
11.8	Zusammenfassung und Aufgaben	161
12	Entwurfsmuster	163
12.1	Entwurfsmuster	163
12.2	Fabrik und Einzelstück	163
12.3	Iterator	166
12.4	Strategie und Besucher	168
12.5	Beobachter	172
12.6	Zusammenfassung und Aufgaben	176
13	Zusammenfassung und Wiederholung	179
13.1	Grundsätze der Objektorientierten Programmierung	179
13.2	Überblick über die Veranstaltung	181
13.3	Prüfung und Credits	182
13.4	Abschluss	183
	Literatur	185

Vorwort

Willkommen

Willkommen!

- In diesem Kurs werden wir ein neues Programmierparadigma kennen lernen: Die Objektorientierte Programmierung.

Diese neue Art der Programmierung verändert vor allem die Art und Weise wie große Systeme strukturiert werden.

- Alles, was wir im ersten Semester über die *Imperative Programmierung* gelernt haben, benötigen wir aber weiterhin.

Bevor wir aber damit starten können, müssen wir einige organisatorische Dinge klären.

- Wir müssen z.B. über die **Lernziele** und **Zusatzinformationen** sprechen,
- welche **Kursmaterialien** Sie nutzen können und wie die **Prüfung** aussieht.

0.1 Dozent und Kommunikation

Dozent

Mein Name ist Alexander Stuckenholz.

- Ich bin der Dozent für diesen Kurs.

Seit 2012 bin ich Professor für praktische Informatik an der Hochschule Hamm-Lippstadt.

- Bevor ich Professor wurde, war ich in der Industrie tätig, hauptsächlich in der Energiewirtschaft.
- Ich habe in unterschiedlichen Rollen gearbeitet, als Programmierer, als Softwarearchitekt, als Projekt- und Teamleiter.

Einige Details zu meinem Werdegang finden Sie hier.

- Darüber hinaus bin ich auch auf Xing oder LinkedIn zu finden.



0.2 Lernziele

Lernziele

Ein erfolgreicher Teilnehmer an dieser Veranstaltung hat objektorientierte Programmierkonzepte verstanden und kann diese erfolgreich in der Programmiersprache C# anwenden.

Daraus ergeben sich dann einige Unterziele:

- Klassen in C# implementieren und sie mit geeigneten Eigenschaften und Methoden ausstatten können.
- Zur Laufzeit Objekte erzeugen und verwalten können.
- Objektbeziehungen aufbauen können.
- komplexe, objektorientierte Systeme so entwerfen können, dass sie verschiedenen Qualitätskriterien genüge tragen.

Achtung!

- Das sind grundlegende Fähigkeiten eines Programmierers!
- Ohne diese Fähigkeiten kommen Sie auch im Studium nicht weiter!

0.3 Vorlesung und Übung

Vorlesung und Übung

Diese Veranstaltung ist eine Pflichtveranstaltung im Studiengang ISD.

- Darüber hinaus können Studierende der ETR diese Veranstaltung im Rahmen der Vertiefungsrichtung *Energieinformatik* wählen.

Die Veranstaltung besteht aus einer **Vorlesung** (2 SWS) und einer **Übung** (1 SWS).

- In der Vorlesung werden neue Konzepte vermittelt.
- In der Übung müssen Sie diese Konzepte dann selber anwenden, um sich die Fähigkeiten auch anzueignen.

Orts- und Zeitangaben sind ihrem jeweiligen Stundenplan zu entnehmen.

- Bitte beachten Sie, dass jede Veranstaltungsstunde 45 Minuten umfasst.
- Entsprechend kann die Anfangszeit der Veranstaltung innerhalb des Slots variieren.
- Genauere Details werden mündlich in der Vorlesung bekannt gegeben.

0.4 Materialien

Kursmaterialien

In dieser Veranstaltung können Sie auf verschiedene Materialien zurückgreifen:

- Ein Script, in welchem jede Vorlesung ein Kapitel gewidmet ist,
- mehrere Lernvideos (mitunter schon älter, aber nicht minder relevant),
- Beispielcode: <https://github.com/LosWochos76/oop>.

Das Script und die Videos finden sich in der Lernplattform unter <https://my.hshl.de>.

- Haben Sie keinen Zugriff auf den Kurs dort, sind Sie (noch) nicht zur Prüfung angemeldet.

Bitte beachten Sie: Alle Materialien sind ausschließlich für Ihre individuelle Prüfungsvorbereitung gedacht.

- Die Weitergaben an Dritte ist nicht gestattet!

Literatur

Natürlich existiert eine schier unendliche Menge an guten Büchern und anderen Quellen zum Thema objektorientierte Programmierung.

- Auch in unserer Bibliothek finden sich entsprechende Werke, z.B.: [17], [15], [11], usw.

Im Verlauf der Veranstaltung wird in den Materialien auch immer wieder auf weiterführende Literatur verwiesen, z.B.: [7], [6], [9], usw.

- Dort geht es dann um Themen, die nicht mehr direkt zu den Lernzielen gehören.

Um aber die Lernziele dieser Veranstaltung zu erreichen, sollte Sie neben den eigentlichen Kursmaterialien keine weitere Literatur benötigen.

- Sollte dennoch etwas unklar bleiben, würde ich Sie bitten, Kontakt mit mir aufzunehmen.
- Dadurch kann ich die Veranstaltung verbessern.
- Und das kommt nicht nur Ihnen zugute, sondern auch den Generationen nach Ihnen.

0.5 Modul und Prüfung

Modul

Im Studiengang ISD besteht das Modul **Grundlagen der Informatik II** aus den folgenden Lehrveranstaltungen

- Objektorientierte Programmierung = 2 SWS Vorlesung + 1 SWS Übung
- Algorithmen und Datenstrukturen = 2 SWS Vorlesung + 2 SWS Übung

Im Studiengang ETR besteht das Modul **Studienschwerpunkt I: Energieinformatik** ebenfalls aus diesen Veranstaltungen.

- Die Übung zu Algorithmen und Datenstrukturen ist allerdings auf 1 SWS gekürzt.

In ISD werden für das Modul insgesamt 8 CPs vergeben, in ETR nur 6 CPs.

- Zur Erinnerung: Für 1 CP werden 30 Arbeitsstunden Aufwand veranschlagt.
- 8 CPs stehen also für 240h Arbeitsaufwand, 6 CPs für 180h.
- Nur die Vorlesung zu besuchen und die Übungsaufgaben zu bearbeiten reicht nicht!

Prüfung

Für beide Teile (OOP+AuD) wird eine gemeinsame, schriftliche Prüfung stattfinden.

- In der Regel ist die Prüfung 3h lang.
- Der gesamte Lehrstoff beider Veranstaltungen ist dabei prüfungsrelevant!
- In ETR wird die Klausur etwas weniger streng bewertet.

Als Hilfsmittel für die Klausur ist eine Formelsammlung (A4, doppelseitig) erlaubt.

- Diese muss selbst angefertigt sein, also nicht einfach vom Nachbarn kopiert.

Kommunikation

Ist während der Veranstaltung irgendetwas unklar, nehmen Sie bitte Kontakt mit mir auf.

- Am besten direkt in der Vorlesung bzw. Übung.
- Ansonsten gerne per E-Mail: `alexander.stuckenholz@hshl.de`
- Wir können aber auch eine Video-Konferenz über WebEx oder Skype ansetzen.

Slack ist ein anderer sehr guter Weg, um direkt miteinander in Kontakt zu kommen.

- Ich habe eine entsprechende Umgebung eingerichtet, die wir in dieser Veranstaltung nutzen können.
- Dort können wir schnell miteinander/untereinander chatten und auch Dateien einfach austauschen.
- Bitte lassen Sie es mich wissen, wenn Sie Slack nutzen wollen.

Hinweise zu gendergerechter Sprache

In allen Materialien dieses Kurses wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet.

- Auch andere Formen gendersensibler Sprache, wie z.B. Gendersternchen, Genderdoppelpunkt usw., kommen nicht zur Anwendung.

Sämtliche Personenbezeichnungen und personenbezogene Hauptwörter gelten gleichermaßen für alle Geschlechter.

- Die gewählte Sprachform hat nur redaktionelle Gründe.
- Sie stellt keinerlei Wertung dar.

1 Einführung

1.1 Programmierparadigmen

Programmierparadigma

Im letzten Semester haben wir gelernt, mit der Programmiersprache C umzugehen.

- Neben C existieren aber noch sehr viele andere Programmiersprachen.
- Manche ähneln C, andere fühlen sich doch ganz anders an.

Programmiersprachen werden in zwei fundamentale **Programmierparadigmen** eingeteilt.

- Die **imperativen** und die **deklarativen** Programmiersprachen.
- Diese Paradigmen legen fest, wie Probleme mithilfe der Sprachen gelöst werden können.

Dabei geht es darum, welche Programmelemente die jeweilige Programmiersprache anbietet und wie diese zusammenwirken.

- Statische Elemente: Variablen, Funktionen, Objekte, Module, ...
- Dynamische Elemente: Kontrollfluss, Datenfluss, ...

Imperative Sprachen

In der **imperativen Programmierung** wird mithilfe von Anweisungen genau dargelegt, wie ein Problem zu lösen ist.

- Die Programmiersprache C ist eine imperative Programmiersprache.

Imperative Programmiersprachen basieren auf einigen fundamentalen Bausteinen:

- **Deklarationen:** Einführung von Variablen, Funktionen etc.
- **Ausdrücke:** Ein Programmkonstrukt, welches zur Laufzeit ausgewertet wird und ein Ergebnis erzeugt.

1 Einführung

- **Datentypen:** Festlegung der Bedeutung eines Speicherinhalts, des erlaubten Wertebereichs und der Operationen.
- **Anweisungen:** Ein einzelner Befehl bzw. Instruktion an den Rechner.
- **Kontrollstrukturen:** Spezielle Anweisungen, welche die Reihenfolge der Abarbeitung der Anweisungen verändern.

Deklarative Sprachen

Deklarativen Programmiersprachen basieren auf einem ganz anderen Prinzip.

- Meist wird lediglich das Problem beschrieben, das was.
- Der Lösungsweg wird dann automatisch ermittelt.

Zu den deklarativen Programmiersprachen gehören z.B. SQL, Haskell, Erlang oder Prolog.

- Das folgende Beispiel in SQL gibt alle Windkraftanlagen im Umkreis von 10 km um Hamm aus.

```
1 select * from geo.renewable_power_plants where technology='Onshore' and
2   ST_distance(geom, ST_SetSRID(ST_Point(7.8398118, 51.6820189)::geography, 4326)) <= 10000
```

Dabei wird nur beschrieben, was gesucht wird.

- Nicht aber, wie diese Ergebnisse zu beschaffen sind.

1.2 Objektorientierung

Objektorientierung

Imperative und deklarative Sprachen basieren auf gänzlich unterschiedlichen Konzepten.

- Aber auch innerhalb der jeweiligen Paradigmen existieren viele unterschiedliche Strömungen.

Innerhalb der imperativen Sprachen ist z.B. die Idee der **Objektorientierung** entstanden.

- Die Objektorientierung hilft vor allem bei der Entwicklung großer und komplexer Systeme.

- Große Aufgaben können durch Objekte in handhabbare Teilprobleme zerlegt werden.
- Die Objektorientierung hilft daher dabei, die Erstellung von Software handhabbarer, wartbarer und testbarer zu machen.

Mit diesem Prinzip werden wir uns im Rest dieser Veranstaltung auseinandersetzen.

Abstrakte Datentypen

In der Programmiersprache C können **abstrakte Datentypen** deklarieren werden.

- Durch Zusammensetzen mehrerer bestehender Datentypen können so neue Datentypen geschaffen werden, z.B. ein Kreis mit einem Radius:

```
1 struct Kreis {  
2     double radius;  
3 };
```

Wir können nun beliebig viele Variablen von diesem Datentyp erzeugen: `Kreis k1 = { 4.5 };`

- Allein mit dieser Variable `k1` können wir aber wenig anfangen.

Wir benötigen Funktionen, um z.B. den Umfang des Kreises zu berechnen.

- In C werden Funktionen getrennt von solchen Datentypen deklariert:

```
1 double berechneUmfang(Kreis k) {  
2     return k.radius * 2 * M_PI ;  
3 }
```

Objektorientierte Programmierung

Daten und Funktionen bilden aber meist eine Einheit!

- Eine Variable vom Typ `Kreis` ist ohne passende Funktionen nutzlos.
- Diese Erkenntnis ist eine Grundidee der objektorientierten Programmierung.

Objekte verbinden Daten mit Funktionen.

- Im Gegensatz zu abstrakten Datentypen verbergen (schützen) Objekte ihre Daten.
- Öffentlich zugänglich sind nur Funktionen, die auf den Daten erlaubte Operationen zulassen.

1 Einführung

Mithilfe der objektorientierten Programmierung lässt sich auch gut die Realität abbilden.

- Es ist nicht schwer, in Objekten zu denken!
- Alles ist ein Objekt!

Eigenschaften von Objekten

Objekte haben drei wichtige Eigenschaften (siehe [16, S. 223])

Jedes Objekt hat eine Identität.

- Objekte sind voneinander unterscheidbar.
- Die Identität tragen Objekte von ihrer Entstehung bis zu ihrem Ende.

Jedes Objekt hat einen Zustand.

- Der Zustand eines Objekts ergibt sich durch seine Daten.
- Für diese Daten ist das Objekt selbst verantwortlich.

Jedes Objekt zeigt ein Verhalten.

- Das Verhalten des Objekts sind die Funktionen, die das Objekt zur Nutzung anbietet.
- Die Interaktion mit dem Objekt ist ausschließlich über diese Funktionen möglich.

1.3 Klassen und Objekte

Klasse

Das folgende Beispiel realisiert das Konzept eines Kreises in C# als sog. **Klasse**:

```
1  class Kreis
2  {
3      private int radius;
4
5      public Kreis(int radius)
6      {
7          this.radius = radius;
8      }
9
10     public double BerechneUmfang()
11     {
```



```

12     return radius * 2 * Math.PI;
13 }
14 }

```

Die Deklaration der Klasse verbindet die Daten (Radius) mit den Funktionen (Umfang).

Objekte

Aus so einer Klasse können nun beliebig viele Objekte erzeugt werden.

- Jedes Objekt besitzt dann seine eigenen, individuellen Daten.
- Einen initialen Wert für den Radius übergeben wir bei der Erzeugung der Objekte:

```

1 Kreis k1 = new Kreis(4.5);
2 Kreis k2 = new Kreis(9);

```

Die Funktionen sind nun integraler Bestandteil der Objekte.

- Sie können die ureigenen Daten des Objektes nutzen, um Ergebnisse zu berechnen.
- Die Funktionen werden auf dem Objekt aufgerufen.

```

1 double umfang1 = k1.BerechneUmfang();
2 double umfang2 = k2.BerechneUmfang();

```

Strukturelement Objekt

Die objektorientierte Programmierung verändert die Strukturierung der Programme.

- Dinge, die zusammen gehören, können nun auch zusammengebracht werden.
- Das Objekt ist ein neues Programmelement aus denen Systeme aufgebaut werden.

Objekte helfen dabei, komplexe Systeme zu konstruieren.

- Man kann sich zunächst überlegen, welche Objekte sinnvollerweise gebraucht werden.
- Danach kann man die Daten und Funktionen realisieren.
- Das hilft auch bei der Aufgabenverteilung im Team.

Die objektorientierte Programmierung setzt auf der imperativen Programmierung auf.

- Algorithmen werden noch immer auf dieselbe Art realisiert.

1 Einführung

- Nur eben als Teil von Objekten.

Objektorientierte Prinzipien

Objekte sind die Bausteine, aus denen objektorientierte Systeme bestehen.

- Die Objekte werden dabei auf Basis von vier Kernkonzepten entworfen:

Abstraktion

- Programmierer lieben Abstraktionen.
- Anstelle Programmcode zu duplizieren, werden z.B. Funktionen eingeführt.
- Mit Klassen und Schnittstellen existieren dazu ganz neue Möglichkeiten.

Datenkapselung

- Objekte verbergen die Details ihrer Implementierung und schützen ihre Daten.
- Über die Schnittstelle legt ein Objekt die sinnvolle Nutzung fest.

1.4 Kernprinzipien der Objektorientierung

Objektorientierte Prinzipien

Vererbung

- Vererbung erlaubt, neue Arten von Objekten aus bestehenden Definitionen abzuleiten.
- Durch die Bildung einer Vererbungshierarchie kann zusätzliche Abstraktion erzeugt werden.

Polymorphie

- Polymorphie meint, dass sich Objekte abhängig vom Kontext ihrer Verwendung unterschiedlich verhalten können.
- Sie sorgt u.a. dafür, dass objektorientierte Systeme flexibel erweitert werden können, ohne bestehenden Programmcode anpassen zu müssen.

Diese Konzepte zielen darauf ab, die Kohäsion zu erhöhen und die Koppelung zu reduzieren.

- Eine Voraussetzung, um evolvierbare Softwaresysteme herzustellen.
- Was das alles genau bedeutet, werden wir in dieser Veranstaltung lernen.

1.5 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, ...

- welche fundamental unterschiedlichen Programmierparadigmen existieren.
- aus welcher Idee die objektorientierte Programmierung entstanden ist.
- was Objekte von abstrakten Datentypen unterscheidet.
- was die vier Kernkonzepte der Objektorientierung sind.

Aufgaben

Erstellen Sie in der Programmiersprache C einen abstrakten Datentypen, der einen mathematischen Bruch (Zähler und Nenner) umsetzt.

- Realisieren Sie eine Funktion zur Multiplikation zweier Variablen des Typs.

Welche Werte darf der Nenner eines Bruchs nicht annehmen?

- Können Sie verhindern, dass Variablen mit solch ungültigen Werten überhaupt erzeugt werden?

2 C# für C-Programmierer

2.1 Warum eine neue Programmiersprache

Neue Programmiersprache

In dieser Veranstaltung wollen wir lernen, objektorientierte Konzepte einzusetzen.

- Dazu benötigen wir natürlich eine Programmiersprache, die dieses Konzept unterstützt.

Im letzten Semester haben wir die Programmiersprache C eingesetzt.

- C ist eine imperative Programmiersprache mit der man systemnah programmieren kann.
- C spielt insbesondere im Umfeld kleiner eingebetteter Systeme eine wichtige Rolle.
- C unterstützt allerdings keine objektorientierten Konzepte.

Für C existiert aber eine objektorientierte Erweiterung: C++

- Gerade für das Erlernen der Objektorientierung hat C++ aber gewisse Nachteile.
- Da C++ zu C kompatibel ist, wird eine Menge alter Ballast mitgeschleppt.
- Man kann in C++ auf viele Dinge Einfluss nehmen, das macht die Sache aber kompliziert.

Wir werden daher in diesem Semester zu einer anderen Programmiersprache wechseln:
C#

2 C# für C-Programmierer

Nicht ganz ernst zu nehmen

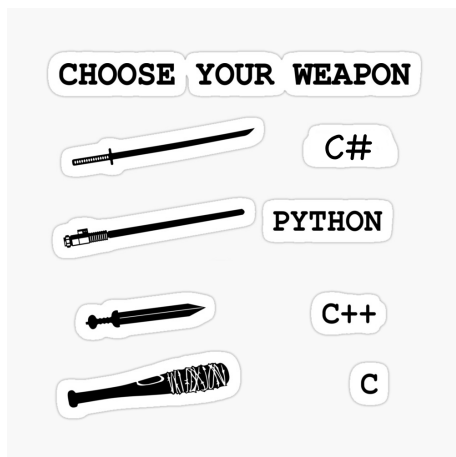
Zitat von Bjarne Strastrup, dem Entwickler von C++:

C macht es einfach, sich in den Fuß zu schießen; C++ erschwert es, aber wenn man es tut, bläst es einem das ganze Bein weg.



1

Nicht ganz ernst zu nehmen



2

2.2 C# und .Net

Grundlagen von C#

Mit C# wollen wir also eine neue Programmiersprache lernen.

¹Bildquelle: <https://commons.wikimedia.org/wiki/File:BjarneStroustrup.jpg>

²Bild angelehnt an: <https://rdbl.co/3rg0j8W>

- C# ist eine typsichere, imperative und objektorientierte Allzweck-Programmiersprache.
- Die Sprache inkludiert zudem funktionale, generische, parallele ...Konzepte.
- Die Sprache wurde im Auftrag von Microsoft von Anders Hejlsberg entwickelt (auch verantwortlich für TypeScript).

Die Syntax von C# ist zu C sehr ähnlich.

- Viele Anweisungen können wir genau so hinschreiben wie in C, z.B. `for`, `while`, `if`, ...
- Das `#` von C# kann man auch als vier mal `+` verstehen, also als C++++.

C# ist im Prinzip plattformunabhängig.

- Compiler setzen aber immer auf einer Laufzeitumgebung namens .Net auf.

Konzept von .Net

Ähnlich zu Java wird C# durch den Compiler nicht direkt in Maschinensprache übersetzt.

- Das Ergebnis ist ein **Bytecode**, die sog. Microsoft Intermediate Language (MSIL).

Zur Laufzeit wird der Bytecode dann in Maschinensprache übersetzt.

- Das übernimmt der sog. Just-in-Time-Compiler (JIT).
- Der JIT ist Teil der Common Language Runtime (CLR).

Dieses Prinzip ist nicht so performant, wie direkt Maschinencode auszuführen.

- Es bietet aber gewisse Vorteile, z.B. Speicherverwaltung, Sicherheitschecks, ...

Um Bytecode auszuführen, muss eine CLR für das Betriebssystem verfügbar sein.

- Früher existierten hier mehrere Alternativen: .Net Framework, .Net Core und Mono.
- Mit .Net 5 sollen alle diese Plattformen vereint werden.

Common Type System

Neben C# existieren weitere Sprachen, die Bytecode für .Net erzeugen können, z.B. F#.

- Alle Sprachen der .Net Plattform teilen sich ein gemeinsames System von Datentypen.
- Dieses wird als **Common Type System** (CTS) bezeichnet.

Daten können daher leicht zwischen Komponenten ausgetauscht werden.

- Selbst dann, wenn einzelne Teile in unterschiedlichen Sprachen realisiert wurden.
- Eine Klasse in C#, eine andere in F#, usw.

Das CTS unterscheidet zwischen **Werte-** und **Referenztypen**.

- Wertetypen liegen auf dem Stack, z.B. `int`, `double`.
- Wertetypen werden als Wert übergeben, es wird eine Kopie erzeugt.
- Referenztypen liegen grundsätzlich auf dem Heap.
- Hierzu gehören vorallem die Referenzen auf Objekte, die aus Klassen erzeugt werden.

Die .Net Klassenbibliothek

Neben dem CTS ist die .Net Klassenbibliothek ein wichtiger Teil der .Net Plattform.

- Für alle .Net Sprachen steht eine riesige Auswahl an Klassen zur Verfügung.
- Datentypen, Elemente für grafische Benutzeroberflächen, Netzwerkkommunikation, ...
- Für C#-Programmierer ist es enorm wichtig, einen Überblick über die unterschiedlichen Klassen zu bekommen.
- Die Online-Dokumentation ist frei zugänglich, siehe [19].

Die Klassen des Frameworks sind in sog. *Namespaces* (Namensräume) aufgeteilt.

- System.Collections → Klassen, die dynamische Datenstrukturen beinhalten
- System.Data → Klassen für den Zugriff auf Datenbanken
- System.IO → Ein-/Ausgabe in z.B. Dateien
- System.Web → Klassen für Web-Entwicklung

Installation und Entwicklungsumgebung

Um mit C# programmieren zu können, muss mindestens das .Net SDK installiert werden.

- Dies beinhaltet nicht nur die Laufzeitumgebung, sondern auch die Compiler etc.

Man kann das SDK als eigenständiges Paket herunterladen und installieren:

- Siehe: <https://dotnet.microsoft.com/download>.

Darüber hinaus wird auch noch eine Entwicklungsumgebung (IDE) benötigt, z.B.:

- Visual Studio für Windows (in der Community Version kostenlos)
- Visual Studio für Mac (kostenlos)
- Visual Studio Code (plattformunabhängig nutzbar und kostenlos)

Die ersten beiden IDEs können auch das SDK direkt mitinstallieren.

- Welche IDE genutzt wird, ist Geschmacksache.

.Net CLI

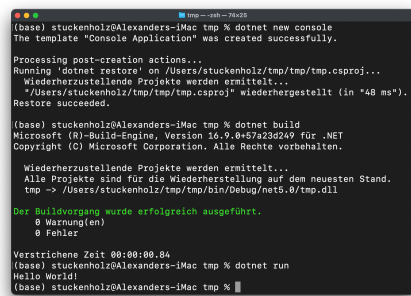
Auch in der .Net Welt wird gerne die Konsole (Command Line Interface, CLI) genutzt.

- Das .Net SDK stelle einen entsprechenden Befehl bereit: `dotnet`
- Damit können neue Programmierprojekte aus Vorlagen angelegt, übersetzt, getestet und ausgeliefert werden.

Einige Beispiele:

- Neues Konsolenprojekt erzeugen: `dotnet new console`
- Übersetzen des Projekts: `dotnet build`
- Projekt starten: `dotnet run`

2 C# für C-Programmierer



```
(base) stuckenholz@Alexanders-iMac tmp % dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running "dotnet restore" on /Users/stuckenholz/tmp/tmp.csproj...
Wiederherzustellende Projekte werden ermittelt...
"/Users/stuckenholz/tmp/tmp.csproj" wiederhergestellt (in "48 ms").
Restore succeeded.

(base) stuckenholz@Alexanders-iMac tmp % dotnet build
Microsoft (R) Build Engine, Version 16.9.0+77223249 für .NET
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Wiederherzustellende Projekte werden ermittelt...
Alle Projekte sind für die Wiederherstellung auf dem neuesten Stand.
tmp -> /Users/stuckenholz/tmp/tmp/bin/Debug/net5.0/tmp.dll

Der Buildvorgang wurde erfolgreich ausgeführt.
0 Warnung(en)
0 Fehler

Verstrichene Zeit 00:00:00.84
(base) stuckenholz@Alexanders-iMac tmp % dotnet run
Hello World!
(base) stuckenholz@Alexanders-iMac tmp %
```

Hello World

Wir erinnern uns an unser erstes Programm in C.

- Mit Hilfe von `printf` haben wir *Hello, World* auf der Konsole ausgegeben.

Das können wir natürlich auch mit C# machen.

- In unserer IDE oder mit Hilfe des CLI erzeugen wir ein neues Konsolenprojekt.
- Die einzige Quellcodedatei in dem neuen Projekt ist *Program.cs*:

```
1 namespace Test
2 {
3     class Program
4     {
5         static void Main(string[] args)
6         {
7             Console.WriteLine("Hello, World!");
8         }
9     }
10 }
```

Erklärung

Die Datei *Program.cs* beinhaltet die Hauptfunktion *Main* des Programms.

- Mit der *using* Anweisung werden zunächst einige Namensbereiche eingebunden.
- Dadurch müssen die Klassennamen darin nicht voll qualifiziert werden.

Das neue Programm liegt in einem eigenen Namensbereich, hier *Test*.

- Danach wird eine neue Klasse *Program* definiert.
- In C# müssen alle Funktionen Teil einer Klasse sein.

Es darf aber nur eine einzige Klasse geben, die eine statische Main-Methode besitzt.

- Diese dient als Einsprungspunkt in die Anwendung.
- In dieser Funktion wird nun der Text auf der Konsole ausgegeben.
- Dazu wird die Funktion *WriteLine* der Klasse *Console* genutzt.

2.3 Variablen und Ausdrücke

Variablen deklarieren

Variablen werden in C# genauso wie in C deklariert und auch benutzt.

- Variablen sind auch in C# sowohl *r-values* als auch *l-values*.
- Variablen können das Ziel einer Zuweisung sein, oder Bestandteil eines Ausdrucks.

Bei der Deklaration einer Variablen wird sein Datentyp (explizit) festgelegt:

```

1  int a = 42;
2  char b = 'a';
3  double c = 9.4;
4  bool d = false;

```

Die oben benutzten Datentypen sind allesamt Wertetypen.

- Variablen von diesen Typen werden auf dem Stack abgelegt.
- Bei Zuweisungen werden die Werte kopiert.

Schlüsselwort var

Die Programmiersprache C# ist statisch typisiert.

- Einer Variable liegt während ihrer Lebenszeit genau ein Datentyp zu Grunde.

Bei der Deklaration der Variablen wird dieser Datentyp festgelegt.

- Wir können diese Festlegung auch dem Compiler überlassen.
- Dazu wird das Schlüsselwort **var** genutzt.

```

1  var a = 42;
2  var b = 'a';
3  var c = 9.4;
4  var d = false;

```

2 C# für C-Programmierer

Die Variablen a, b, c und d bekommen wie zuvor ebenfalls einen Datentyp.

- Dieser wird aber durch die Zuweisung mit einem Wert automatisch erkannt.
- Ohne Zuweisung kann daher `var` nicht genutzt werden.

Operatoren und Ausdrücke

Auch Ausdrücke und Operatoren funktionieren exakt so, wie man es aus C bereits kennt.

- z.B. Zuweisungen und kombinierte Zuweisungen:

```
1 int a = 5;
2 a += 9;
3 a *= 10;
```

Die Operatoren für arithmetische Ausdrücke sind ebenfalls identisch:

```
1 int a = 17 + 9 * 42 / 3;
2 a++;
```

Auch die logischen Operatoren sind die selben:

```
1 bool wert = (42 != 5) && (13 > 9) || (!false);
```

2.4 Datentypen

Zeichen und Zeichenketten

Auch in C# existiert der Datentyp `char`.

- `char` ist ein Werttyp und repräsentiert mit Hilfe von 16 Bit ein einziges Unicode-Zeichen.
- Unicode kann auch Sonderzeichen darstellen.

```
1 char a = 'X';           // Ein einzelnes Zeichen
2 char d = '\u0058';      // Unicode
```

Der Typ `string` repräsentiert hingegen eine Zeichenkette hingegen und ist ein Referenztyp.

- Tatsächlich ist eine Variable vom Typ `string` ein Objekt der Klasse `System.String`.

- Bei der Zuweisung wird nur die Referenz kopiert, nicht der eigentliche Inhalt.

```
1 string s1 = "Peter";
2 string s2 = s1;      // Sowohl s1 als auch s2 verweisen nun auch den selben Speicherplatz
```

Intern wird beim `string` ein `char`-Array genutzt, um die Zeichen zu speichern.

- Daher ist ein `string` selbst auch unveränderlich ist (*immutable*).

Die Klasse Console

In C# müssen alle Funktionen Teil einer Klasse sein.

- Funktionen, die etwas mit der Konsole zu tun haben, finden sich in der Klasse `Console`.

Daten können als `string` eingelesen oder ausgegeben werden.

```
1 Console.WriteLine("Name: ");
2 string name = Console.ReadLine();
3 Console.WriteLine("Dein Name ist: " + name);
```

Aber die Klasse kann weitaus mehr, als `printf()` in C:

```
1 Console.BackgroundColor = ConsoleColor.White;
2 Console.ForegroundColor = ConsoleColor.Black;
3 Console.Clear();
4 Console.Beep();
```

2.5 Typumwandlung

Boxing und Unboxing

C# unterscheidet zwischen Werttypen und Referenztypen.

- Beide Typen können ineinander überführt werden.

Einen Werttypen in einen Referenztypen zu verpacken wird als **Boxing** bezeichnet.

- Boxing kann implizit durchgeführt werden.
- Man kann z.B. eine `int`-Variable in ein `object` umwandeln, die Wurzelklasse des Typsystems:

```
1 int i = 1;
2 object o = i; // boxing
```

Einen Referenztyp in einen Werttyp zu überführen wird als **Unboxing** bezeichnet.

- Unboxing ist immer explizit: `int j = (int)o;`
- Nicht alle solche Umwandlung können sinnvolle Ergebnisse produzieren.

Implizite Typumwandlung

Auch in C# entsteht mitunter die Notwendigkeit, den Wert einer Variablen in einen anderen Datentyp umzuwandeln.

- Umwandlungen ohne besondere Syntax wird als **implizite Typumwandlung** bezeichnet.
- Dies ist in C# nur dann erlaubt, wenn bei der Umwandlung kein Datenverlust entsteht.

Erlaubt:

```
1 int a = 5;
2 long b = a;
```

Nicht erlaubt (Compiler weigert sich zu übersetzen):

```
1 int c = 5.6;
```

Explizite Typumwandlung

Die Programmiersprache C# ist eine stark typisierte Sprache.

- Wenn bei der Umwandlung Informationen verloren gehen können, ist eine **explizite Typumwandlung** erforderlich (*engl. cast*).
- Dabei wird der Zieldatentyp in Klammern vor den umzuwandelnden Ausdruck geschrieben.

```
1 int c = 0;
2 c = (int)5.6;
```

Ist die Umwandlung nicht möglich, wird eine Ausnahme vom Typ `InvalidCastException` geworfen.

Mit Hilfe des `as`-Operators kann ebenfalls eine explizite Typumwandlung durchgeführt werden.

```
1 string s = obj as string;
```

Sollte die Umwandlung nicht möglich sein, wird dann als Ergebnis `null` zurückgegeben.

- Das hat den Vorteil, dass man leichter auf Fehler prüfen kann.

Convert

Oft ist eine Typumwandlung nicht trivial durchführbar.

- Der Kulturkreis hat z.B. Einfluss darauf, wie eine Dezimalzahl aus einem Text extrahiert werden kann (Bedeutung von Komma und Punkt).

Für solche Fälle stellt das .Net-Framework verschiedene Hilfsklassen bereit, z.B. `System.Convert`.

- Die `Convert`-Klasse bietet gängige Konvertierungsfunktionen an, um z.B. aus Texten unterschiedliche numerische Werte zu extrahieren.

```
1 string s = "1234";
2 int i = Convert.ToInt32(s);
3 s = "2016-11-01";
4 DateTime d = Convert.ToDateTime(s);
```

Oft wird auch die Umwandlung eines Objekts in einen `string` benötigt.

- Jedes Objekt bietet dazu die Methode `ToString()` an.
- Diese kann mit eigener Funktionalität überschrieben werden (später mehr).

Aufgabe

Schreiben Sie ein Programm, welches den Energieverbrauch eines Elektroautos in kWh pro 100 km und die Fahrstrecke einliest und den Gesamtverbrauch (evtl. die Kosten) errechnet und ausgibt.

Hinweise:

- Nutzen Sie `Console.WriteLine()` um Meldungen auszugeben.
- Nutzen Sie `Console.ReadLine()` um Daten als `string` einzulesen.

2 C# für C-Programmierer

- Um einen `string` in einen `double` zu konvertieren, kann `Convert.ToDouble()` genutzt werden.

Lösung

```
1 namespace Test
2 {
3     class Program
4     {
5         static void Main(string[] args)
6         {
7             Console.Write("Bitte geben Sie die Distanz in km ein: ");
8             string input = Console.ReadLine();
9             double distanz = Convert.ToDouble(input);
10
11             Console.Write("Bitte geben Sie den Verbrauch pro 100km ein: ");
12             input = Console.ReadLine();
13             double verbrauch_pro_100km = Convert.ToDouble(input);
14
15             double gesamtverbrauch = distanz / 100 * verbrauch_pro_100km;
16             Console.WriteLine("Sie haben " + gesamtverbrauch + " kWh verbraucht!");
17             Console.ReadLine();
18         }
19     }
20 }
```

2.6 Kontrollstrukturen

Verzweigungen

Auch die bekannten Kontrollstrukturen aus C sind in C# verfügbar:

Die `if`-Anweisung hat die selbe Funktionsweise wie in C:

```
1 if (alter > 18) { Console.WriteLine("Sie dürfen eintreten!"); }
2 else { Console.WriteLine("Sie dürfen NICHT eintreten!"); }
```

Natürlich können solche Konstrukte auch beliebig geschachtelt werden.

Darüber hinaus steht auch die `switch`-Anweisung zur Verfügung.

- Diese ist weit flexibler, als in C, da für den Vergleichsausdruck alles genutzt werden kann, dass nicht NULL ist.

```

1  switch (expr)
2  {
3      case 1: Console.WriteLine("Ist 1!"); break;
4      case 2: Console.WriteLine("Ist 2!"); break;
5      default: Console.WriteLine("Ist etwas anderes!"); break;
6  }

```

Schleifen

Auch Schleifen basieren in C# aus der selben Semantik, wie in C:

Eine Schleife mit `while`:

```

1  int u = 1000;
2  while (u > 0)
3  {
4      Console.WriteLine(u);
5      u--;
6  }

```

Eine Schleife mit `for`:

```

1  for (int i=0; i<100; i++)
2  {
3      Console.WriteLine(i);
4  }

```

Methoden

C# zwingt den Programmierer dazu, objektorientiert zu entwickeln.

- Daher müssen alle Funktionen als Bestandteil einer Klasse deklariert werden.
- Eine solche Funktion wird dann als **Methode** bezeichnet.
- Methodennamen werden per Konvention in C# immer groß geschrieben.

Methoden können so konstruiert sein, dass sie entweder auf einer Klasse oder auf einem Objekt aufgerufen werden können.

- Beim Aufruf wird der Name der Klasse/des Objekts um einem Punkt, dem Namen der Methode und den runden Klammern erweitert, z.B. `Console.WriteLine("Hello, World!");`

Methoden können beliebig viele Parameter erwarten.

- Zudem können sie maximal einen Wert an den Aufrufer zurückliefern.

2 C# für C-Programmierer

- Wenn eine Methode Parameter erwartet, müssen diese beim Aufruf auch übergeben werden.

Methoden deklarieren

Wir können eine zusätzliche Methode *Berechne* zu unserer Klasse hinzufügen.

- Die Methode wird mit dem Modifizierer `static` deklariert.
- Daher kann die Methode auf der Klasse aufgerufen werden.

```
1 class Program
2 {
3     static int Berechne(int wert)
4     {
5         return wert * 2;
6     }
7
8     static void Main(string[] args)
9     {
10         int ergebnis = Program.Berechne(42);
11     }
12 }
```

Beim Aufruf einer Methode der selben Klasse kann der Klassenname auch weggelassen werden.

Aufgabe

Schreiben Sie eine Methode, die prüft, ob eine übergebene Zahl eine Primzahl ist.

Lösung

Die Implementierung in C# ist fast identisch zu C.

```
1 static bool isPrime(int n)
2 {
3     if (n < 1 || n == 1)
4         return false;
5     else if (n == 2 || n == 3)
6         return true;
7
8     for (int i = 2; i <= n / 2; i++)
9         if (n % i == 0)
10             return false;
11 }
```

```

12     return true;
13 }

```

Felder

Felder (Arrays) fassen bekanntlich mehrere Variablen des gleichen Typs zusammen.

- Über einen ganzzahligen Index können einzelne Elemente gelesen und geschrieben werden.

Felder sind in der .Net-Welt Objekte vom Basistyp `Array` (Referenztyp).

- Objekte werden in C# immer erst deklariert und dann initialisiert:

```

1  int[] feld;           // Deklariert ein int-Feld:
2  feld = new int[5];    // Initialisiert das Feld für 5 int-Variablen

```

Dies lässt sich auch in einem Schritt schreiben, um z.B. ein zweidimensionales Feld anzulegen:

```

1  int[,] elements = new int[5,5];

```

Da solche Felder Objekte sind, besitzen sie auch Eigenschaften und Methoden:

```

1  int len = feld.Length; // Die Anzahl der Elemente im Array
2  int rank = elements.Rank; // Die Anzahl der Dimensionen im Array

```

Array-Klasse und foreach

Die Array-Klasse bietet eine Vielzahl von Methoden an, um mit Feldern zu arbeiten:

```

1  int[] feld = new int[5];
2  Arrays.Fill(feld, 7);           // Array mit 7 füllen
3  int index = Array.IndexOf(feld, 42); // Die Position der 42 im Array

```

Neben den bekannten `while` und `for`-Schleifen existiert in C# zusätzlich die `foreach`-Schleife.

- Die `foreach`-Schleife kann über alle Elemente einer Aufzählung iterieren.
- Dazu gehören Arrays und ansonsten alle Objekte, welche die Schnittstelle `IEnumerable` implementieren.

```
1 foreach (int i in feld)
2 {
3     Console.WriteLine("Element: " + i);
4 }
```

Aufgabe

Es soll schrittweise ein TicTacToe-Spiel entwickelt werden.

- Das Spielfeld können wir als 3x3-Matrix vom Datentyp `char` darstellen.

Am Anfang des Spiels sind alle Plätze mit dem Leerzeichen belegt.

- Spieler 1 wird im Array durch den Wert 'X' repräsentiert.
- Spieler 2 durch den Wert 'O'.

Schreiben Sie als erstes eine C#-Funktion, die das Spielfeld zufällig mit ' ', 'X' und 'O' initialisiert.

Hinweis:

- Zufallszahlen können mit Hilfe der Klasse `Random` erzeugt werden, siehe hier.

Lösung

```
1 static char[,] Zufallsfeld()
2 {
3     char[,] feld = new char[3, 3];
4     Random rnd = new Random();
5
6     for (int x=0; x<3; x++)
7     {
8         for (int y=0; y<3; y++)
9         {
10             switch (rnd.Next(0, 3))
11             {
12                 case 1: feld[x, y] = 'X'; break;
13                 case 2: feld[x, y] = 'O'; break;
14                 default: feld[x, y] = ' '; break;
15             }
16         }
17     }
18
19     return feld;
20 }
```

Aufgabe

Schreiben Sie als nächstes eine Methode, die das Spielfeld auf der Konsole ausgibt.

- Die Methode bekommt das Spielfeld als Parameter übergeben.

Lösung

```

1 static void Ausgeben(char[,] feld)
2 {
3     for (int i=0; i<3; i++)
4     {
5         for (int j=0; j<3; j++)
6         {
7             Console.Write("'" + feld[i,j] + "' ");
8         }
9
10        Console.WriteLine();
11    }
12 }
```

Aufgabe

Schreiben Sie eine weitere Methode, die prüft, ob ein bestimmter Spieler gewonnen hat.

- Das Spielfeld und der Spieler (X, O) sollen als Parameter übergeben werden.

Lösung

```

1 static bool Gewonnen(char[,] feld, char spieler)
2 {
3     for (int i = 0; i < 3; i++)
4     {
5         if (feld[0, i] == spieler && feld[1, i] == spieler && feld[2, i] == spieler)
6             return true;
7
8         if (feld[i, 0] == spieler && feld[i, 1] == spieler && feld[i, 2] == spieler)
9             return true;
10    }
11
12    if (feld[0, 0] == spieler && feld[1, 1] == spieler && feld[2, 2] == spieler)
13        return true;
14
15    if (feld[0, 2] == spieler && feld[1, 1] == spieler && feld[2, 0] == spieler)
16        return true;
17
18    return false;
19 }
```

2.7 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt...

- warum wir nicht C++, sondern C# benutzen wollen.
- welche Kerneigenschaften C# und die .Net Plattform besitzen.
- Wie das Hello-World-Programm in C# aussieht.
- wie man Variablen in C# deklariert.
- wie man arithmetische und logische Ausdrücke in C# formuliert.
- was der Unterschied zwischen Werte- und Referenztypen sind.
- welche zeichenbasierte Datentypen in C# existierten.
- wie man in C# mit der Konsole interagiert.
- wie man in C# Typumwandlung nutzt.
- wie in C# Felder deklariert und wie Methoden definiert werden.

Aufgaben

Erweitern Sie das TicTacToe-Spiel:

- Erstellen Sie eine Methode, die prüft, ob noch ein weitere Zug möglich ist.
- Erstellen Sie eine Methode, die vom Benutzer einen Zug einliest und dann ein 'X' setzt.

Lösen Sie die Aufgaben 1-4 aus dem Euler Projekt mit Hilfe von C#.

3 Klassen und Objekte

Objekte

Im letzten Abschnitt haben wir einige Grundlagen von C# kennen gelernt.

- C# ist eine objektorientierte Programmiersprache.
- Aber wirklich objektorientiert haben wir noch nicht programmiert.

Um objektorientiert zu programmieren, müssen wir lernen, mit Objekten umzugehen.

- Über Objekte müssen wir die folgende Dinge wissen:
 1. Objekte **verwalten und schützen** ihre eigenen Daten.
 2. Daten, die ein Objekt zur Nutzung anbietet, werden als **Eigenschaften** bezeichnet.
 3. Objekte bieten Dienstleistungen mit Hilfe ihrer **Methoden** an.

Diese Prinzipien werden wir uns heute näher ansehen.

Bruchrechnung

Wir wollen die Objektorientierung dazu nutzen, um die **Bruchrechnung** im Rechner abzubilden (siehe [12, S. 5-12])

- Jeder Bruch wird dann durch ein **Objekt** repräsentiert.

Jedes Bruchobjekt besitzt seinen eigenen Zähler und einen Nenner.

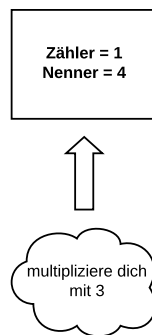
- Dies sind die sog. **Attribute** oder **Eigenschaften** des Objektes.
- Das Objekt *besitzt* diese Daten, es ist für sie verantwortlich.
- Es kann anderen Objekten den Zugriff erlauben oder ihn verbieten.

Über **Methoden** bietet das Objekt zudem weitere Dienstleistungen an.

- Man kann das Objekt z.B. nach seinem Zähler oder Nenner fragen oder den Kehrwert bilden.

3 Klassen und Objekte

- Solche Methoden können den inneren Zustand des Objekts verändern.



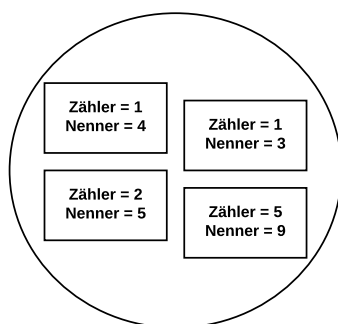
Klassen

Zur Laufzeit werden meist viele Objekte benötigt.

- Gemeinsam lösen sie dann die gewünschte Aufgabe.
- Manche Objekte sind strukturell gleich, manche unterscheiden sich sehr stark.

Objekte, mit gleiche Informationen als inneren Zustand und ähnlichen Nachrichten, können zu **Klassen** zusammengefasst werden.

- Alle Bruchobjekte gehören zur Klasse **Bruch**.
- Objekte, die zu einer Klasse gehören, werden als **Exemplar** oder **Instanz** dieser Klasse bezeichnet.



Klassen als Datentypen und Module

Eine Klasse realisiert einen **Datentyp**, wie `int` oder `double`.

- Eine Klasse legt fest, über welche Eigenschaften und Methoden ein Objekt verfügt.
- Genau das macht ein Datentyp: die Bedeutung und die Operationen auf Daten festlegen.

Eine Klasse ist auch eine Implementierungseinheit, ein **Modul**.

- Ein Modul ist ein Strukturelement für Softwaresysteme.
- Ein Modul dient dazu, Programmelemente (z.B. Funktionen) zu einer Einheit zusammenzufassen.
- Es ist eine wichtige Aufgabe des Systementwurfs, eine entsprechend sinnvolle Aufteilung zu wählen.

Klassen in C#

In C# wird eine Klasse benötigt, um zur Laufzeit Objekte zu erzeugen.

- Klassen dienen als Vorlage für Objekte.
- Sie legen fest, welche Daten und welche Operationen ein jedes Exemplar besitzen soll.
- Die Bruchklasse muss also festlegen, dass alle Brüche einen Zähler und einen Nenner besitzen und bestimmte Methoden anbieten.

In C# wird eine Klasse mit dem Schlüsselwort `class` eingeführt.

```
1 class Bruch {  
2 }
```

Eine neue Klasse wird in C# immer in einer eigenen Datei abgelegt.

- Diese Datei sollte den Namen der Klasse tragen, z.B. *Bruch.cs*.
- Klassennamen werden in C# immer groß geschrieben.

Objekte erzeugen

Um in C# ein Objekt aus einer Klasse zu erzeugen, sind zwei Schritte nötig:

1. Deklarieren einer Objektvariable, die auf ein Objekt verweisen kann (als sog. Referenz).
2. Erzeugen des Objektes mit Hilfe der Anweisung `new`.

```
1 Bruch b;           // Die Variable b ist vom Typ Bruch, zeigt aber noch nicht auf ein Objekt
2 b = new Bruch();   // Es wird ein neues Objekt erzeugt und der Variable b zugewiesen
```

Nach dem ersten Schritt verweist `b` auf nichts, auf `null`.

- Dass Objektreferenzen auch auf nichts verweisen können, ist eine häufige Fehlerquelle.
- Erst nach dem zweiten Schritt verweist `b` auf ein Objekt.

Eine solche Objekterzeugung kann aber auch in einem Schritt durchgeführt werden:

```
1 Bruch b = new Bruch();
```

Variablen

Bekanntlich werden Variablen dazu benutzt, um Daten zu speichern.

- Meistens nutzen wir die sog. **lokalen Variablen**.
- Deklarieren wir eine Variable z.B. innerhalb einer Funktion, existiert die Variable nur solange, wie die Funktion abgearbeitet wird.

Objektorientierte Programmiersprachen kennen aber noch zwei andere Arten von Variablen.

Die **Objektvariablen** (auch als Felder bezeichnet) gehören zum Objekt.

- Die Gültigkeit von Objektvariablen ist auf ein einzelnes Objekt beschränkt.
- Der Wert der Variablen kann daher für jedes Objekt unterschiedlich sein.

Die **Klassenvariablen** besitzen hingegen eine für die Klasse globale Gültigkeit.

- Klassenvariablen werden mit dem Schlüsselwort `static` deklariert.

3.1 Objektvariablen und Zugriffsmodifizierer

Objektvariablen

Die Objekte unserer Klasse *Bruch* besitzen bislang noch keinerlei Fähigkeiten.

- Entsprechend müssen wir die Klasse erweitern.

Jedes Bruchobjekt soll über einen eigenen Zähler und einen Nenner verfügen.

- Wir führen dazu zwei Objektvariablen *zaehler* und *nenner* ein.
- Die beiden Variablen werden innerhalb der Bruchklasse deklariert.

```

1 class Bruch
2 {
3     public int zaehler;
4     public int nenner;
5 }
```

Solche Objektvariablen realisieren den objekteigenen Speicher.

- Die Objektvariablen können in allen Methoden des Objekts genutzt werden.

Punktoperator

In der Main-Methode kann nun ein Objekt der Klasse erzeugt werden.

- Mit Hilfe des **Punktoperators** kann auf Elemente des Objekts zugegriffen werden.
- Links vom Punktoperator muss eine gültige Objektreferenz stehen.
- Rechts vom Punktoperator folgt dann der Name einer Objektvariablen oder einer Methode.

```

1 Bruch b = new Bruch();
2 b.zaehler = 1;
3 Console.WriteLine(b.zaehler);
```

So, wie wir die Klasse *Bruch* definiert haben, gibt es aktuell kaum einen Unterschied zu den abstrakten Datentypen in C.

- Lediglich das Schlüsselwort `public` vor der Deklaration von Zähler und Nenner ist neu.

Zugriffsmodifizierer

Das Schlüsselwort `public` ist ein sog. **Zugriffsmodifizierer**.

- Ein Zugriffsmodifizierer steuert die Sichtbarkeit von Elementen.

Objektvariablen, die `public` deklariert wurden, können außerhalb des Objekts genutzt werden.

- Private Elemente (`private`) sind nur durch das Objekt selbst nutzbar.
- Der Zugriffsmodifizierer `protected` wird im Kontext von Vererbung benutzt (später mehr).
- Ohne weitere Angabe ist ein Element immer privat.

Wir haben Zähler und Nenner `public` deklariert.

- Dadurch können wir z.B. aus der Main-Methode heraus auf die Objektvariablen eines Bruch-Objekts zugreifen.
- Diese Objektvariablen können daher auch als **Eigenschaften** aufgefasst werden.

3.2 Geheimnisprinzip und Eigenschaften

Geheimnisprinzip

Öffentliche Objektvariablen sind problematisch.

- Wird bei einem Bruchobjekt der Nenner auf 0 gesetzt, entsteht ein fehlerhafter Zustand.

Ein Objekt ist aber für seine Daten verantwortlich.

- Es muss es sich vor solchen fehlerhaften Daten schützen.

Objekte können sich aber mit Hilfe von Methoden vor fehlerhaften Daten schützen.

- Der Zugriff auf Objektvariablen darf dann nur über entsprechende Methoden geschehen.
- Bevor Objektvariablen verändert werden, können die übergebenen Daten geprüft werden.
- Die Methoden dienen dann als Schutzschicht vor den Daten des Objekts.

Diese Systematik wird als **Geheimnisprinzip** oder **Datenkapselung** bezeichnet.

- Dies ist eines der wichtigsten Prinzipien der Objektorientierten Programmierung (siehe auch [17, S. 31f]).

3.3 Getter, Setter und Eigenschaftsmethoden

Getter und Setter

Wir können nun unsere Bruchklasse entsprechend umbauen.

- Zunächst ändern wir den Zugriffsmodifizierer der beiden Objektvariablen auf `private`.
- Dadurch können wir nicht mehr von außerhalb des Objekts diese Objektvariablen lesen oder schreiben.

Danach führen wir vier neue Methoden ein:

- Die Methoden `GetZaehler()` und `GetNenner()` dienen dazu, den aktuellen Wert der Objektvariablen an den Aufrufer zurück zu liefern.
- Die Methoden `SetZaehler()` und `SetNenner()` können für das Schreiben der Daten genutzt werden.

Solche Methoden nennt man auch **Getter** bzw. **Setter**.

- Durch die Existenz von Getter und/oder Setter werden Daten von außerhalb zugreifbar.
- Dadurch besitzt das Objekt erneut implizite *Eigenschaften* (engl. *properties*).

Bruch mit Eigenschaften

```
1 class Bruch {
2     private int zaehler;
3     private int nenner;
4
5     public int GetZaehler() {
6         return zaehler;
7     }
8
9     public void SetZaehler(int wert) {
10        zaehler = wert;
11    }
12
13    public int GetNenner() {
14        return nenner;
15    }
16}
```

3 Klassen und Objekte

```
17     public void SetNenner(int wert) {
18         if (wert != 0)
19             nenner = wert;
20     }
21 }
```

Die Objektvariablen `zaehler` und `nenner` sind in allen Methoden des Objekts verfügbar.

- Die Objektvariablen sind also innerhalb des Objekts global gültig.

Die Getter- und Setter-Methoden sind alle mit dem Zugriffsmodifizierer `public` deklariert.

- Dadurch können die Methoden von außerhalb des Objekts genutzt werden.

Die Setter-Methoden können übergebene Werte prüfen.

- In `SetNenner()` wird verhindert, dass der Nenner auf 0 gesetzt wird.

Objekte nutzen

Ein so erzeugtes Objekt kann über die Getter- und Setter-Methoden benutzt werden.

- Diese bieten die Möglichkeit, Zähler und Nenner zu lesen und zu schreiben.

```
1 Bruch b = new Bruch();
2 b.SetZaehler(1);
3 b.SetNenner(3);
4 Console.WriteLine(b.GetZaehler() + "/" + b.GetNenner());
```

Wir können nun beliebig viele Objekte aus der Klasse erzeugen.

- Jedes Objekt besitzt dann individuelle Werte für Zähler und Nenner.

Objekte als Parameter

Solche Objekte können natürlich auch als Parameter an Methoden übergeben werden.

- Dabei wird die Referenz auf das Objekt in den Parameter kopiert.
- Es wird keine Kopie des Objekts erzeugt.
- Änderungen schlagen daher auf das ursprüngliche Objekt durch.

```
1 public static void VeraendereDenBruch(Bruch b)
2 {
3     b.SetNenner(5);
4 }
5
6 public static void Main()
7 {
8     Bruch b = new Bruch();
9     b.SetZaehler(1);
10    b.SetNenner(3);
11    VeraendereDenBruch(b); // Danach ist der Nenner von b auch hier gleich 5!
12 }
```

Eigenschaftsmethoden

Durch die Definition der Methoden *GetZaehler()* und *SetZaehler()* haben wir implizit dafür gesorgt, dass jedes Bruchobjekt eine Eigenschaft *Zaehler* besitzt.

- In vielen objektorientierten Programmiersprachen wird dies so gehandhabt, z.B. in Java.

Mit **Eigenschaftsmethoden** können in C# Eigenschaften auch explizit deklariert werden.

- Eine solche Eigenschaftsmethode kann (muss nicht) zwei Teile umfassen.

Der **get**-Teil gewährt lesenden, der **set**-Teil schreibenden Zugriff auf die Eigenschaft.

- Wird kein **get**- oder **set**-Teil realisiert, ist die Eigenschaft entweder nur les- oder nur schreibbar.

Eigenschaftsmethoden machen Eigenschaften in C# explizit.

- Sie sollten daher immer den Getter- und Setter-Methoden vorgezogen werden.

Nenner als Eigenschaftsmethode

Im Folgenden wird der Nenner als Eigenschaftsmethode eingeführt:

```
1 public int Nenner
2 {
3     get
4     {
5         return nenner;
6     }
7     set
8     {
```

3 Klassen und Objekte

```
9         if (value != 0)
10             nenner = value;
11     }
12 }
```

Innerhalb des **set**-Teils ist die Variable **value** vordefiniert.

- Sie trägt den neuen Wert, der von außen an die Eigenschaft herangetragen werden soll.

Hier können erneut Prüfungen vorgenommen werden.

- Mit Hilfe von Ausnahmen (*Exceptions*) können hier auch Fehler aufgezeigt werden.

Eigenschaftsmethoden benutzen

Eigenschaftsmethoden fühlen sich in ihrer Benutzung wie Objektvariablen an.

- Durch eine Zuweisung wird unter der Haube automatisch der **set**-Teil der Eigenschaftsmethode aufgerufen.

```
1 Bruch b = new Bruch();
2 b.Zaehler = 1;
3 b.Nenner = 3;
```

Entsprechend einfach ist auch der lesende Zugriff.

- Die Eigenschaftsmethode muss lediglich als R-Value in einem Ausdruck benutzt werden:

```
1 Console.WriteLine(b.Zaehler + "/" + b.Nenner);
```

Abgeleitete Eigenschaften

Eigenschaftsmethoden müssen nicht unbedingt auf den Objektvariablen des Objekts basieren.

- Es können auch sog. **abgeleitete Eigenschaften** realisiert werden.

Der Wert einer solchen Eigenschaften wird erst zur Laufzeit berechnet.

- Abgeleitete Eigenschaften können entsprechend nur gelesen werden.

Ein Bruchobjekt könnte man z.B. nach seiner Dezimalzahl fragen.

- Die Klasse Bruch kann um eine weitere Eigenschaftsmethode erweitert werden:

```

1 public double Dezimalzahl
2 {
3     get
4     {
5         return (double) zaehler / nenner;
6     }
7 }

```

3.4 Weitere Methoden

Weitere Methoden

Jedes unserer Bruchobjekte besitzt nun einen Zähler und einen Nenner.

- Mit Hilfe von Methoden können wir auf diese Attribute regelkonform zugreifen.

Unser Ziel war es, die Bruchrechnung im Rechner abzubilden.

- Dafür müssen wir insbesondere dafür sorgen, dass wir mit Brüchen auch rechnen können.
- Dies machen wir, indem wir neue Methoden in die Klasse einbauen.
- Methoden können auf alle Objektvariablen des Objekts zugreifen, auch auf private.

Als erstes fügen wir eine neue Methode *Ausgeben()* hinzu.

- Diese dient dazu, ein einzelnes Bruchobjekt auf der Konsole auszugeben.

```

1 public void Ausgeben()
2 {
3     Console.WriteLine(zaehler + "/" + nenner);
4 }

```

Kehrwert

Die Methode *Ausgeben()* verändert die Objektvariablen des Objekts nicht.

- Dies ist aber ohne weiters möglich.

Wir können z.B. eine Methode realisieren, die den Kehrwert eines Bruchs bildet.

- Dafür wird einfach der Wert von Zähler und Nenner miteinander getauscht:

3 Klassen und Objekte

```
1 public void BildeKehrwert()
2 {
3     int tmp = nenner;
4     nenner = zaehler;
5     zaehler = tmp;
6 }
```

Ruft man diese Methode auf einem Objekt auf, ändert sich sein innerer Zustand.

Methoden benutzen

Wir können nun wie zuvor beliebig viele Bruchobjekte erzeugen:

```
1 Bruch b = new Bruch();
2 b.Zaehler = 1;
3 b.Nenner = 3;
```

Die zusätzlichen Methoden können auf allen Objekten aufgerufen werden:

```
1 b.Ausgeben();           // gibt "1/3" auf der Konsole aus
2 b.BildeKehrwert();
3 b.Ausgeben();           // gibt "3/1" auf der Konsole aus
```

Methoden mit Rückgaben

Methoden sind bekanntlich Funktionen auf Objekten.

- Sie können beliebig viele Parameter erwarten und maximal einen Wert an den Aufrufer zurück liefern.

Die Methode *BildeKehrwert()* könnten wir so realisieren, dass sie das Ergebnis an den Aufrufer zurück liefert.

- Dazu erzeugen wir ein neues Ergebnisobjekt, welches den Kehrwert abbildet.

```
1 public Bruch BildKehrwert()
2 {
3     Bruch ergebnis = new Bruch();
4     ergebnis.Zaehler = nenner;
5     ergebnis.Nenner = zaehler;
6     return ergebnis;
7 }
```

Der Zähler des Ergebnisobjekts ist der Nenner des Ursprungsobjekts (und anders herum).

Methoden mit Parametern

Mit Hilfe von Methoden können wir auch die Multiplikation mit einer ganzen Zahl umsetzen:

```

1 public Bruch Multipliziere(int zahl)
2 {
3     Bruch ergebnis = new Bruch();
4     ergebnis.Zaehler = zaehler * zahl;
5     ergebnis.Nenner = nenner;
6     return ergebnis;
7 }

```

Genauso lässt sich auch die Multiplikation mit einem anderen Bruch umsetzen:

```

1 public Bruch Multipliziere(Bruch b)
2 {
3     Bruch ergebnis = new Bruch();
4     ergebnis.Zaehler = zaehler * b.Zaehler;
5     ergebnis.Nenner = nenner * b.Nenner;
6     return ergebnis;
7 }

```

Methoden überladen

Unsere erweiterte Bruchklasse besitzt nun zwei Methoden mit dem selben Namen.

- Eine Methode **Multipliziere()** für Ganzzahlen.
- Eine Methode **Multipliziere()** für Brüche.

Obwohl es zwei gleichnamige Methoden in der selben Klasse gibt, übersetzt der Compiler das Projekt erfolgreich.

- Solange sich die Methoden anhand von Anzahl oder Datentyp der Parameter unterscheiden lassen, gibt es kein Problem.
- Während der Übersetzung (nicht zur Laufzeit) entscheidet der Compiler, welche der Varianten besser passt.

Diese Technik wird als **Methodenüberladung** bezeichnet.

Methoden benutzen

Die zusätzlichen Methoden können nun auch wieder auf allen Bruchobjekten benutzt werden.

- Als Ergebnis erhalten wir aber neue Objekte.
- Das Ursprungsobjekt ändert sich nicht.

```
1 Bruch kehrwert = b.BildeKehrwert();
2 kehrwert.Ausgeben(); // Gibt "3/1" auf der Konsole aus
3 b.Ausgeben()         // Gibt "1/3" auf der Konsole aus
```

Bei der Methode *Multipliziere()* haben wir das selbe Prinzip angewandt:

```
1 Bruch ergebnis = b.Multipliziere(5);
2 ergebnis.Ausgeben(); // Gibt "5/3" auf der Konsole aus
```

Wenn wir wollen, können die Methodenaufrufe sogar aneinander hängen:

```
1 kehrwert.Multipliziere(b).Ausgeben();
```

3.5 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, ...

- was Objekte sind und wie daraus Klassen entstehen.
- wie man Objekte und Klassen in der UML grafisch darstellt.
- wie man Klassen in C# definiert.
- wie man in C# Exemplare aus einer Klasse erzeugt.
- was Eigenschaften von Objekten sind.
- warum öffentliche Objektvariablen nicht gut sind.
- wie man Getter- und Setter-Methoden einführt, um das Geheimnisprinzip zu wahren.
- dass man in C# anstelle von Getter- und Setter-Methoden auch Eigenschaftsmethoden nutzen kann.
- wie man abgeleitete Eigenschaftsmethoden definieren kann.

- wie man weitere Methoden realisiert.

Aufgabe 1

Erweitern Sie die Klasse `Bruch`.

- Ein `Bruch` soll durch eine Ganzzahl und einen anderen `Bruch` dividierbar sein.
- Als Ergebnis soll jeweils ein neues `Bruch`objekt zurückgegeben werden.
- Nutzen Sie (wenn möglich) bereits bestehende Methoden.

Aufgabe 2

Definieren Sie eine Klasse **`Rechteck`** in `C#`.

- Fügen Sie der Klasse zwei private Objektvariablen *seite1* und *seite2* hinzu.
- Erlauben Sie mit Hilfe von Getter- und Setter-Methoden den Zugriff.
- Verhindern Sie, dass die Seitenlängen unsinnige Werte annehmen können.
- Stellen Sie die Getter- und Setter-Methoden auf Eigenschaftsmethoden um.
- Erzeugen Sie mehrere Objekte aus der Klasse mit individuellen Seitenlängen.
- Fügen Sie der Klasse zwei abgeleitete Eigenschaften hinzu, die Fläche und Umfang zurückliefern.

4 Objekte zur Laufzeit

4.1 Objekterzeugung

Uninitialisierte Objekte

Mithilfe der Klasse Bruch haben wir einen neuen Datentyp eingeführt.

- Wir haben damit begonnen, dem Rechner die Bruchrechnung beizubringen.

Was gibt aber das folgende Programm aus?

```
1 Bruch b = new Bruch();  
2 b.Ausgeben();
```

Es wird 0/0 ausgegeben!

- Da hier der Nenner 0 ist, darf dieses Objekt eigentlich gar nicht existieren dürfen!

Wir müssen dafür sorgen, dass die Objektvariablen des Objekts direkt beim Erstellen mit sinnvollen Werten belegt werden können.

- Dazu können wir sog. **Konstruktoren** einführen.

Konstruktor definieren

Ein Konstruktor ist eine spezielle Methode eines Objektes.

- Er besitzt den Namen der Klasse (in der exakt gleichen Schreibweise).
- Er definiert keine Rückgabe (auch nicht `void`).

Ein Konstruktor kann (muss aber nicht) Parameter erwarten.

- Diese können dann dazu genutzt werden, um das neue Objekt zu initialisieren.
- Entsprechend können wir unsere Bruchklasse um einen Konstruktor erweitern:

4 Objekte zur Laufzeit

```
1 public Bruch(int z, int n)
2 {
3     Zaehler = z;
4     Nenner = n;
5 }
```

Konstruktor benutzen

Der Konstruktor wird nicht direkt aufgerufen.

- Der Konstruktor wird automatisch aufgerufen, sobald mit `new` ein Objekt erzeugt wird.

```
1 Bruch b = new Bruch(1,3); // Die Werte 1 und 3 werden als Parameter an den Konstruktor übergeben
2 b.Ausgeben();           // Gibt "1/3" auf der Konsole aus
```

Durch die Existenz des Konstruktors kann nun kein Objekt mehr erzeugt werden, ohne die beiden Parameter für Zähler und Nenner zu übergeben.

- Der Nutzer der Bruchklasse wird zur korrekten Nutzung gezwungen.

In einigen Methoden unsere Klasse erzeugen wir Ergebnisobjekte.

- Überall dort müssen wir dann ebenfalls den Programmcode anpassen.
- Sonst wird der Compiler den Programmcode nicht mehr übersetzen wollen.

Anpassungen

Durch die Änderungen wird der Programmcode teilweise sogar kompakter.

- Die geänderte Methode *BildeKehrwert()* sieht dann z.B. wie folgt aus:

```
1 public Bruch KehrwertBilden()
2 {
3     return new Bruch(nenner, zaehler);
4 }
```

Auch die Multiplikation mit einer Ganzzahl sieht viel schlanker aus:

```
1 public Bruch Multipliziere(int zahl)
2 {
3     return new Bruch(zaehler * zahl, nenner);
4 }
```

Objektinitialisierer

Auch ohne Konstruktor können Objekte bei der Erzeugung mit Werten initialisiert werden.

- Dazu kann der sog. **Objektinitialisierer** genutzt werden, der immer zur Verfügung steht.
- Im Gegensatz zum Konstruktor, muss der Objektinitialisierer nicht erst definiert werden.

Ohne Konstruktor haben wir ein *Bruch*-Objekt wie folgt erzeugt:

```
1 var b = new Bruch();
2 b.Zaehler = 1;
3 b.Nenner = 3;
```

Diese Anweisungen können mithilfe des Objektinitialisierers in einer einzigen Anweisung zusammen gefasst werden:

```
1 var b = new Bruch() { Zaehler=1, Nenner=3 };
```

In der geschweiften Klammer können (müssen aber nicht) alle öffentlichen Objektvariablen oder Eigenschaftsmethoden mit Werten initialisiert werden.

Unterschied zwischen Konstruktor und Objektinitialisierer

Der Objektinitialisierer dient letztendlich eher der Bequemlichkeit.

- Er kann, muss aber nicht für die Initialisierung genutzt werden.

Ein Konstruktor hingegen muss genutzt werden.

- Er stellt die korrekte Initialisierung eines Objekts sicher.
- Da unsere Bruch-Klasse einen Konstruktor definiert, müssen wir dort auch Werte für Zähler und Nenner übergeben.

Beide Arten der Initialisierung können allerdings miteinander gemischt werden.

- Nehmen wir an, die Bruch-Klasse würde eine weitere Eigenschaft definieren, die nicht durch den Konstruktor initialisiert werden muss.
- Diese könnten wir nun zusätzlich beim Erzeugen initialisieren: `var b = new Bruch(1, 3) { WeitereEigenschaft =`

Namenskonflikte

Parameter sind ein Beispiel für lokale Variablen der Methode.

- Sie liegen auf dem Stack und sind nur innerhalb der Methode gültig.
- Lokale Variablen werden nach dem Verlassen der Methode automatisch zerstört.
- Im Gegensatz dazu sind Objektvariablen in allen Methoden des Objekts gültig.

Objektvariablen und lokale Variablen können gleiche Namen besitzen:

```
1 class Test
2 {
3     private int z;
4
5     public TueEtwas(int z)
6     {
7         // Es gibt nun zwei z, ein Objektvariable und eine lokale Variable in der Methode
8     }
9 }
```

Obwohl hier ein Namenskonflikt vorliegt, ist das Programm korrekt übersetzbar.

Schlüsselwort `this`

Solche Namenskonflikte zwischen Objektvariablen und lokalen Variablen lassen sich aber auflösen.

- Innerhalb von Methoden existiert eine besondere Variable: `this`
- Die Variable `this` ist eine Referenz, die auf das eigene Objekt verweist.
- Mit Hilfe dieser Variable kann man im Falle von Namenskonflikten wieder auf die Objektvariable des Objektes zugreifen.

```
1 class Test
2 {
3     private int z;
4
5     public TueEtwas(int z)
6     {
7         z = 5;           // Die lokale Variable erhält den Wert 5
8         this.z = 10;     // Die Objektvariable des Objekts erhält den Wert 10
9     }
10 }
```

4.2 Objektzerstörung

Destruktor

Der Konstruktor dient am Anfang der Lebenszeit eines Objektes dazu, Objektvariablen zu initialisieren.

- Zum Konstruktor existiert auch ein Gegenstück: der sog. **Destruktor**.

Auch der Destruktor ist eine spezielle Methode des Objektes.

- Er kann dazu genutzt werden, Ressourcen wieder freigegeben, z.B. Dateihandles oder Datenbankverbindungen.

Auch der Destruktor wird nicht direkt aufgerufen.

- Beschließt der **Garbage Collector**, ein Objekt zu löschen, wird der Destruktor aufgerufen.

Folgende Regeln gelten für den Destruktor:

- Er trägt den Namen der Klasse mit einer führenden Tilde, z.B. *~Bruch()*.
- Er hat keinen Zugriffsmodifizierer keinen Rückgabewert und keine Parameter.
- Es kann nur einen Destruktor geben, er kann also nicht überladen werden.

Klasse Bruch mit Destruktor

```

1  class Bruch
2  {
3      private int zaehler;
4      private int nenner;
5
6      // Ein Konstruktor:
7      public Bruch(int zaehler, int nenner)
8      {
9          this.zaehler = zaehler;
10         this.nenner = nenner;
11     }
12
13     // Der Destruktor:
14     ~Bruch()
15     {
16         Console.WriteLine("Ein Bruch-Objekt wird nun zerstört!");
17     }
18 }
```

Nachteile des Destruktors

Auf der .Net Plattform ist der Garbage Collector dafür zuständig, Objekte zu löschen.

- Existiert keine Referenz mehr auf ein Objekt, kann dieses freigegeben werden.
- Wird der Speicher freigegeben, wird auch der Destruktor aufgerufen.

Man kann aber eine Aussage darüber treffen, wann der Garbage Collector diese Arbeit erledigt.

- Die Zerstörung kann mitunter sehr lange auf sich warten lassen.
- Entsprechend kann es ein, dass Ressourcen erst sehr spät wieder freigegeben werden.

Ressourcen sollten aber so zeitnah wie möglich freigegeben werden.

- Daher ist die Nutzung eines Destruktors in C# (im Gegensatz zu C++) zwar möglich, aber nicht sinnvoll.

Dispose

In .Net wird ein anderer Mechanismus genutzt, um die Ressourcenfreigabe zu beeinflussen.

- Um Ressourcen freizugeben, wird die Methode *Dispose()* genutzt.
- Dazu muss die Schnittstelle `IDisposable` implementiert werden.
- Klassen, die diese Schnittstelle implementieren, haben die Fähigkeit, sich selbst aufzuräumen.

```
1 class Bruch : IDisposable
2 {
3     // Der Ganze Rest fehlt
4
5     public void Dispose()
6     {
7         Console.WriteLine("Hier wird nun aufgeräumt!");
8     }
9 }
```

using-Anweisung

Auf einem Objekt kann die Methode *Dispose()* manuell aufgerufen werden.

- Es gibt aber einen besseren Weg, um die Freigabe von Ressourcen auch im Fehlerfall sicherzustellen.

Die **using**-Anweisung hilft dabei, dass Ressourcen wieder freigegeben werden.

- Sie kann auf alle Objekte angewandt werden, welche die Schnittstelle `IDisposable` implementieren.

```

1 using (Bruch b = new Bruch(1,3))
2 {
3     b.Ausgeben();
4 }

```

Wird der using-Block verlassen, wird auf dem Objekt `b` die Methode *Dispose()* aufgerufen.

- Dies wird auch im Fehlerfall sicher gestellt.

4.3 Laufzeitfehler

Ausnahmen

Auch mit der Einführung des Konstruktors haben wir leider noch immer ein Problem.

- Der Konstruktor nutzt die Eigenschaftsmethode, um den Wert des Nenners zu initialisieren.
- Zwar wird der Wert 0 abgelehnt, der Bruch erhält dann aber keine Initialisierung.
- Es entsteht also noch immer ein fehlerhaftes Bruchobjekt.

Wir müssen final dafür sorgen, dass solche Brüche nicht entstehen können.

- Zur Not müssen wir das Programm mit einem Fehler abbrechen.
- Dies ist besser, als mit fehlerhaften Daten weiterzuarbeiten.

Die Programmiersprache C# stellt dafür die sog. **Ausnahmen** (engl. *Exceptions*) bereit.

- Ausnahmen signalisieren dem Aufrufer einer Methode einen kritischen Laufzeitfehler.
- Der Aufrufer kann dann versuchen, den Fehler zu beheben.

Ausnahmen werfen

Das Eintreten einer Ausnahme wird in C# mit der Anweisung `throw` signalisiert.

- Man sagt, eine Ausnahme wird *geworfen*.

Die Syntax von `throw` lautet: `throw e;`

- Dabei steht `e` für ein Objekt der Klasse `System.Exception` bzw. einer abgeleiteten Klasse.
- Dadurch können dem Aufrufer zusätzliche Informationen zum Fehler übermittelt werden.
- Im einfachsten Fall trägt das Objekt lediglich eine Fehlermeldung, z.B. `throw new Exception("Nenner darf nicht 0 werden!");`

Durch das Werfen der Ausnahme wird der Programmablauf unterbrochen.

- Die aktuelle Methode wird sofort verlassen.
- Der Kontrollfluss springt zum Aufrufer zurück.
- Es wird auch keine Rückgabe erzeugt

Ausnahme in der Eigenschaftsmethode werfen

Die Eigenschaftsmethode bzw. den Setter für den Nenner können wir nun erweitern.

- Ein fehlerhafter Wert soll dazu führen, dass eine Ausnahme geworfen wird.

```
1 public int Nenner
2 {
3     get { return nenner; }
4     set
5     {
6         if (value == 0)
7             throw new Exception("Nenner darf nicht 0 werden!");
8
9         nenner = value;
10    }
11 }
```

Der Versuch, den Nenner auf 0 zu setzen, wird nun mit einer Fehlermeldung abgebrochen.

- Da wir im Konstruktor ebenfalls die Eigenschaftsmethode nutzen, kann daher nun auch kein Objekt mehr mit einem falschen Nenner erzeugt werden.

Ausnahmen behandeln

Eine unbehandelte Ausnahme führt zum Abbruch des Programms.

- Das ist besser, als mit falschen Daten weiterzuarbeiten.

Der Aufrufer einer Methode kann aber versuchen, den Fehler zur Laufzeit zu beheben.

- Man könnte den Nutzer z.B. nach anderen Werten fragen.

Ausnahmen können mithilfe eines try-catch-Blocks behandelt werden.

- Die potenziell gefährlichen Anweisungen werden dabei von einem try-Block umschlossen.
- Darauf folgt mindestens eine catch-Klausel, um den Fehler zu behandeln, z.B.:

```

1  try
2  {
3      // hier steht Programmcode, der eine Ausnahme werden könnte
4  }
5  catch (Exception e)
6  {
7      // Hier steht Programmcode, um die Ausnahme zu behandeln
8  }

```

Aufgabe

Lesen Sie Zähler und Nenner von der Konsole ein.

- Erzeugen Sie daraus ein Bruch-Objekt.
- Sollten Ausnahmen auftreten, sollen die Werte erneut abgefragt werden.

Lösung

```

1  Bruch b = null;
2  bool fehler_aufgetreten = false;
3
4  do
5  {
6      fehler_aufgetreten = false;
7      Console.Write("Bitte zähler eingeben: ");
8      int zaehler = Convert.ToInt32(Console.ReadLine());
9      Console.Write("Bitte Nenner eingeben: ");
10     int nenner = Convert.ToInt32(Console.ReadLine());
11
12     try

```

4 Objekte zur Laufzeit

```
13     {
14         b = new Bruch(zaehler, nenner);
15     }
16     catch (Exception e)
17     {
18         Console.WriteLine("Sorry! Es wurden falsche Werte eingegeben!");
19         fehler_aufgetreten = true;
20     }
21 } while (fehler_aufgetreten);
```

Mehrere catch-Klauseln

Die Ausnahmebehandlung kann auf unterschiedliche Fehlerarten individuell eingehen.

- Dazu können mehrere catch-Klauseln genutzt werden.
- Ein finally-Block kann zudem für abschließende Aufräumarbeiten genutzt werden.

```
1  try
2  {
3      // Hier steht Programmcode, der zur Laufzeit eine Ausnahme erzeugen kann
4  }
5  catch (FileNotFoundException e)
6  {
7      // Dieser Teil wird aufgerufen, wenn ein Fehler vom Typ FileNotFoundException geworfen wurde.
8  }
9  catch (ArgumentException e)
10 {
11     // Dieser Teil wird aufgerufen, wenn ein Fehler vom Typ ArgumentException geworfen wurde.
12 }
13 finally
14 {
15     // Dieser Teil wird immer ausgeführt, unabhängig davon, ob ein Fehler auftritt, oder nicht.
16     // Das ist gut, um Ressourcen freizugeben, z.B. eine Datei zu schließen
17 }
```

Praxis der Fehlerbehandlung

Die Fehlerbehandlung mithilfe von Ausnahmen ist eine wichtige Programmiertechnik.

- Objekte müssen gegen ungültige Daten abgesichert werden.
- Bei allen Methoden sollten daher die sog. **Vorbedingungen** geprüft werden.

Die Klassen des .Net-Framework werfen im Fehlerfall ebenfalls Ausnahmen.

- Die Klasse `Convert` wirft z.B. Ausnahmen, wenn Daten nicht konvertiert werden können.

- Die Anweisung `Convert.ToInt32("Hallo")` wirft eine Ausnahme vom Typ `FormatException`.

Das .Net-Framework bietet eine Reihe von Klassen, die von der Basisklasse `Exception` ableiten.

- z.B. `FileNotFoundException`, `ArgumentException`, ...
- Wenn möglich sollte eine dieser bestehenden Klassen benutzt werden.
- Man kann aber auch eigene Fehlerklassen ableiten (siehe Vererbung).

4.4 Objektidentität

Werte- und Referenztypen

Es gibt in C# zwei Typen von Variablen.

- Wertetypen und Referenztypen.

Variablen von Werttypen (z.B. `int` oder `bool`) beinhalten den Wert direkt.

- Variablen von Referenztypen hingegen speichern einen Verweis auf die Daten.

Objekte sind grundsätzlich Referenztypen.

- Eine Objektvariable kann auch auf nichts verweisen: `Bruch b = null;`
- Der Versuch auf `b` eine Eigenschaft oder Methode zu nutzen schlägt mit einer `NullReferenceException` fehl.

Dies hat entsprechende Konsequenzen bei Zuweisungen.

- Die Zuweisung zu einer Objektvariablen kopiert lediglich die Referenz.
- Es wird keine Kopie des Objekts selbst erzeugt.

Klon erzeugen

Wir erzeugen ein neues Objekt und weisen die Referenz `b` einer zweiten Variable `c` zu:

```
1 Bruch b = new Bruch();
2 b.Zaehler = 1;
3 Bruch c = b;
```

In `c` entsteht aber keine Kopie des Objekts.

- Lediglich die Referenz wird kopiert.
- Sowohl `b` als auch `c` verweisen nun auf dasselbe Objekt.
- Eine Änderung des Zählers auf `b` wird auch bei `c` eine entsprechende Auswirkung haben.

Um eine echte Kopie (`clone`) zu erzeugen, müssen die Daten einzeln kopiert werden:

```
1 Bruch kopie_von_b = new Bruch();
2 kopie_von_b.Zaehler = b.Zaehler;
3 kopie_von_b.Nenner = b.Nenner;
```

Dazu kann natürlich auch eine eigene Methode `Clone()` erstellt werden.

Identität

Objekte besitzen ihre eigene **Identität**.

- Zwei Objekte sind voneinander unterscheidbar.
- Selbst dann, wenn ihr innerer Zustand, ihre Daten, gleich sind.

Im folgenden Beispiel wird die Variable `gleich` den Wert `false` annehmen.

```
1 Bruch a = new Bruch(1,3);
2 Bruch b = new Bruch(1,3);
3 bool gleich = a == b;
```

Der Operator `==` prüft auf **Referenzgleichheit**.

- Der Ausdruck wird nur dann wahr, wenn beide Referenzen auf dasselbe Objekt verweisen.
- Dies ist hier nicht der Fall, da `a` und `b` auf unterschiedliche Objekte verweisen.

Wertgleichheit

Oft wollen wir aber nicht die Referenzen vergleichen, sondern die Attribute der Objekte.

- Das heißt, wir wollen die **Wertgleichheit** prüfen.
- Dazu bedarf es dann einer eigenen Methode.

In C# erben alle Klassen implizit von der Basisklasse `Object`.

- Mit dem Konzept der Vererbung werden wir uns später noch genauer auseinandersetzen.
- Die Basisklasse *Object* implementiert die Methode *Equals()*, d.h. alle Objekte besitzen diese Methode.
- Im Standardfall prüft *Equals()* ebenfalls auf Referenzgleichheit.
- Sie kann aber **überschrieben** werden, um auf Wertgleichheit zu prüfen.

Um eine geerbte Methode mit einer neuen Implementierung zu überschreiben, wird das Schlüsselwort `override` genutzt.

Equals

Die Methode *Equals()* erwartet ein beliebiges Objekt als Parameter.

- Es muss sich dann nicht notwendigerweise um ein Bruch-Objekt handeln.
- Vor einem Wertvergleich von Zähler und Nenner muss also erst eine explizite Typumwandlung vorgenommen werden.

```

1 public override bool Equals(Object o)
2 {
3     if (o == null)
4         return false;
5
6     Bruch b = o as Bruch;
7     if (b == null)
8         return false;
9
10    return b.Zaehler == Zaehler && b.Nenner == Nenner;
11 }

```

Konnte das übergebene Objekt erfolgreich in ein Bruch-Objekt umgewandelt werden, können Zähler und Nenner miteinander verglichen werden.

Wertvergleich anwenden

Wir können nun erneut zwei Bruchobjekte miteinander vergleichen.

- Anstelle des Operators `==` nutzen wir nun die Methode *Equals()*.

```
1 Bruch a = new Bruch(1,3);  
2 Bruch b = new Bruch(1,3);  
3 bool gleich = a.Equals(b);
```

Die überschriebene Methode *Equals()* prüft nun auf Wertgleichheit der Attribute.

- Entsprechend wird die Variable `gleich` nun den Wert `true` annehmen.

Größenvergleich

Die Methode *Equals()* kann lediglich dazu genutzt werden, um die Wertgleichheit oder -ungleichheit festzustellen.

- Sie kann keine Aussage über Größenverhältnisse liefern (größer oder kleiner).

Ein Größenvergleich wird aber z.B. dann benötigt, wenn wir Bruch-Objekte sortieren wollen.

- Für solche Aussagen müssen wir dann eine weitere Methode implementieren.
- Im .Net Framework wird dazu meist die Methode *CompareTo()* eingeführt.
- Diese wird in der Schnittstelle `IComparable` definiert.

Der Rückgabewert der Methode zeigt die Größenverhältnisse der beteiligten Objekte an:

- Das übergebene Objekt ist gleich groß, als das eigene Objekt: 0
- Das übergebene Objekt ist größer: 1
- Das übergebene Objekt ist kleiner: -1

CompareTo implementieren

```

1  class Bruch implements IComparable
2  {
3      ...
4
5      public int CompareTo(object obj)
6      {
7          if (obj == null)
8              return -1;
9
10         var b = obj as Bruch;
11         if (b == null)
12             throw new ArgumentException("Übergebenes Objekt ist kein Bruch!");
13
14         if (b.Dezimalzahl > Dezimalzahl)
15             return 1;
16         else if (b.Dezimalzahl < Dezimalzahl)
17             return -1;
18         else
19             return 0;
20     }
21 }

```

4.5 Zusammenfassung und Aufgaben**Zusammenfassung**

Wir haben heute gelernt...

- wie Objekte mithilfe eines Konstruktors initialisiert werden können.
- wie Namenskonflikte mithilfe von `this` aufgelöst werden können.
- wie Ressourcen mit einem Destruktor und noch besser mit `Dispose()` freigegeben werden.
- wie Laufzeitfehler mithilfe von Ausnahmen geworfen werden.
- wie solche Laufzeitfehler auch behandelt werden können.
- wie wir einen Klon eines Objekts erzeugen.
- dass Objekte unabhängig von ihrem inneren Zustand voneinander unterscheidbar sind.
- wie Objekte auf Referenz- und Wertgleichheit geprüft werden können.
- wie man den Größenvergleich von Objekten implementieren kann.

Aufgabe 1

Erweitern Sie die Klasse *Rechteck* aus der letzten Übung um einen Konstruktor.

- Werfen Sie Ausnahmen, wenn ungültige Werte an ein Objekt der Klasse *Rechteck* herangetragen werden sollen.
- Überschreiben Sie die Methode *Equals()*.
- Kann die Methode *CompareTo()* sinnvollerweise implementiert werden? Wie?

Aufgabe 2

Erstellen Sie eine Klasse *Bigint*.

- Ein Objekt der Klasse soll eine beliebig große Ganzzahl abbilden können.
- Intern werden die Ziffern der Zahl in einem *Int*-Array verwaltet.
- Die Klasse soll auch das Rechnen mit solchen Zahlen ermöglichen.

Realisieren Sie Ihre Klasse soweit, dass folgender Programmcode ein sinnvolles Ergebnis produziert:

```
1 var b1 = new Bigint("10000000000000000000000000000000");
2 var b2 = new Bigint("20000000000000000000000000000000");
3 var b3 = b1.Add(b2);
```

5 Schnittstellen und Aufzählungen

5.1 Klassenmethoden und -attribute

Main-Methode

Unser erstes Programm in C# war das Hello-World-Programm.

```
1 class Program
2 {
3     public static void Main()
4     {
5         Console.WriteLine("Hello, World!");
6     }
7 }
```

Wir wissen nun, dass wir hier eine neue Klasse *Program* deklariert haben.

- Die Main-Methode stellt den Einsprungspunkt in unsere Anwendung dar.

Diese Methode ist mit dem Modifizierer `static` deklariert.

- Eine statische Methode gehört zur Klasse, nicht zu einem Objekt.
- Entsprechend kann diese Methode ausschließlich auf der Klasse aufgerufen werden.

Beispiel

```
1 class Test
2 {
3     // Diese Methode kann nur auf der Klasse Test aufgerufen werden
4     public static void Klassenmethode() { }
5
6     // Diese Methode kann nur auf einem Objekt der Klasse Test aufgerufen werden.
7     public void Objektmethode() { }
8
9     public static void Main()
10    {
11        Test.Klassenmethode();
12
13        Test t = new Test();
```

```
14         t.Objektmethode();  
15     }  
16 }
```

Console

Solche **Klassenmethoden** sind sinnvoll, wenn Funktionalität nicht objektspezifisch ist.

- Als Beispiel haben wir schon die Klasse *Console* gesehen.
- Die Konsole existiert nur einmal.
- Es macht daher wenig Sinn, Objekte der Klasse erzeugen zu müssen, um mit der Konsole zu arbeiten.
- Methoden, wie *WriteLine()* oder *ReadLine()* sind daher Klassenmethoden.
- Diese Methoden werden auf der Klasse aufgerufen, z.B. `Console.WriteLine("Hello");`

Auch die Klassen *Math* und *Convert* bieten ihre Funktionalität über Klassenmethoden an.

- Die Klasse *Math* stellt mathematische Methoden bereit, z.B. `Math.Sin(0.5)`.
- Die Klasse *Convert* hilft bei der nichttrivialen Konvertierung elementarer Datentypen.

5.2 Arrays und Datenstrukturen

Datenstrukturen

Die Objektorientierung stellt Strukturen bereit, um komplexe Probleme zu lösen.

- Große Systeme werden mithilfe von Objekten in kleinere Teile zerlegt.
- Die Objekte wirken dann zur Laufzeit zusammen, um ein gemeinsames Ziel zu erreichen.

Jedes Objekt soll dabei **überschaubar komplex, wartbar und in sich abgeschlossen** sein.

- Es ist aber nicht einfach, ein komplexes System sinnvoll auf Objekte aufzuteilen.
- Damit werden wir uns noch ausführlich befassen.

Klar ist aber, dass zur Laufzeit viele Objekte miteinander interagieren müssen.

- Wir müssen also in der Lage sein, viele Objekte zu verwalten.
- Dazu brauchen wir zunächst entsprechende **Datenstrukturen**.

Arrays von Objekten

Arrays stellen die einfachste Möglichkeit dar, um viele Elemente gleichen Typs zu verwalten.

- Arrays können nicht nur elementare Datentypen, wie `int` oder `char` in sich aufnehmen.

Wir können natürlich auch Arrays deklarieren, um Objektreferenzen zu speichern.

- In C# sind Arrays ebenfalls Objekte, die von der abstrakten Basisklasse *Array* erben.
- Entsprechend wird ein Array auch mit dem `new` Operator deklariert:

```
1 Bruch[] brueche = new Bruch[20];
```

Dadurch wurden allerdings noch keine Bruch-Objekte erzeugt.

- Das Array verweist an allen Positionen noch auf `null`.
- Der Versuch, den Punktoperator auf eine beliebige Position anzuwenden, schlägt mit einer `NullReferenceException` fehl.

Arrays von Objekten benutzen

In einem solchen Array kann nun an einer Position kleiner der vordefinierten Größe ein neues Bruch-Objekt abgelegt werden.

- Die Position wird durch einen ganzzahligen Index bestimmt.

```
1 brueche[0] = new Bruch(1,2);
```

Genau so kann natürlich nun auch auf das entsprechende Objekt zugegriffen werden.

- Der Punktoperator ermöglicht den Zugriff auf die öffentlichen Elemente des Objekts.

```
1 brueche[0].Nenner = 5;
2 brueche[0].Ausgeben();
```

Mit Hilfe von Schleifen kann nun auch über das gesamte Array iteriert werden:

```
1 for (int i=0; i<20; i++)  
2     if (brueche[i] != null)  
3         brueche[i].Ausgeben();
```

5.3 Auflistungen und ArrayList

Auflistungen

Arrays haben den Vorteil, dass man über den Index effizient auf jedes Element zugreifen kann.

- Arrays können aber nicht wachsen oder schrumpfen.
- Einmal deklariert, bleiben sie in ihrer Größe konstant.

Oft wissen wir aber nicht, wie viele Elemente wie zur Laufzeit verwalten müssen.

- Entsprechend werden dynamische Datenstrukturen benötigt.

Solche Datenstrukturen werden in .Net als **Auflistungen (engl. Collections)** bezeichnet.

- Das .Net Framework beinhaltet eine Vielzahl fertig nutzbarer Auflistungen.
- Die entsprechenden Klassen finden sich meist im Namensraum `System.Collections`.

Wie diese Klassen funktionieren und intern aufgebaut sind, lernen wir in der Veranstaltung Algorithmen und Datenstrukturen.

ArrayList

Objekte der Klasse *ArrayList* kann man sich wie dynamische Arrays vorstellen.

- Über einen ganzzahligen Index kann auf die Elemente zugegriffen werden.
- Die Größe der Liste kann sich aber zur Laufzeit dynamisch an den Bedarf anpassen.
- Darüber hinaus bietet die Klasse eine Vielzahl von Methoden an.

Mit *Add()* oder *Insert()* können Elemente in die Liste aufgenommen werden.

```

1 Bruch a = new Bruch(1,3);
2 Bruch b = new Bruch(1,6);
3 l.Add(b);           // Neues Element hinten an die Liste anhängen
4 l.Insert(0, a);     // Neues Element an einer bestimmten Position einfügen
5 int anzahl = l.Count; // ergibt 2

```

Mit *Contains()* oder *IndexOf()* kann die Liste linear durchsucht werden:

```

1 bool ist_drin = l.Contains(a); // ergibt true
2 int pos = l.IndexOf(b);       // ergibt 1

```

Methoden der Klasse *ArrayList*

Mit *Remove()* oder *RemoveAt()* werden Elemente wieder entfernt.

```

1 l.Remove(a); // liefert true zurück, wenn das Element a erfolgreich entfernt werden konnte
2 l.Remove(0); // Wirft eine Ausnahme, wenn die Position nicht vorhanden ist.

```

Über den Index kann ein Element gelesen, als auch geschrieben werden.

```

1 l[0] = new Bruch(1, 3);
2
3 // Das Objekt aus der Liste muss erst wieder in ein Bruch-Objekt umgewandelt werden:
4 var obj = l[0] as Bruch;

```

Überall dort, wo wir sonst ein Array eingesetzt haben, kann nun leicht ein Objekt der Klasse *ArrayList* benutzt werden.

- Das hat den enormen Vorteil, dass nicht vorher schon klar sein muss, wie viele Elemente verwaltet werden sollen.

5.4 Generische Klassen

Typsicherheit

Ein Nachteil der Klasse *ArrayList* ist, dass die Auflistungen nicht typsicher ist.

- Wir können in der Liste ganz unterschiedliche Elemente ablegen.
- Alle Elemente werden intern als *Object* verwaltet (siehe Abschnitt Vererbung).

```
1 ArrayList l = new ArrayList();
2 l.Add("Apfel");
3 l.Add(42);
4 l.Add(new Bruch(1,4));
```

Lesen wir ein Element einer *ArrayList*, wird es als Objekt vom Typ *Object* geliefert.

- Vor der Benutzung müssen wir ein solches Element erst wieder in den Ursprungstyp zurückwandeln:

```
1 var bruch = l[3] as Bruch;
2 bruch.Ausgeben();
```

Generische Klassen

Dass die *ArrayList* unterschiedliche Typen in sich aufnehmen kann, ist meist eher lästig.

- In der Regel wollen wir in einer Liste nur Elemente eines einzigen Typs verwalten, z.B. *Bruch*-Objekte.

Am besten wäre eine Liste, die von vornherein nur *Bruch*-Objekte verwalten kann.

- Natürlich wäre es aber unsinnig für jeden Datentyp eine eigene Listenklasse erstellen zu müssen, z.B. eine *BruchArrayList*.

Wir müssten eine Möglichkeit haben, der *ArrayList* zu sagen, dass wir ausschließlich mit Objekte der Klasse *Bruch* arbeiten wollen.

- Wir müssten den Datentyp **parametrisieren** können.

Genau das erlauben **generische Datentypen** (engl. **generics**).

- Diese wurden mit Version 2.0 in die Sprache C# eingeführt.

Generische Klassen

Generische Klassen und Methoden erlauben die Einführung von **Typparametern**.

- Wir können dadurch Klassen konstruieren, die von konkreten Datentypen abstrahieren.
- Das ist insbesondere bei Datenstrukturen, wie z.B. einer Liste sehr hilfreich.

Wir können z.B. die Klasse *ListNode* aus der Veranstaltung Algorithmen und Datenstrukturen umbauen.

```

1 public class ListNode<T> where T : IComparable
2 {
3     public T Value { get; set; }
4     public ListNode Next { get; set; }
5
6     public ListNode(T value)
7     {
8         Value = value;
9     }
10 }

```

Generische Klasse benutzen

Überall dort, wo zuvor der Datentyp `int` genutzt wurde, wird nun der Platzhalter T verwendet.

- Welche Arten von T erlaubt werden, kann zudem weiter eingeschränkt werden.
- In unserem Fall wünschen wir nur solche Datentypen, die einen Größenvergleich erlauben.

Erzeugen wir nun Objekte aus der Klasse, müssen wir angeben, wofür T konkret steht:

```

1 ListNode<double> a = new ListNode<double>(3.142);
2 ListNode<char> b = new ListNode<char>('u');

```

Auf dieser Basis kann dann auch die Klasse *LinkedList* umgebaut werden.

- Wir können dann Objekt genau eines Datentyps in ihr verwalten.
- Wir müssen dann auch beim Zugriff auf die Elemente keine Typumwandlung mehr durchführen.

Generische Klassen im .Net-Framework

Mit der Einführung der generischen Klassen in C# wurden im .Net-Framework auch neue Auflistungsklassen eingeführt.

- Die generischen Varianten der bisher verfügbaren Collections sind im Namensraum `System.Collections.Generic` abgelegt.

Die generische Variante der Klasse `ArrayList` ist die Klasse `List<T>`.

- Wann immer möglich sollten die generischen Varianten der Collection-Klassen bevorzugt werden!

```
1 var l = new List<int>();
2 l.Add(42);
3
4 // Folgende Anweisung würde einen Übersetzungsfehler erzeugen:
5 // l.Add("Hallo");
6
7 // Die Liste liefert nun auch direkt int-Werte:
8 int wert = l[0];
```

Weitere Auflistungsklassen im .Net-Framework

Die generische Klasse *List* legt die Elemente intern in einem Array ab.

- Über den Index kann dann sehr effizient auf die Elemente zugegriffen werden.

Ein **assoziatives Array** erlaubt die effiziente Suche nach einem bestimmten Element (ansatzweise in $O(1)$).

- Im .Net-Framework existiert dazu die generische Klasse *Dictionary*.

Binäre Suchbäume organisieren die Elemente wiederum auf eine andere Art und Weise.

- Ein höhenbalancierter Baum kann z.B. das größte/kleinste Element leicht bestimmen.
- Die Klasse *SortedDictionary* des .Net-Framework bildet eine solche Struktur ab.

Darüber hinaus sind viele weitere Auflistungsklassen im .Net-Framework verfügbar.

- z.B. *LinkedList*, *Stack*, *Queue*, *HashSet*, ...

5.5 Schnittstellen

Gemeinsamkeiten

Alle diese Klassen implementieren spezifische Datenstrukturen.

- Die Art und Weise, wie sie ihre Elemente verwalten, unterscheidet sich aber stark.
- Dies dient bekanntlich dazu, bestimmte Operationen effizient abbilden zu können.
- In einem Algorithmus wählt man daher diejenige Datenstruktur, welche die meiste Effizienz verspricht.

All diese Datenstrukturen besitzen aber gewisse **Gemeinsamkeiten**.

- Unabhängig davon, wie die Daten intern verwaltet werden, wollen wir gewisse Grundoperationen auf den Datenstrukturen durchführen können.
- Wir wollen z.B. Elemente hinzufügen oder entfernen können.

Methoden, die von mehreren Klassen angeboten werden sollen, können wir in **Schnittstellen** sammeln.

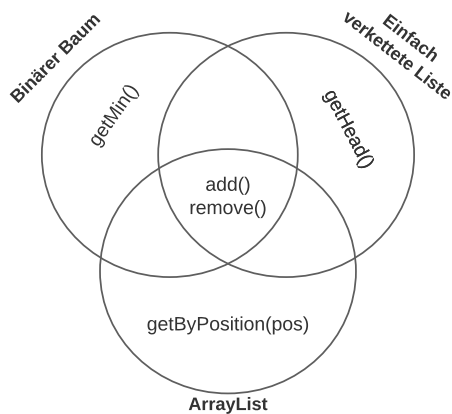
Schnittstelle

Schnittstellen definieren Gemeinsamkeiten über mehrere Klassen hinweg.

- z.B. das gemeinsame Verhalten von Datenstrukturen, Elemente hinzufügen und entfernen zu können.

Eine Schnittstelle deklariert Methoden, ohne diese zu implementieren.

- Eine Klasse kann eine oder mehrere Schnittstellen implementieren.



Schnittstelle als Vertrag

Schnittstellen gleichen einem **Vertrag** zwischen Dienstanbieter (Server) und Nutzer (Client).

- Implementiert eine Klasse (Server) eine bestimmte Schnittstelle, kann sich der Nutzer (Client) auf bestimmte Fähigkeiten (Methoden) verlassen.
- Jede Klasse, die eine Schnittstelle implementiert, wird gezwungen, eine Implementierung aller Methoden der Schnittstelle anzubieten.

5 Schnittstellen und Aufzählungen

Eine Schnittstelle kann von vielen unterschiedlichen Klassen implementiert werden.

- Jede Klasse verfügt dann über gleiche, gesicherte Fähigkeiten.

Comparable und Disposable

Wir haben bereits zwei solcher Schnittstellen kennen gelernt, *Comparable* und *Disposable*.

- Objekte, deren Klasse diese Schnittstellen implementieren, verfügen über ganz bestimmte Fähigkeiten, z.B. Größenvergleiche anstellen zu können bzw. Ressourcen freizugeben.

Eine Schnittstelle wird implementiert, indem man ihren Namen mit dem Doppelpunkt hinter dem Klassennamen aufführt.

- Mehrere Schnittstellen werden durch ein Komma voneinander getrennt angegeben.

```
1 class Bruch : Comparable
2 {
3     public int CompareTo(object obj)
4     {
5         // Hier muss eine sinnvolle Implementierung stehen
6     }
7 }
```

IAusgebbar

Wir können auch eigene Schnittstellen definieren.

- Wir könnten z.B. eine Schnittstelle *IAusgebbar* deklarieren.
- Klassen, die diese Schnittstelle implementieren, sorgen dafür, dass ihre Objekte auf der Konsole ausgegeben werden können.

Eine Schnittstelle wird mit dem Schlüsselwort `interface` deklariert.

- Alle Methoden der Schnittstelle sind öffentlich.
- Entsprechend muss kein Zugriffsmodifizierer angegeben werden.

```
1 interface IAusgebbar
2 {
3     void Ausgeben();
4 }
```

Der Schnittstellenname sollte großgeschrieben werden und mit einem führenden I beginnen.

- Die Schnittstelle sollte in einer eigenen Datei abgelegt werden (wie Klassen).

Schnittstelle implementieren

Die Schnittstelle kann nun von beliebig vielen Klassen implementiert werden:

```

1  class Bruch : IAusgebbar
2  {
3      // Alle anderen Methoden werden hier nicht aufgeführt...
4
5      public void Ausgeben()
6      {
7          Console.WriteLine(zaehler + "/" + nenner);
8      }
9  }
```

Die Klasse muss dann jede Methode der Schnittstelle implementieren.

- Die Implementierung der Schnittstelle sieht natürlich in jeder Klasse anders aus.
- Die Ausgabe eines Bruch-Objekts unterscheidet sich intern von der Ausgabe z.B. eines Mitarbeiterobjekts.

Schnittstelle als Datentyp

Schnittstellen sind (genauso wie auch Klassen) Datentypen.

- Ein Objekt, dessen Klasse eine bestimmte Schnittstelle implementiert, kann als Instanz der Schnittstelle angesehen werden.

```

1  Bruch b = new Bruch();
2  IAusgebbar c = b;
```

Auf dem Objekt *c* können dann allerdings nur solche Methoden aufgerufen werden, die durch *IAusgebbar* deklariert wurden.

Es kann auch eine Methode deklariert werden, die einen Parameter vom Typ *IAusgebbar* erwartet:

```

1  public void MacheEtwas(IAusgebbar b) { ... }
```

Einer solchen Methode kann jedes Objekt einer Klasse übergeben werden, dass die Schnittstelle *IAusgebbar* implementiert.

Abstraktion

Dies ist ein extrem wichtiger Mechanismus!

- Eine Schnittstelle hilft dabei, von konkreten Klassen zu abstrahieren.

Mit Schnittstellen können Anforderungen an Objekte reduziert werden.

- Nicht der schwergewichtige Bruch wird verlangt, sondern lediglich etwas *ausgebautes*.
- Dies reduziert Abhängigkeiten zwischen Programmteilen.
- Ein wichtiger Beitrag zu erweiterbarer/evolvierbarer Software.

Wie wir diesen Mechanismus gewinnbringend einsetzen, werden wir später noch sehen.

- Siehe Abschnitt zur Polymorphie.

5.6 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt...

- was Klassenmethoden sind und wo man sie sinnvollerweise einsetzt.
- wie mit Hilfe von Arrays viele Objekte verwaltet werden können.
- was der Vorteil von Datenstrukturen, wie z.B. der Klasse *ArrayList* ist.
- wie wir mit Hilfe von Generics solche Auflistungen typsicher machen können.
- was Schnittstellen sind, wie man sie deklariert und implementiert.
- dass Schnittstellen auch Datentypen sind und was das bedeutet.

Aufgabe 1

Erzeugen Sie die folgenden 20 Bruchobjekte und legen Sie sie in einem Array ab:

- $\frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$
- Was ist das für eine Reihe?

Schreiben Sie eine Methode, um die Summe der Reihe zu bestimmen.

Ersetzen Sie das Array durch ein Objekt der Klasse *List*.

- Mussten Sie die Methode zur Berechnung der Summe ändern?

Aufgabe 2

Erstellen Sie eine neue Klasse *GeometrischeReihe*.

- Die Klasse soll Brüche wie in der vorgehenden Aufgabe in eine Objektvariable vom Typ *List* verwalten.
- Die Anzahl der Objekte soll dem Konstruktor übergeben werden.
- Eine Eigenschaftsmethode soll die Summe der Bruch-Objekte als Dezimalzahl liefern können.

Erstellen Sie eine Methode, um eine Liste von Bruch-Objekten zurückzuliefern, deren Nenner größer ist als ein übergebener Wert.

- Welcher Datentyp sollte für das Resultat dieser Suche genutzt werden?

Aufgabe 3

Im Begleitprojekt der Veranstaltung *Algorithmen und Datenstrukturen* ist die Klasse *ArrayList* zu finden¹.

- Passen Sie die Klasse so an, dass beliebige Datentypen und nicht nur `int` gespeichert werden können.

¹<https://github.com/LosWochos76/AUD>

Aufgabe 4

Erstellen Sie eine Klasse *Mitglied*, deren Objekte Mitglieder eines Sportvereins repräsentieren.

- Ein einzelnes Mitglied hat einen Vornamen, Nachnamen und ein Geschlecht.
- Erstellen Sie entsprechende Eigenschaftsmethoden und einen Konstruktor.
- Erstellen Sie einige Objekte und legen sie in einem Array ab.

Aufgabe 5

Erstellen Sie eine weitere Klasse *MitgliederListe*.

- Die Klasse soll beliebig viele Objekte der Klasse *Mitglied* verwalten können.
- Erstellen Sie eine Methode, um Mitglieder hinzuzufügen.

Erstellen Sie Methoden, um Mitglieder nach Namen oder Geschlecht zu suchen und entsprechende Objekte zurückzugeben.

- Welcher Datentyp sollte für das Ergebnis der Suche sinnvollerweise genutzt werden?

6 Objektbeziehungen

6.1 Vereinsverwaltung

Mitglied

In den letzten Abschnitten haben wir die *Bruch*-Klasse realisiert.

- Dadurch konnten wir viele konkrete *Bruch*-Objekte erzeugen.
- Mit Hilfe von Methoden konnten wir dadurch die Bruchrechnung im Rechner abbilden.

Wir wollen nun noch ein anderes Beispiel umsetzen.

- Wir wollen damit beginnen, eine Verwaltungssoftware für einen Sportverein zu implementieren.

Ein Sportverein hat bekanntlich Mitglieder.

- Jedes Mitglied hat einen Vor- und einen Nachnamen.
- Zudem noch ein Geburtsdatum.

Um Mitglied-Objekte erzeugen zu können, benötigen wir also die *Mitglied*-Klasse.

Klasse Mitglied

Die *Mitglied*-Klasse besitzt Eigenschaftsmethoden für Vor- und Nachnamen, sowie das Geburtsdatum.

- In den Set-Teilen der Eigenschaftsmethoden benötigen wir keine besondere Prüfung der übergebenen Werte.
- Entsprechend können wir auch die abgekürzte Schreibweise nutzen.

6 Objektbeziehungen

```
1 class Mitglied
2 {
3     public string Vorname { get; set; }
4     public string Nachname { get; set; }
5     public DateTime Geburtsdatum { get; set; }
6
7     public Mitglied(string v, string n, string g)
8     {
9         Vorname = v;
10        Nachname = n;
11        Geburtsdatum = Convert.ToDateTime(g);
12    }
13 }
```

Sportverein

Nun reicht die *Mitglied*-Klasse allein aber nicht aus, um eine Verwaltungssoftware für einen Sportverein umzusetzen.

- Der Sportverein selbst besitzt ja auch Eigenschaften.
- Er hat einen Namen, er *kennt* einen Vorsitzenden und seine Mitglieder.

In der objektorientierten Programmierung ist alles ein Objekt.

- Also auch der Sportverein.

Für den Sportverein erstellen wir dementsprechend auch eine Klasse.

- Das ist notwendig und sinnvoll.
- Auch dann, wenn wir zur Laufzeit nur ein einziges Objekt dieser Klasse besitzen werden.

Klasse Sportverein

Für alle Daten, die ein Objekt der Klasse *Sportverein* verwalten soll, werden Objektvariablen benötigt.

- Der Name ist eine einfache Zeichenkette vom Typ `string`.
- Dazu erstellen wir eine Eigenschaftsmethode.

Der Vorsitzende und die Mitglieder werden allerdings über Objektreferenzen abgebildet.

```

1 class Sportverein
2 {
3     private Mitglied vorsitzender;
4     private List<Mitglied> mitglieder = new List<Mitglied>();
5
6     public string Name { get; set; }
7
8     public Sportverein(string n)
9     {
10         Name = n;
11     }
12 }

```

Vorsitzender

Die Objektvariable *vorsitzender* kann eine Objektreferenz auf ein einzelnes *Mitglied*-Objekt aufnehmen.

- Die Objektvariable ist zunächst vor dem Zugriff von außerhalb des Objekts geschützt (*private*).
- Über eine Eigenschaftsmethode können wir aber erneut den Zugriff gewähren.
- Im Set-Teil können wir dann prüfen, ob die übergebene Objektreferenz auch auf ein Objekt verweist.

```

1 public Mitglied Vorsitzender
2 {
3     get { return vorsitzender; }
4     set
5     {
6         if (value == null)
7             throw new Exception("Der Verein muss einen Vorsitzenden haben!");
8
9         vorsitzender = value;
10    }
11 }

```

Mitglieder hinzufügen

Die Objektvariable *mitglieder* ist ein Objekt der Klasse *List*.

- Das Objekt kann beliebig viele Objektreferenzen auf Mitglied-Objekte verwalten.
- Auch diese Objektvariable ist sinnvollerweise zunächst privat.

Um dem Sportverein Mitglieder hinzufügen zu können, erstellen wir eine Methode:

6 Objektbeziehungen

```
1 public void MitgliedHinzufuegen(Mitglied m)
2 {
3     mitglieder.Add(m);
4 }
```

Der Sportverein nimmt dann die übergebene Objektreferenz in seiner eigenen Objektvariable auf.

- Der Sportverein *kennt* dadurch seine Mitglieder.

Mitglieder zurückgeben

Nachdem wir so dem Sportverein Mitglieder bekannt machen können, wollen wir den Sportverein auch fragen können, welche Mitglieder er kennt.

- Dazu implementieren wir natürlich erneut eine eigene Methode.

```
1 public IEnumerable<Mitglied> AlleMitglieder()
2 {
3     return mitglieder;
4 }
```

Als Antwort der Methode sollte nicht die Objektvariable *mitglieder* selbst zurückgegeben werden.

- Da dann ja die Objektreferenz selbst zurückgegeben wird, könnte der Nutzer die Liste dadurch dann manipulieren, z.B. komplett leeren.
- Stattdessen sollte ein Objekt des Typs *IEnumerable* zurückgegeben werden.
- Da die Klasse *List* diese Schnittstelle implementiert, kann sie implizit in diesen Typ konvertiert werden.

Sportverein benutzen

Mit der Klasse *Mitglied* können wir nun Mitglieder als Objekte abbilden.

- Die Klasse *Sportverein* überträgt das Konzept des Sportvereins selbst in den Rechner.

```
1 Sportverein s = new Sportverein("TUS Hamm");
2 s.Vorsitzender = new Mitglied("Erika", "Wischnowski", "3.2.2003");
3 s.MitgliedHinzufuegen(new Mitglied("Peter", "Pan", "6.8.1997"));
4
5 foreach (var m in s.AlleMitglieder())
6 {
```



```

7 Console.WriteLine(m.Vorname);
8 }

```

Durch die beiden Klassen können wir zur Laufzeit ein Geflecht aus Objekten erzeugen.

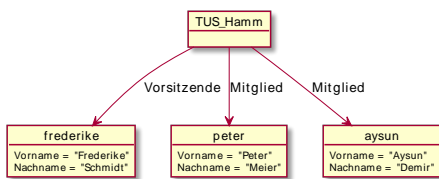
- Der Sportverein kennt seinen Vorsitzenden und seine Mitglieder.
- Der Sportverein kann die Methoden und Eigenschaften der Mitglied-Objekte nutzen.
- Dadurch können Aufgaben auf unterschiedliche Objekte verteilt werden.

6.2 Objektbeziehungen

Objektbeziehungen

Das Beispiel zeigt, wie **Objektbeziehungen** realisiert werden können.

- Auf dieser Basis interagieren Objekte in objektorientierten Systemen miteinander.
- In unserem Beispiel kann der Sportverein dann die Eigenschaften und Methoden des Mitglieds nutzen, um z.B. bestimmte Mitglieder zu filtern.



Solche Objektbeziehungen sind unidirektional, sie zeigen in eine bestimmte Richtung.

- Im Beispiel kennt der Sportverein seine Mitglieder.
- Die Mitglied-Objekte wissen aber nichts von einem Sportverein-Objekt.

Strukturen

Objekte und Objektbeziehungen bilden die **Grundlage der Objektorientierung**.

- Mit diesen Bausteinen lassen sich beliebige Strukturen im Rechner abbilden.
- Solche Strukturen aufzubauen zu können, ist der Kern der Objektorientierung.

Mithilfe von Objekten und Objektbeziehungen können beliebige Systeme entworfen werden.

6 Objektbeziehungen

- Sinnvolle Objektgeflechte können für alle Bereiche gefunden werden.
- Ganz egal, ob es sich um ein Computerspiel, ein Steuer- und Regelungssystem für ein Kernkraftwerk oder eine Finanzbuchhaltung handelt.

Entscheidend dabei ist, wie die Aufteilung auf Objekte gewählt wird.

- Über die entsprechenden Entwurfsprinzipien werden wir im weiteren Verlauf noch reden.

6.3 Tic Tac Toe

Tic Tac Toe

Das Spiel Tic Tac Toe kennen wir bereits.

- Bislang haben wir das Spiel allerdings noch nicht objektorientiert umgesetzt.
- Entsprechend konnten wir den Spielablauf, das Spielfeld und die Spieler nicht konzeptuell voneinander trennen.
- Alles war irgendwie miteinander vermischt.

Mithilfe der Objektorientierung können wir nun einen besseren Entwurf realisieren.

- Das Spielfeld und die Spieler setzen wir getrennt voneinander als Objekte um.

Ein weiteres Objekt bildet das Spiel und seinen Ablauf selbst ab.

- Dieses Objekt referenziert die anderen Objekte und nutzt sie um den Spielablauf zu steuern.

Spielfeld

Wir beginnen mit dem Spielfeld und erstellen dazu eine gleichnamige Klasse.

- Um die Belegung des Spielfelds abzubilden, nutzen wir natürlich intern ein Array.
- Der Zugriff darauf wird aber ausschließlich über entsprechende Methoden gestattet.

```

1  class Spielfeld
2  {
3      private char[,] feld;
4
5      public Spielfeld()
6      {
7          // Hier das feld initialisieren
8      }
9
10     public void Setzen(int zeile, int spalte, char spielstein)
11     {
12         feld[zeile, spalte] = spielstein;
13     }
14 }

```

Weitere Methoden des Spielfelds

Das Setzen des Spielsteins basiert auf einer einfachen Zuweisung in einem Array.

- Dennoch ist es sehr sinnvoll, dies in Form einer eigenen Methode zu implementieren.
- Wenn die Methode später benutzt wird, versteht man durch die Benennung sehr schnell, was gemacht wird.

Um mit dem Spielfeld vernünftig interagieren zu können, werden weitere Methoden benötigt.

- Sinnvoll ist z.B. eine Methode zur Ausgabe auf der Konsole: `void Ausgeben() { ... }`

Auch werden mehrere Prüfungen benötigt, z.B.:

- ob ein Platz im Spielfeld belegt ist: `public void IstBelegt(int zeile, int spalte) { ... }`
- ob ein bestimmter Spieler gewonnen hat: `public bool HatGewonnen(char spielstein) { ... }`
- ob irgendein Spieler gewonnen hat: `public bool HatIrgendwerGewonnen() { ... }`
- ob noch irgendein Feld frei ist: `public bool IstEinFeldFrei() { ... }`

Es sollte nicht schwierig sein, diese Methoden zu implementieren.

Spieler

Das Spielfeld allein macht noch kein Tic Tac Toe.

- Wir benötigen noch zwei Spieler, die ihre Spielsteine auf dem Spielfeld ablegen können.
- Dies sind natürlich auch Objekte, die wiederum das Spielfeld-Objekt benutzen.

Später wollen wir dafür sorgen, dass ein Mensch gegen den Computer spielen kann.

- Zunächst wollen wir das Spiel aber so umsetzen, dass zwei Computergegner gegeneinander spielen.

Entsprechend erstellen wir eine Klasse *ComputerSpieler*.

- Zur Laufzeit erzeugen wir dann zwei Objekte dieser Klasse.
- Sie benutzen ein Objekt der Klasse Spielfeld, um ihre Spielsteine dort abzulegen.

Klasse ComputerSpieler

Jedes Objekt der Klasse *ComputerSpieler* muss sich seinen Spielstein merken.

- Also benötigen wir eine entsprechende Eigenschaft, in der wir dann ein X oder ein O ablegen können.

```
1 class Spieler
2 {
3     public char Spielstein { get; set; }
4
5     public Spieler(char spielstein)
6     {
7         // Der eigene Spielstein (X oder O) wird dem Konstruktor übergeben
8         Spielstein = spielstein;
9     }
10 }
```

Spielstein setzen

Die wichtigste Methode eines Spielers ist *Ziehe()*.

- Als Parameter wird ein Spielfeld-Objekt übergeben.
- Der *ComputerSpieler* legt seinen Spielstein einfach auf einem zufälligen freien Feld ab.

```

1 public void Ziehe(Spielfeld feld) {
2     if (!feld.IstEinFeldFrei())
3         return;
4
5     while (true) {
6         int zeile = rnd.Next(0, 3);
7         int spalte = rnd.Next(0, 3);
8
9         if (!feld.IstBelegt(zeile, spalte)) {
10             feld.Setzen(zeile, spalte, Spielstein);
11             return;
12         }
13     };
14 }

```

Spiel

Zu guter Letzt müssen wir diese Objekte noch miteinander zu einem Spiel verbinden.

- Auch dazu erstellen wir eine neue Klasse *TicTacToe*.
- Ein Objekt der Klasse kennt ein Spielfeld und zwei Objekte der Klasse *ComputerSpieler*.

```

1 class TicTacToe
2 {
3     private Spielfeld feld;
4     private ComputerSpieler spieler1 = new ComputerSpieler('X');
5     private ComputerSpieler spieler2 = new ComputerSpieler('O');
6     private Spieler aktueller_spieler;
7
8     ...
9 }

```

Wir führen zudem noch ein weiteres Feld `aktueller_spieler` ein.

Aktueller Spieler

Der Spielverlauf wechselt bekanntlich zwischen den beiden Spielern hin und her.

- Dieses Verhalten können wir so abbilden, dass das Feld `aktueller_spieler` abwechselnd auf eines der beiden Spieler-Objekte verweist.
- Diesen Wechsel können wir mit einer eigenen Methode abbilden:

```
1 private void WechsleSpieler()
2 {
3     if (aktueller_spieler == spieler1)
4         aktueller_spieler = spieler2;
5     else
6         aktueller_spieler = spieler1;
7 }
```

Diese Mechanik können wir überhaupt nur deswegen abbilden, weil wir nun Spieler-Objekte besitzen.

Spielablauf

Der eigentliche Spielablauf kann nun in einer Methode *StarteSpiel()* abgebildet werden.

- Der Programmcode liest sich fast wie eine Spielanleitung.

```
1 public void StarteSpiel() {
2     feld = new Spielfeld();
3
4     do {
5         WechsleSpieler();
6         Console.WriteLine("Spieler " + aktueller_spieler.Spielstein + " ist an der Reihe:");
7         aktueller_spieler.Ziehe(feld);
8         feld.Ausgeben();
9
10        if (feld.HatGewonnen(aktueller_spieler.Spielstein)) {
11            Console.WriteLine("Spieler " + aktueller_spieler.Spielstein + " hat gewonnen!");
12            break;
13        }
14    }
15    while (feld.IstEinFeldFrei());
16
17    if (!feld.HatGewonnen() && !feld.IstEinFeldFrei())
18        Console.WriteLine("Unentschieden!");
19 }
```

TicTacToe spielen

Die Klasse TicTacToe kann nun dazu genutzt werden, das eigentliche Spiel zu starten:

```
1 TicTacToe ttt = new TicTacToe();
2 ttt.StarteSpiel();
```

```
06 -- zsh - 47x24
'O' 'O' ' '
Spieler 0 ist an der Reihe:
' ' 'X' 'O'
' ' 'X' 'X'
'O' 'O' ' '
Spieler X ist an der Reihe:
' ' 'X' 'O'
' ' 'X' 'X'
'O' 'O' 'X'
Spieler 0 ist an der Reihe:
'O' 'X' 'O'
' ' 'X' 'X'
'O' 'O' 'X'
Spieler X ist an der Reihe:
'O' 'X' 'O'
'X' 'X' 'X'
'O' 'O' 'X'
Spieler X hat gewonnen!
(base) stuckenholz@Alexanders-iMac 06 %
```

Erkenntnisse

Das Spiel wurde nun **modular** mithilfe objektorientierter Prinzipien realisiert.

- Die Spieler, das Spielfeld, und der Spielablauf wurden getrennt voneinander umgesetzt.
- Mithilfe von Objektbeziehungen können wir die Objekte zu einem **großen Ganzen** zusammensetzen.

Beim Entwurf der Klassen mussten wir jeweils überlegen, welche Aufgaben die Objekte **im Zusammenspiel** übernehmen müssen.

- Ist der Entwurf einigermaßen gelungen, können die Klassen dann unabhängig voneinander erweitert, getestet und ausgetauscht werden.
- Später können wir z.B. die aktuelle Spieler-Klasse durch eine andere ersetzen, ohne dass wir das Spielfeld usw. ändern müssen.

Der Schlüssel zum Erfolg ist dabei ein gutes Design der Klassen und der Zuschnitt der Aufgaben im Gesamtkontext.

6.4 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, dass...

- Objekte zur Laufzeit Beziehungen untereinander aufbauen können.
- solche Beziehungen auf Objektreferenzen basieren.

6 Objektbeziehungen

- man mit diesem Mechanismus komplexe Systeme aus vielen unterschiedlichen Objekten umsetzen kann.

Aufgabe 1

Setzen Sie das TicTacToe-Spiel komplett um, ohne sich den Programmcode im Quellcode-Repository anzusehen.

- Implementieren Sie alle Methoden der Klassen.

Ersetzen Sie die Klasse *ComputerSpieler* durch eine neue Variante.

- Diese Variante soll einen menschlichen Spieler realisieren.
- In jedem Zug wird nach der gewünschten Position für den Spielstein gefragt.
- Bitte beachten: Ein Spielstein kann nur auf eine freie Position gesetzt werden.

Aufgabe 2

Eine Mitfahrzentrale braucht eine neue Software, um Fahrten zu koordinieren.

- Das System soll dabei objektorientiert realisiert werden.

Erstellen Sie eine Klasse *Person*, mit entsprechenden Eigenschaften und Eigenschaftsmethoden.

- Erstellen Sie eine weitere Klasse *Fahrt*.
- Eine *Fahrt* hat einen Start- und einen Zielort (**string**).
- Eine *Fahrt* kennt einen Fahrer (Person) und mehrere Mitfahrer (Personen).
- Sorgen Sie dafür, dass nicht mehr als 3 Mitfahrer hinzugefügt werden können.

Nutzen Sie die Klassen, um mehrere Objekte der Klasse *Fahrt* zu erzeugen.

- Wie kann man dafür sorgen, dass ein Objekt der Klasse *Fahrt* nicht erzeugt werden kann, ohne einen Fahrer zu benennen?

7 Vererbung

7.1 Redundanzen im Programmcode

Hochschulverwaltung

Stellen wir uns vor, wir müssten eine Anwendung entwerfen, um eine Hochschule zu verwalten.

- Neben vielen Anderen Dingen finden wir an einer Hochschule auch Dozenten und Studierende.
- Damit wir solche Objekte erzeugen können, benötigen wir entsprechende Klassen.

Wir erstellen also die beiden Klassen *Student* und *Dozent*.

- Beide Klassen benötigen Eigenschaftsfunktionen für Vor- und Nachname.
- Der Student hat eine Matrikelnummer, der Dozent ein Lehrgebiet.

```
1 class Student
2 {
3     public string Vorname { get; set; }
4     public string Nachname { get; set; }
5     public int Matrikelnummer { get; set; }
6 }
```

```
1 class Dozent
2 {
3     public string Vorname { get; set; }
4     public string Nachname { get; set; }
5     public string Lehrgebiet { get; set; }
6 }
```

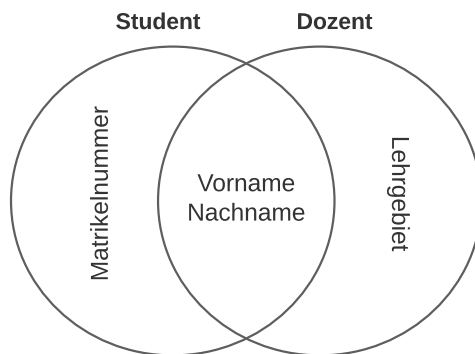
Gemeinsamkeiten

Wir stellen fest, dass Student und Dozent einigen Programmcode gemeinsam haben.

- Beide Klassen besitzen Eigenschaftsmethoden für Vor- und Nachnamen.

Dass Student und Dozent solche Gemeinsamkeiten aufweisen ist natürlich kein Zufall.

- Studenten und Dozenten haben ja gewisse Ähnlichkeiten.
- Beide sind menschliche Wesen!



Redundanzen

Redundanter Programmcode ist aus vielen Gründen sehr schlecht!

- Redundanter Programmcode ist schwer zu lesen und zu verstehen.
- Redundanter Programmcode ist schwer zu warten und zu pflegen!

Redundanzen im Programmcode sollten unbedingt vermieden werden!

- *Don't repeat yourself (DRY)* ist eine wichtige Basisregel, um qualitativen Programmcode zu erzeugen! (Siehe [7, S. 48])

Die Frage ist aber, wie wir hier diesen redundanten Programmcode vermeiden können?

- Um Ähnlichkeiten bei Klassen zusammenzuführen, bietet die Objektorientierung ein bestimmtes Konzept an.
- Die **Vererbung**.

Vererbung

Bei der Vererbung vererbt eine **Basisklasse** (auch Super-, Ober- oder Elternklasse) alle Eigenschaften an eine oder mehrere **Kindklassen**.

- Man spricht auch davon, dass man neue Klassen aus einer Basisklasse **ableitet**.

Durch die Vererbung besitzen die Kindklassen alle Eigenschaften der Basisklasse.

- Alle Objektvariablen, Eigenschaften und Methoden der Basisklasse.

Die Kindklassen können diese Elemente benutzen, ohne sie selbst definieren zu müssen.

- Vererbung hilft also dabei, Programmcode wiederverzuwenden und Redundanzen zu vermeiden, siehe auch [15, S. 68]

Die Kindklassen können zusätzliche Elemente definieren.

- Neue Variablen, Eigenschaften und Methoden.
- Die Kindklassen erweitern die Basisklassen dann um weitere Elemente.

7.2 Vererbung in C#

Person

Wir können nun eine gemeinsame Basisklasse *Person* schaffen.

- Dort können wir solche Elemente sammeln, die Student und Dozent gemeinsam haben.
- Wir verschieben also die beiden Eigenschaftsmethoden für Vor- und Nachname in diese Klasse.

```

1 public class Person
2 {
3     public string Vorname { get; set; }
4     public string Nachname { get; set; }
5 }

```

Student und Dozent

Die Klassen *Student* und *Dozent* können wir nun von der Klasse *Person* ableiten.

- Der Name der Basisklasse wird dazu hinter dem Doppelpunkt aufgeführt.
- Es wird also dieselbe Schreibweise genutzt, als würde die Klasse eine Schnittstelle implementieren.

```
1 public class Student : Person
2 {
3     public int Matrikelnummer { get; set; }
4 }
```

```
1 public class Dozent : Person
2 {
3     public string Lehrgebiet { get; set; }
4 }
```

Über die geerbten Elemente hinaus definieren beide Kindklassen noch zusätzliche Elemente.

- Zusätzlich zum Vor- und zum Nachnamen besitzt der Student noch eine Matrikelnummer.
- Der Dozent weist ein Lehrgebiet aus.

Klassen benutzen

Aus allen drei Klassen können nun Objekte erzeugt werden.

- Alle Objekte besitzen einen Vor- und einen Nachnamen.
- Nur Objekte der Klasse Student besitzen zusätzlich eine Matrikelnummer.
- Nur Objekte der Klasse Dozent besitzen zusätzlich ein Lehrgebiet.

```
1 var p = new Person() { Vorname = "Ingrid", Nachname = "Müller" };
2 var s = new Student() { Vorname = "Demir", Nachname = "Öztürk", Matrikelnummer = 12345 };
3 var d = new Dozent() { Vorname = "Ulrike", Nachname = "Demirel", Lehrgebiet = "Mathematik" };
```

Einfach- vs Mehrfachvererbung

In C# darf eine Klasse nur von maximal einer Basisklasse ableiten.

- Dies wird als **Einfachvererbung** bezeichnet.

Andere Programmiersprachen erlauben auch die Mehrfachvererbung, z.B. C++.

- Dabei kann es aber zu Konflikten kommen.
- Bei namensgleichen Methoden in zwei Basisklassen muss geklärt werden, welche Implementierung die Kindklasse nutzen soll.

In C# darf eine Klasse aber beliebig viele Schnittstellen implementieren.

- Selbst bei Namensgleichheit mehrere Methoden kann es dabei nicht zu Konflikten kommen.
- Eine Schnittstelle sorgt dafür, dass eine bestimmte Methode vorhanden sein muss.
- Wie diese aber implementiert wird, ist der Klasse selbst überlassen.

Spezialisierung und Generalisierung

Mithilfe der Vererbung können komplexe Hierarchien gebildet werden.

- In mehreren Ebenen können Klassen voneinander ableiten.
- Eine solche Vererbungshierarchie hat eine baumartige Struktur.

Je tiefer man in der Hierarchie absteigt, desto mehr Eigenschaften weisen die Klassen auf.

- In Vererbungsrichtung werden die Klassen dadurch spezieller (**Spezialisierung**).

Blickt man hingegen entgegen der Vererbungsrichtung, werden die Klassen immer allgemeiner.

- Klassen vereinen dann die Eigenschaften mehrerer Kindklassen.
- Sie verallgemeinern die Konzepte (**Generalisierung**).

7.3 Vererbungshierarchie und Typumwandlung

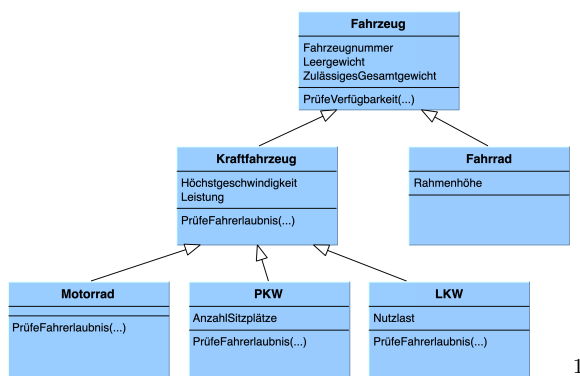
Vererbungshierarchie

Eine Vererbungshierarchie von mehreren Klassen kann auch grafisch dargestellt werden.

- Dazu wird meist ein UML-Diagramm genutzt (später mehr.)

Das folgende Beispiel zeigt ein solches UML-Diagramm.

- Die Klasse *Fahrzeug* ist die Basisklasse.
- Alle anderen Klassen sind abgeleitet.



1

Erkenntnisse aus dem Beispiel

Aus der Vererbungshierarchie des Beispiels können einige Sachverhalte abgeleitet werden:

- Ein Kraftfahrzeug ist ein Fahrzeug.
- Ein Motorrad ist ein Kraftfahrzeug.
- Ein Fahrzeug ist jedoch kein LKW.

Die Basisklasse *Fahrzeug* stellt generelle Eigenschaften bereit.

- Diese Eigenschaften werden von allen Klassen in der Hierarchie geteilt.
- Alle Klassen besitzen die Attribute Fahrzeugnummer, Leergewicht, ...
- Alle Klassen besitzen die Elementfunktion *PrüfeVerfügbarkeit()*.

¹Bildquelle: Cactus26 (<https://commons.wikimedia.org/wiki/File:InheritancePgmExample.svg>), „InheritancePgmExample“, <https://creativecommons.org/licenses/by-sa/3.0/legalcode>

Die Klasse *Kraftfahrzeug* spezialisiert die Klasse *Fahrzeug* z.B. um die Eigenschaft Leistung.

- Ein Kraftfahrzeug ist demnach ein spezielles Fahrzeug.

Typumwandlung in Vererbungshierarchien

Objekte einer Kindklasse bieten alle Methoden an, die auch die Basisklasse besitzt.

- Objekte der Kindklasse lassen sich also genauso benutzen, wie Objekte der Basisklasse.
- Ein Objekt einer Kindklasse kann daher in den Typ der Basisklasse umgewandelt werden (engl. *up-casting*).

```
1 var s = new Student() { Vorname="Maria", Nachname="Eitekin", Matrikelnummer=12345 };
2 var p = (Person)s;
```

Ein Objekt der Basisklasse kann aber nicht in ein Objekt der Kindklasse umgewandelt werden.

- Eine fehlende Matrikelnummer einer Person kann ja nicht hinzuerfunden werden.
- Folgender Versuch einer Typumwandlung wird daher vom Compiler mit einer Fehlermeldung verweigert.

```
1 Person p = new Person() { "Elrike", "Üzgür" };
2 Student s = (Student) p;
```

7.4 Konstruktoren und Vererbung

Konstruktoren und Vererbung

Durch Vererbung werden Variablen und Methoden an die Kindklassen weiter gegeben.

- Konstruktoren und der Destruktor werden allerdings nicht vererbt.
- Ansonsten könnte die korrekte Initialisierung von Objekten von Kindklassen nicht sicher gestellt werden.

Die Klasse *Person* kann um einen Konstruktor erweitert werden:

7 Vererbung

```
1 public class Person
2 {
3     public string Vorname { get; set; }
4     public string Nachname { get; set; }
5
6     public Person(string vorname, string nachname)
7     {
8         Vorname = vorname;
9         Nachname = nachname;
10    }
11 }
```

Konstruktor der Basisklasse verwenden

Da die Basisklasse *Person* nun einen Konstruktor besitzt, müssen auch die Kindklassen einen Konstruktor definieren.

- Ansonsten ist das Projekt nicht übersetzbar.

Man kann aber einen Teil der Arbeit an den Konstruktor der Basisklasse delegieren.

- Dazu wird das Schlüsselwort `base` benutzt, der auf das Objekt der Basisklasse verweist.

```
1 public class Student : Person
2 {
3     public int Matrikelnummer { get; set; }
4
5     public Student(string vorname, string nachname, int matrikelnummer) : base(vorname, nachname)
6     {
7         Matrikelnummer = matrikelnummer;
8     }
9 }
```

Adresse

Wir wollen eine neue Klasse einführen, um die Wohnadresse einer Person abbilden zu können.

```
1 class Adresse
2 {
3     public string Strasse { get; set; }
4     public int PLZ { get; set; }
5     public string Ort { get; set; }
6
7     public Adresse(string strasse, int plz, string ort)
```



```

8      {
9          this.Strasse = strasse;
10         this.PLZ = plz;
11         this.Ort = ort;
12     }
13 }

```

Von dieser Klasse können wir natürlich erneut leicht Objekte erzeugen.

```

1  var w = new Adresse("Im Rosenhang 12", 59063, "Hamm");

```

7.5 Objektbeziehungen und Vererbung

Wohnadresse

Objekte vom Typ *Adresse* sind für sich allein allerdings nicht besonders hilfreich.

- Wir müssen in der Lage sein, ein solches Objekt mit Personen in Verbindung zu setzen.
- Dies können wir mithilfe einer Objektbeziehung erreichen.
- Die Klasse *Person* erweitern wir dazu um eine Eigenschaftsmethode Wohnadresse.
- Diese nimmt eine Referenz auf ein Objekt vom Typ *Adresse* auf.

```

1  class Person
2  {
3      // Der Rest fehlt hier...
4      public Adresse Wohnadresse { get; set; }
5  }

```

Einem Objekt der Klasse *Person* können wir nun zur Laufzeit eine *Wohnadresse* zuweisen:

```

1  var p = new Person("Ingrid", "Müller");
2  p.Wohnadresse = new Adresse("Im Rosenhang 12", 59063, "Hamm");

```

Objektbeziehungen und Vererbung

Die Klassen *Student* und *Dozent* sind von der Klasse *Person* abgeleitet.

- Beide Klassen erben also alle Variablen, Eigenschaften und Methoden der Basis-klasse.
- Dazu gehört auch die Eigenschaftsmethode *Wohnadresse*.
- Entsprechend können wir auch Objekten der Klassen *Student* und *Dozent* nun eine Wohnadresse zuweisen.

```
1 var s = new Student("Demir", "Öztürk", 12345);
2 s.Wohnadresse = new Adresse("Schlehenstraße 26", 59063, "Hamm");
3
4 var d = new Dozent("Elvira", "Kosmolowski", "Physik");
5 d.Wohnadresse = new Adresse("Marker Allee 123", 59063, "Hamm");
```

Bei der Vererbung wird auch die Möglichkeit für Objektbeziehungen weiter vererbt.

- Erlaubt die Basisklasse bestimmte Objektbeziehungen, können auch die Kindklassen diese Objektbeziehungen aufbauen.

Private Objektvariablen und Vererbung

Nehmen wir an, die Klasse *Person* würde noch weitere Objektvariablen deklarieren, z.B. die Schuhgröße.

```
1 class Person
2 {
3     // Der Rest fehlt hier
4
5     private double schuhgroesse;
6 }
```

Obwohl die Klassen *Student* und *Dozent* Kindklassen sind und alle Variablen erben, dürfen wir dort aber nicht auf die Schuhgröße zugreifen.

- Der Compiler würde die Übersetzung des Projekts mit einem Fehler abbrechen, würden wir dies versuchen.
- Wenn Variablen privat sind, dürfen selbst Objekte von Kindklassen diese nicht nutzen.

Schlüsselwort `protected`

Neben den beiden Zugriffsmodifizierern `private` und `public` wurde speziell für die Vererbung noch eine dritte Variante eingeführt: `protected`

- Der Zugriffsmodifizierer `protected` bewirkt, dass die abgeleiteten Klassen Zugriff auf die sonst geschützten Elemente erhalten.

Der Zugriffsmodifizierer `protected` ist also eine Kombination aus `private` und `public`.

- Ein als `protected` markiertes Element ist `public` für alle Objekte der Kindklassen.
- Es ist aber `private` für alle anderen Objekte außerhalb der Vererbungshierarchie.

Ändern wir den Zugriffsmodifizierer der Schuhgröße in der Klasse *Person* von `private` auf `protected`, können die Objekte der Kindklassen diese Objektvariable nutzen.

```
1 protected double schuhgroesse;
```

7.6 Methoden und Vererbung**Methoden vererben**

Wir wollen nun in der Lage sein, Objekte der Klassen *Person*, *Student* und *Dozent* auf der Konsole auszugeben.

- Wir können dazu eine Methode *Ausgeben()* in die Klasse *Person* einbauen.

```
1 class Person
2 {
3     // Der Rest fehlt hier
4
5     public void Ausgeben()
6     {
7         Console.WriteLine(Vorname + " " + Nachname);
8     }
9 }
```

Durch die Vererbung ist die Methode dann auf auch auf allen Kindklassen verfügbar.

```
1 var s = new Student("Demir", "Öztürk", 12345);
2 s.Ausgeben();
```

Methoden verbergen und überschreiben

Die Methode *Ausgeben()* wird von der Klasse *Person* an alle Kindklassen weiter vererbt.

- Im Gegensatz zu einer *Person* besitzt der *Student* aber zusätzlich eine Matrikelnummer.
- Der *Dozent* zudem ein Lehrgebiet.
- Wenn wir *Ausgeben()* auf einem *Studenten* oder *Dozenten* aufrufen, sollten diese Infos auch mit ausgegeben werden.

In einer Kindklasse von *Person* können wir eine neue Methode *Ausgeben()* hinzufügen.

- Wir müssen dann aber festlegen, ob die geerbte Methode durch die neue Methode **verborg**en oder **überschrie**ben werden soll.
- Die geerbte Methode wird mit dem Schlüsselwort **new** verborgen.
- Die geerbte Methode wird mit dem Schlüsselwort **override** überschrieben.

Bei beiden Varianten gibt es einen kleinen, aber feinen Unterschied.

Methode verbergen

In der Klasse *Student* fügen wir eine eigene Methode *Ausgeben()* hinzu.

- Mit dem Schlüsselwort **new** verbergen wir die geerbte Methode.

```
1 public new void Ausgeben()  
2 {  
3     Console.WriteLine("{0} {1} - {2}", Vorname, Nachname, Matrikelnummer);  
4 }
```

Weiterhin kann auf den Objekten aller Klassen die Methode *Ausgeben()* aufgerufen werden.

- Beim *Studenten* wird nun die neue Methode genutzt.
- Diese gibt auch die Matrikelnummer aus.
- Konvertieren wir den *Studenten* in ein Objekt vom Typ *Person*, wird wieder nur der Vor- und Nachname ausgegeben.

```

1 var s = new Student("Demir", "Öztürk", 12345);
2 s.Ausgeben(); // gibt die Matrikelnummer mit aus
3 var p = (Person) s;
4 p.Ausgeben(); // gibt nur Vor- und Nachname aus

```

Methode überschreiben

Anstelle die geerbte Methode zu verbergen, können wir sie auch überschreiben.

- Dazu muss aber die Methode *Ausgeben()* in der Klasse *Person* zunächst mit dem Schlüsselwort **virtual** gekennzeichnet werden.

```

1 public virtual void Ausgeben()
2 {
3     Console.WriteLine(Vorname + " " + Nachname);
4 }

```

Erneut fügen wir der Klasse *Student* eine neue Methode *Ausgeben()* hinzu.

- Dieses mal nutzen wir aber das Schlüsselwort **override**.

```

1 public override void Ausgeben()
2 {
3     Console.WriteLine("{0} {1} - {2}", Vorname, Nachname, Matrikelnummer);
4 }

```

Unterschied zwischen Verbergen und Überschreiben

Das Schlüsselwort **virtual** aktiviert das sog. **Dynamische Binden**.

- Dadurch wird erst zur Laufzeit die Methode ermittelt, die auf einem Objekt aufgerufen werden soll.

Im Rahmen der Typkonvertierung können wir den Unterschied sehen.

- Erneut konvertieren wir ein Objekt der Klasse *Student* in den Typ seiner Basis-klasse.
- Rufen wir nun die Methode *Ausgeben()* auf, wird noch immer die Methode der Ursprungs-klasse genutzt.
- Das Objekt hat sich quasi *gemerkt*, welcher Typ es ursprünglich war.

```
1 var s = new Student("Demir", "Öztürk", 12345);
2 s.Ausgeben();    // gibt die Matrikelnummer mit aus
3 var p = (Person) s;
4 p.Ausgeben();    // gibt noch immer die Matrikelnummer mit aus
```

7.7 Double Dispatch Problem

Das Double Dispatch Problem

Dynamisches Binden wählt die passende Methode auf einem Objekt erst zur Laufzeit aus.

- Bei überladenen Methoden gibt es (zumindest in C#) diese Dynamik aber nicht.

Im folgenden Beispiel existieren zwei überladene Methoden.

- Eine Methode erwartet ein Objekt vom Typ *Person*, eine andere ein Objekt vom Typ *Student*.

```
1 public static void TueEtwas(Person a) { Console.WriteLine("Person"); }
2 public static void TueEtwas(Student a) { Console.WriteLine("Student"); }
3
4 public static void Main(string[] args)
5 {
6     Person obj = new Student("Alexander", "Stuckenholz");
7     TueEtwas(obj);
8 }
```

Das Double Dispatch Problem

Wir stellen fest, dass der Text «Person» ausgegeben wird.

- Wir würden aber erwarten, dass «Student» ausgegeben wird.
- Das Objekt wurde ja als Student erzeugt und dann nachträglich in eine Person umgewandelt.

Dies liegt daran, dass beim Überladen die passende Methode schon zur Übersetzungszeit festgelegt wird.

- Der Compiler sieht beim Übersetzen aber nur einen Parameter vom Typ *Person*.

Wird zur Laufzeit ein abgeleiteter Typ übergeben, kann dies nicht mehr festgestellt werden.

- Dieser Effekt ist als das **Double Dispatch Problem** bekannt.
- Andere Programmiersprachen, z.B. Lisp, zeigen hier ein anderes Verhalten.

7.8 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, ...

- aus welchen Gründen Redundanzen im Programmcode schlecht sind.
- wie man mit Vererbung bestimmte Redundanzen vermeiden kann.
- was mit dem Konzept der Vererbung in der objektorientierten Programmierung genau gemeint ist.
- was es bedeutet, wenn eine Kindklasse von einer Basisklasse abgeleitet ist.
- wie man Vererbung in C# umsetzt.
- was Vererbung mit Konstruktoren macht.
- wie sich Vererbung auf Objektbeziehungen auswirkt.
- welche Bedeutung der Zugriffsmodifizierer `protected` besitzt.
- wie man geerbte Methoden verbergen und erweitern kann.
- was Mehrfachvererbung ist und warum sie in C# nicht erlaubt ist.

Aufgabe 1

Pac Man ist ein sehr bekanntes Computerspiel.

- Auf einem Spielfeld sind verschiedene Elemente sichtbar, die untereinander eine Vererbungsbeziehung aufbauen.

Als Basisklasse existiert die Klasse Spielelement.

- Ein Spielelement hat eine X-/Y-Position.
- Wand und Pille sind jeweils Spielelemente.
- Zudem existiert die Klasse Figur, die ein Spielelement ist.
- Eine Figur besitzt die Methode `bewege()`.
- Ein PacMan und ein Geist sind Figuren.

7 Vererbung

Stellen Sie diese Zusammenhänge grafisch dar.



Aufgabe 2

Erstellen Sie die Klassen Spielfeld, Spielelement, Wand, Pille, Figur, PacMan und Geist in C#.

- Das Spielfeld *kennt* eine Menge von Wänden, eine Menge von Pillen, genau einen PacMan und genau vier Geister.
- Wie kann das in C# realisiert werden?

Überschreiben Sie die Methode Bewege() in den Klassen PacMan und Geist.

- Geben Sie jeweils eine Zeichenkette „PacMan bewegt sich“ bzw. „Geist bewegt sich“ auf der Konsole aus.

Welche Methoden werden in welchen Klassen noch gebraucht?

- Ist es noch schwierig, nach dieser Aufteilung das Spiel komplett fertig zu implementieren?

8 Polymorphie

8.1 Wurzelklasse Object und das base-Schlüsselwort

Wurzelklasse Object

Leitet in C# eine Klasse nicht explizit von einer anderen ab, wird sie implizit von der Basisklasse *Object* abgeleitet.

- Die Klasse *Object* wird durch das .Net-Framework selbst definiert.
- Die Klasse *Object* ist somit die ultimative Wurzelklasse in C#.

Da alle Klassen implizit von *Object* ableiten, können auch alle Objekte in den Basistyp *Object* konvertiert werden.

```
1 var s = new Student("Hannelore", "Klimcak", 12345);  
2 var o = (Object) s;
```

Auf *o* können dann aber nur Methoden aufgerufen werden, die durch die Klasse *Object* auch definiert wurden.

Methoden der Klasse Object

Die Klasse *Object* bietet einige Low-Level-Dienste an.

- Jedes Objekt in C# verfügt über diese Methoden:
- *GetType*: Gibt Laufzeitinformationen zum Objekt zurück.
- *Equals*: Überprüft, ob zwei Objekte verweisgleich sind.
- *GetHashCode*: Erzeugt einen Hash-Code eines Objektes.
- *ToString*: Erzeugt eine String-Repräsentation des Objektes.

Alle diese Methoden können in den Kindklassen auch überschrieben werden.

- Insbesondere die Methode *ToString()* ist dafür prädestiniert.

8 Polymorphie

- Wann immer ein Objekt Text umgewandelt werden soll, wird diese Methode benutzt.

Das base-Schlüsselwort

Auch wenn in einer Kindklasse eine Methode verborgen oder überschrieben wurde, kann in dem Objekt dennoch auf die geerbte Methode zugegriffen werden.

- Hierfür kann das Schlüsselwort `base` benutzt werden.

Ähnlich zu `this` stellt `base` einen Objektverweis dar.

- Im Objekt einer Kindklasse verweist `base` auf das *Basisobjekt*.
- In einem Student-Objekt können wir so auch die Ausgabe-Routine der Person aufrufen.

```
1 public class Student : Person
2 {
3     // Der Rest fehlt hier...
4
5     public new void Ausgeben()
6     {
7         base.ausgeben();
8         Console.WriteLine("Matrikelnummer: {0}", Matrikelnummer);
9     }
10 }
```

8.2 Abstrakte Klassen

Abstrakte Klassen

Klassen können Schnittstellen implementieren.

- Dabei definieren Schnittstellen lediglich Methodensignaturen, also einen Vertrag.
- Die Klasse muss die Schnittstelle dann aber mit einer eigenen Implementierung ausfüllen.

Die Vererbung ist hier noch stärker.

- Erbt eine Klasse von einer Basisklasse, so wird nicht nur ein Vertrag an die Kindklasse weitergegeben.
- Auch die Implementierung der Methoden wird vererbt.

Abstrakte Klassen vermischen diese beiden Prinzipien.

- Abstrakte Klassen können für einige Methoden lediglich die Signaturen definieren (wie eine Schnittstelle).
- Sie können für andere Methoden aber auch die Implementierung mitliefern.

Grafikprogramm

Stellen wir uns vor, wir wollten ein Grafikprogramm schreiben.

- Darin wollen wir einige Elemente, wie z.B. Kreise, Rechtecke, Sterne usw. anbieten.

Alle diese Elemente haben gewisse Gemeinsamkeiten.

- Jedes dieser Elemente besitzt z.B. eine Bildschirmposition in Form von X/Y-Koordinaten.
- Entsprechend können wir eine Basisklasse einführen: *GrafischesElement*.

Jedes Element muss auch später auf dem Bildschirm gezeichnet werden.

- Wie die einzelnen Elemente aber gezeichnet werden müssen, unterscheidet sich aber.
- Ein Dreieck sieht ganz anders aus, als ein Stern.
- Daher kann die Basisklasse nur vorgeben, dass eine Methode Zeichne() geben soll.
- Nicht aber, wie diese Methode implementiert werden soll.
- Die Methoden Zeichne() besitzt in der Klasse *GrafischesElement* daher nur eine Signatur.

Abstrakte Klasse in C#

Eine Klasse, die für mindestens eine Methode nur eine Signatur besitzt, ist abstrakt.

- Sowohl die Klasse selbst, als auch alle nicht implementierten Methoden sind mit dem Schlüsselwort **abstract** markiert.

```

1  abstract class GrafischesElement
2  {
3      public int X { get; set; }
4      public int Y { get; set; }
5
6      public void Bewege(int delta_x, int delta_y)
7      {
8          this.X += delta_x;

```

8 Polymorphie

```
9         this.Y += delta_y;
10    }
11
12    public abstract void Zeichne();
13 }
```

Abstrakte Klassen benutzen

Von einer abstrakten Klasse können keine Objekte erzeugt werden.

- Eine abstrakte Klasse dient ausschließlich dazu, gemeinsame Funktionalität zu sammeln.
- Über die Vererbung wird diese Funktionalität dann an die Kindklassen weiter gegeben.
- Abstrakte Klassen dienen als Basisklasse in Vererbungshierarchien.

Abstrakte Methoden müssen in einer Kindklasse implementiert werden.

- Die zu implementierenden Methoden müssen dazu überschrieben werden.
- Dazu wird erneut das Schlüsselwort `override` benutzt.

```
1 class Dreieck : GrafischesElement
2 {
3     public override void Zeichne()
4     {
5         Console.WriteLine("Das Dreieck wird gezeichnet");
6     }
7 }
```

Typkonvertierung und abstrakte Klassen

Wir können nun natürlich Objekte der Klasse Dreieck erzeugen.

- Rufen wir die Methode Zeichne() auf, wird der Text »*Das Dreieck wird gezeichnet*« auf der Konsole ausgegeben.

```
1 Dreieck d = new Dreieck() { X=10, Y=15 };
2 d.Zeichne();
```

In einer Vererbungshierarchie können wir Objekte in den Typ der Basisklasse konvertieren.

- Ein *Dreieck* können wir demnach in ein *GrafischesElement* umwandeln.

- Auf diesem Objekt können wir dann auch die Methode `Zeichne()` aufrufen.
- Es wird dann die Implementierung der Dreieck-Klasse benutzt.

```

1 GrafischesElement g = d;
2 g.Zeichne();

```

8.3 Polymorphie

Polymorphie

Dieses Verhalten haben wir bereits im letzten Abschnitt beobachtet und ist dennoch bemerkenswert.

- Ein Objekt sieht aus wie ein *GrafischesElement*, verhält sich zur Laufzeit aber wie ein *Dreieck*.

Dieses Verhalten wird auch als **polymorphes Verhalten** bzw. **Polymorphie** bezeichnet.

- Das Wort Polymorphie kommt aus dem Griechischen und bedeutet Vielgestaltigkeit.
- Die Polymorphie ist ein sehr wichtiges Konzept der Objektorientierung.

Durch die Polymorphie können wir überall dort, wo ein *GrafischesElement* benötigt wird, auch ein *Dreieck* verwenden.

- Die Basisklasse *GrafischesElement* definiert im übertragenen Sinn die Steckdose.
- Zur Laufzeit können in diese Steckdose unterschiedliche Geräte (Objekte) eingesteckt werden, z.B. ein Dreieck.

Tic Tac Toe

Wir wollen an einem Beispiel sehen, wie die Polymorphie genutzt werden kann.

- In einem vorhergehenden Abschnitt haben wir uns mit dem Spiel Tic Tac Toe befasst.
- Wir wollen das Spiel nun so erweitern, dass wir das Spiel mit unterschiedlichen Arten von Spielern spielen können (menschlicher Spieler oder Computerspieler)

Bislang haben wir lediglich eine Klasse *ComputerSpieler* erstellt.

- Diese Klasse setzt einen automatisierten Spieler um.

- Der Spielstein wird jeweils zufällig auf ein freies Feld abgelegt.
- Die Klasse besitzt dazu die Methode *Ziehe()*.

Wir wollen nun auch eine Klasse *MenschlicherSpieler* umsetzen.

- Ein solcher Spieler soll nach Koordinaten für den nächsten Zug fragen.
- Wenn der Platz frei ist, wird der Spielstein dort abgelegt.

8.4 Polymorphie mit abstrakter Basisklasse

Basisklasse für Spieler

Die Klassen *ComputerSpieler* und *MenschlicherSpieler* haben Gemeinsamkeiten.

- Objekte beider Klassen verwalten ihren Spielstein in einer Eigenschaft.
- Objekte beider Klassen benötigen auch eine Methode *Ziehe()*, um ihren Zug umzusetzen.

Weisen Klassen solche Gemeinsamkeiten auf, ist es sinnvoll, eine Basisklasse zu schaffen.

- Die Methode *Ziehe()* ist abstrakt und muss durch jede Kindklasse selbst implementiert werden.

```
1  abstract class Spieler
2  {
3      public char Spielstein { get; set; }
4
5      public Spieler(char spielstein)
6      {
7          Spielstein = spielstein;
8      }
9
10     public abstract void Ziehe(Spielfeld feld);
11 }
```

ComputerSpieler

Die Klasse *ComputerSpieler* können wir nun so ändern, dass sie von der Basisklasse *Spieler* ableitet.

```

1  class ComputerSpieler : Spieler
2  {
3      private Random rnd = new Random();
4
5      public ComputerSpieler(char spielstein) : base(spielstein)
6      {
7      }
8
9      public override void Ziehe(Spielfeld feld)
10     {
11         // Der Programmcode hier ist noch der selbe, wie zuvor
12     }
13 }
```

MenschlicherSpieler

Auch die Klasse *MenschlicherSpieler* lässt sich nun leicht erstellen.

- Die Klasse leiten wir ebenso von der Basisklasse *Spieler* ab.
- In der Methode *Ziehe()* müssen wir lediglich den Nutzer nach Koordinaten fragen.

```

1  class MenschlicherSpieler : Spieler
2  {
3      public MenschlicherSpieler(char spielstein) : base(spielstein)
4      {
5      }
6
7      public override void Ziehe(Spielfeld feld)
8      {
9          // Nutzer nach gültigen und freien X/Y-Koordinaten fragen
10
11         feld.Setzen(x, y, Spielstein);
12     }
13 }
```

TicTacToe

In der Klasse *TicTacToe* nutzen wir bislang zwei Objekte der Klasse *ComputerSpieler*.

- Hier können wir nun stattdessen zwei Objekte vom Typ *Spieler* benutzen.

```
1 class TicTacToe
2 {
3     private Spielfeld feld;
4     private Spieler aktueller_spieler;
5
6     public Spieler Spieler1 { get; set; }
7     public Spieler Spieler2 { get; set; }
8
9     // Der Rest bleibt gleich
10 }
```

Spieler1 und *Spieler2* müssen vor Beginn des Spiels Objekte zugewiesen bekommen.

- Da wir aus der abstrakten Klasse *Spieler* keine Objekte erzeugen können, können dies nur Objekte der Klassen *MenschlicherSpieler* oder *ComputerSpieler* sein.

Spiel starten

Dies ermöglicht uns, zur Laufzeit dynamisch festzulegen, wer gegen wen spielen soll.

- Zwei Computerspieler gegeneinander, ein Computerspieler gegen einen Menschen, ...
- Wir konfigurieren die Zusammenstellung des Spiels durch unterschiedliche Objekte, z.B.:

```
1 TicTacToe ttt = new TicTacToe();
2 ttt.Spieler1 = new ComputerSpieler('X');
3 ttt.Spieler2 = new MenschlicherSpieler('O');
4 ttt.StarteSpiel();
```

Jenachdem, welche Art von Objekten wir den Eigenschaften *Spieler1* und *Spieler2* zuweisen, verhält sich das Spiel dann anders.

- Das Spiel weiß nicht, um welche Art von Spieler es sich jeweils konkret handelt.
- Es ist auch nicht wichtig, da sichergestellt ist, dass jedes Objekt eine eigene Implementierung von *Ziehe()* besitzt.

Erkenntnisse

Im TicTacToe-Spiel haben wir die Polymorphie bewusst eingesetzt.

- Es ergibt sich dadurch die Möglichkeit, erst zur Laufzeit die konkrete Art des Spielers festlegen zu müssen.

- Die beiden Eigenschaften *Spieler1* und *Spieler2* stellen die Steckdosen dar, in die wir konkrete Objekte der Klassen *MenschlicherSpieler* oder *ComputerSpieler* einstecken können.

Technisch wird die Steckdose durch die abstrakte Klasse *Spieler* realisiert.

- In den abgeleiteten Klassen überschreiben wir die Methode *Ziehe()*.
- Dazu wird das Schlüsselwort `override` genutzt.

Abstrakte Klassen sind ein Weg, um polymorphes Verhalten zu erhalten.

- Denselben Effekt können wir auch mithilfe von Schnittstellen erzeugen.

8.5 Polymorphie mit Schnittstelle

Polymorphie mit Schnittstelle

Wir wollen nun polymorphes Verhalten mithilfe einer Schnittstelle erzeugen.

- Wir führen dazu die Schnittstelle *ISpieler* ein.

Sie definiert, welche Methoden eine Klasse anbieten muss, um als Spieler zu gelten.

- Wie bereits zuvor benötigen wir eine Eigenschaft, um den Spielstein zu speichern.
- Zudem wird auch die Methode *Ziehe()* gebraucht.

```

1 interface ISpieler
2 {
3     char Spielstein { get; set; }
4     void Ziehe(Spielfeld feld);
5 }

```

Schnittstelle Implementieren

Anstelle von einer Basisklasse abzuleiten, implementieren die beiden Klassen *MenschlicherSpieler* und *ComputerSpieler* nun die Schnittstelle *ISpieler*.

- Es ist dann aber etwas mehr zu tun, weil z.B. die Eigenschaftsmethode *Spielstein* in den Klassen selbst implementiert werden muss.

```
1 class MenschlicherSpieler : ISpieler
2 {
3     public char Spielstein { get; set; }
4
5     public MenschlicherSpieler(char spielstein)
6     {
7         Spielstein = spielstein;
8     }
9
10    // Der Rest bleibt gleich
11 }
```

Klasse TicTacToe anpassen

In der Klasse *TicTacToe* müssen die Eigenschaften *Spieler1* und *Spieler2* angepasst werden.

- Der Datentyp der beiden Eigenschaften ist dann *ISpieler*.
- Den Eigenschaften können all jene Objekte zugewiesen werden, deren Klasse die Schnittstelle *ISpieler* implementiert.
- Dies ermöglicht erneut die dynamische Auswahl von passenden Objekten.

```
1 class TicTacToe
2 {
3     private Spielfeld feld;
4     private ISpieler aktueller_spieler;
5
6     public ISpieler Spieler1 { get; set; }
7     public ISpieler Spieler2 { get; set; }
8
9     // Der Rest bleibt gleich...
10 }
```

8.6 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben gelernt, ...

- dass in C# die Klasse *Object* die ultimative Basisklasse ist.
- welche Methoden daher alle Objekte in C# besitzen.
- was es bedeutet, wenn man von Polymorphie spricht.

- wie man polymorphes Verhalten mithilfe Vererbung herstellen kann.
- wie man polymorphes Verhalten mithilfe von Schnittstellen herstellen kann.

Aufgabe 1

Erstellen Sie eine Klasse *Grafikprogramm*.

- Die Klasse soll eine Menge von Objekten des Typs *GrafischesElement* verwalten können.
- Nutzen Sie dazu eine Objektvariable vom Typ *List*.
- Erstellen Sie eine Methode, um der Liste ein Objekt hinzufügen zu können.

Implementieren Sie die Klassen *Rechteck* und *Kreis* als Kindklassen der Klasse *GrafischesElement*.

Aufgabe 2

Das Grafikprogramm soll die grafischen Elemente als Bilddatei speichern können.

- Bekanntlich existieren unterschiedliche Bildformate, z.B. Jpeg oder Png.
- Das Grafikprogramm soll zur Laufzeit genau ein solches Format kennen.
- Erstellen Sie zwei Dummy-Klassen für die Speicherung in einem Format, z.B. die Klassen *JpegFormat* und *PngFormat*.
- Beide Klassen sollen über die Methode *Speichern()* verfügen.

Sorgen Sie mithilfe von Polymorphie dafür, dass das Grafikprogramm dynamisch zur Laufzeit eines der beiden Formate zur Speicherung nutzen kann.

- Nutzen Sie entweder eine abstrakte Basisklasse oder eine Schnittstelle.

9 Analyse

9.1 Softwareentwicklung und Softwaretechnik

Softwaredesign

Bislang haben wir uns ausschließlich mit der Programmierung beschäftigt.

- Bevor wir aber Programmieren können, müssen einige Vorbedingungen erfüllt sein.
- Wir benötigen eine Vorstellung davon, was in welcher Weise umgesetzt werden soll.

Das Ergebnis dieser Vorüberlegungen ist das **Design** bzw. der **Entwurf**.

- Das Design beschreibt, aus welchen Teilen die Software im Inneren besteht und wie diese Teile zusammenwirken.

Das Softwaredesign ist immer vorhanden!

- Es entsteht auch dann, wenn man sich darüber keine Gedanken macht.

Die Alternative zu gutem Design ist schlechtes Design, nicht überhaupt kein Design ([1]).

- Softwaresysteme, bei denen das Design rein zufällig entstanden ist, werden auch als *big ball of mud* bezeichnet.

Softwareentwicklung und Softwaretechnik

Echte **Softwareentwicklung** ist weit mehr, als nur Programmieren!

- Das Design des Softwaresystems sollte vor der Programmierung explizit festgelegt werden!
- Dies ist aber nicht der einzige wichtige Schritt zu qualitativ hochwertiger Software.

Wie man in einem geordneten Prozess von den Anforderungen bis zum lauffähigen Softwaresystem gelangen kann, wird durch die Disziplin der **Softwaretechnik** (*engl. software engineering*) beantwortet.

Die Softwaretechnik hat das Ziel, Prinzipien, Methoden und Werkzeuge bereitzustellen und zielorientiert anzuwenden, so dass eine arbeitsteilige Entwicklung von umfangreichen Softwaresystemen möglich wird, siehe [8], [10], [9].

Kernprozesse der Softwaretechnik

Die Softwareentwicklung umfasst fünf elementare Kernprozesse:

1. **Planung:** Es werden Anforderungen erhoben (Lasten- bzw. Pflichtenheft).
2. **Analyse:** Es wird ein Informationsmodell erstellt und die elementare Verarbeitungslogik definiert.
3. **Entwurf:** Die sog. Softwarearchitektur wird festgelegt.
4. **Programmierung:** Der Programmcode wird umgesetzt.
5. **Qualitätssicherung:** Die Qualität der Umsetzung wird verifiziert und validiert.

Vorgehensmodelle

Wie und in welcher Reihenfolge diese Kernprozesse der Softwareentwicklung abgearbeitet werden, legt ein **Vorgehensmodell** fest.

- Das einfachste dieser Vorgehensmodelle wird als **Wasserfallmodell** bezeichnet.
- Darin wird jeder Prozess erst vollständig abgearbeitet, bevor zum nächsten Prozess fortgeschritten werden kann.

In der Praxis hat sich aber gezeigt, dass es nicht sinnvoll ist, z.B. erst alle Anforderungen komplett aufzuschreiben, bevor die Entwicklung beginnen kann.

- Es haben sich eher agile Verfahrensmodelle etabliert, z.B. Scrum oder Kanban.

Darin werden für kleinere Produktteile alle Prozesse bearbeitet.

- Dies sorgt dafür, dass schneller entsprechende Ergebnisse sichtbar werden.

Planung

Was genau umgesetzt werden soll, wird in der **Planungsphase** festgelegt.

- Das Ergebnis sind genaue funktionale und nicht funktionale Anforderung an das Produkt.

Funktionale Anforderungen beschreiben, was ein Produkt leisten soll.

- Nicht funktionale Anforderungen beschreiben Qualitätsmerkmale, z.B. wie schnell eine Reaktion erfolgen muss usw.

Die Anforderungen können z.B. in Form von Anwendungsfällen (*engl. use cases*), User Stories oder Prozessabläufen dargestellt werden.

- Sie werden in Lasten-, Pflichtenheften oder Backlogs gesammelt.

Anforderungen so zu erheben und zu verwalten, dass sie eine gute Vorstellung vom finalen Produkt liefern, ist eine Kunst für sich (*engl. requirements engineering*).

- Damit wollen wir uns hier aber nicht weiter befassen.

9.2 Analyse und Modellierung mit der UML

Analysephase und UML

An die Planungsphase schließt sich die **Analysephase** an.

- Aus den Anforderungen wird hier ein erstes Informationsmodell bzw. ein Modell der elementaren Verarbeitungslogik abgeleitet.
- Ein solches Modell ist dabei eine erste Vorstellung, eine Idee, ein grober Plan.

Modelle, z.B. der Bauplan von einem Gebäude, werden meist grafisch dargestellt.

- Auch in der Softwareentwicklung hat sich die grafische Modellierung durchgesetzt.
- Für die Notation dieser grafischen Darstellung haben sich gewisse Standards etabliert.

Die **Unified Modeling Language (UML)** ist eine normierte Modellierungssprache, siehe [11].

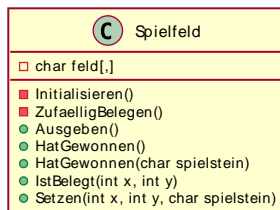
- Die UML definiert unterschiedliche Diagrammarten, um die Struktur oder das Verhalten eines Systems zu beschreiben.

9.3 Klassendiagramm

Klassen in der UML

Das Klassendiagramm ist sicherlich das wichtigste Diagramm der UML, um die Struktur eines Systems zu beschreiben.

- Klassen werden im Klassendiagramm als Rechteck dargestellt.
- Im oberen Teil findet sich der Klassenname, darunter die Variablen und die Methoden.



Variablen und Methoden können zudem mit ihrer Sichtbarkeit dargestellt werden.

- Die rot markierten Elemente sind privat, die grün markierten öffentlich.
- Mitunter werden auch andere Symbole für die Sichtbarkeit benutzt (+, -, #).

9.4 Assoziationen und Multiplizitäten

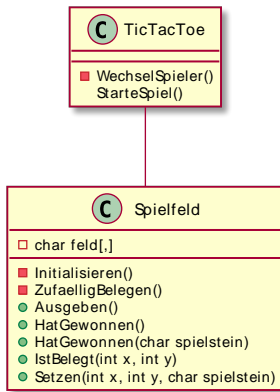
Assoziationen

Eine Klasse bildet den Bauplan für konkrete Objekte.

- Eine **Assoziation** stellt den Bauplan für konkrete Objektbeziehungen dar.
- Nur wenn zwischen Klassen eine Assoziation vorhanden ist, können die Objekte dieser Klassen Beziehungen ausprägen.

Assoziationen werden im Klassendiagramm als Linie zwischen Klassen dargestellt.

- Diese Linie ist zunächst ungerichtet.
- Das bedeutet, dass die Objekte auf beiden Seiten der Beziehung einander kennen.



Klassendiagramme auf Basis von Klassen und Assoziationen erlauben die Modellierung beliebig komplexer Informationsmodelle.

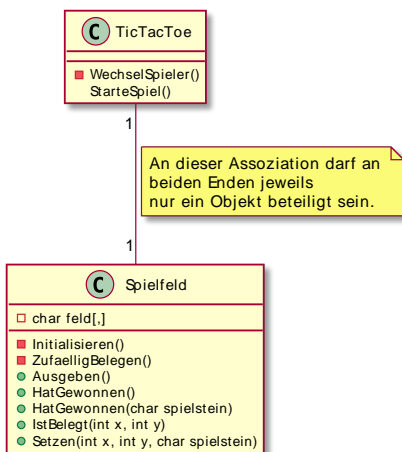
Multiplizität

Eine Assoziation wird durch **Multiplizitäten** weiter spezifiziert.

- An jedem Ende der Assoziation findet sich eine solche Multiplizität.
- Sie legt fest, wie viele Objekte zur Laufzeit an der Beziehung beteiligt sein müssen.

Bei der Assoziation zwischen der Klasse *TicTacToe* und dem *Spielfeld* muss an beiden Enden der Assoziation jeweils ein Objekt beteiligt sein.

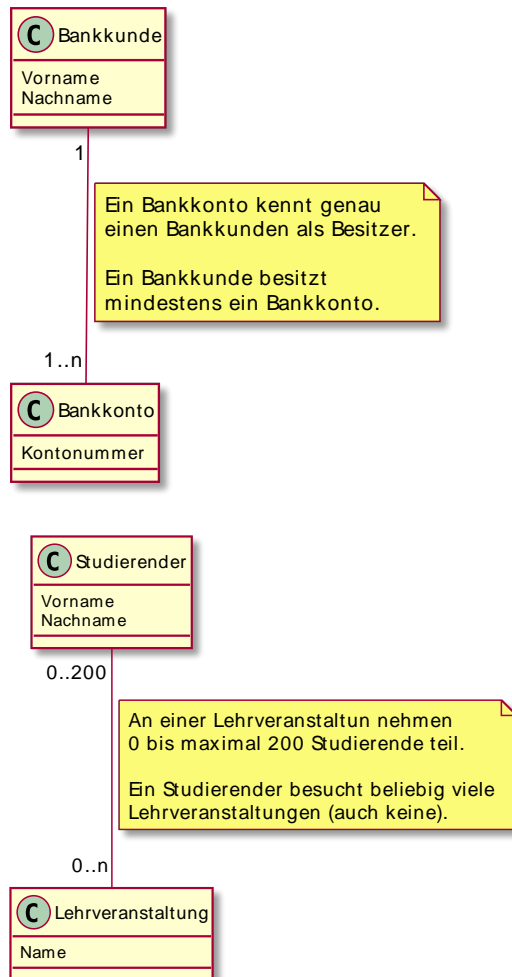
- Ein Objekt der Klasse *TicTacToe* muss also ein Objekt der Klasse *Spielfeld* kennen (und anders herum).
- Ansonsten ist das System in keinem erlaubten Zustand.



9 Analyse

Multiplizitäten machen also wichtige Vorgaben für das System.

Beispiele für Multiplizitäten im Klassendiagramm



Optionale Beziehung

Die Multiplizitäten machen im Modell wichtige Vorgaben für gültige Systemzustände.

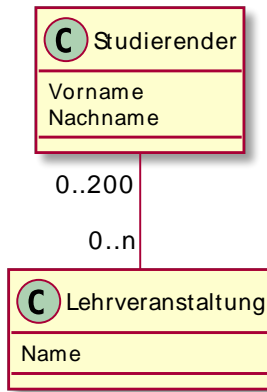
- Ein System muss sich zur Laufzeit an diese Modellvorgaben halten.

Eine Assoziation ist optional, wenn zur Laufzeit einem Objekt ein anderes Objekt zugeordnet werden kann, aber nicht muss.

- Dies wird durch die Untergrenze 0 in der Multiplizität ausgedrückt.

Im Beispiel kann einem Objekt der Klasse Studierender zur Laufzeit eine Menge von Lehrveranstaltungen zugeordnet werden.

- Aber auch keine Lehrveranstaltung zuzuordnen wäre eine gültige Ausprägung des Modells.



Verpflichtende Beziehung

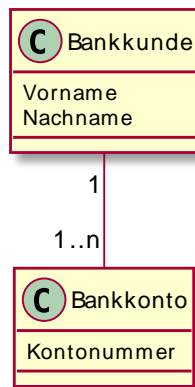
Eine Assoziation ist verpflichtend, wenn zur Laufzeit einem Objekt ein anderes zugeordnet werden muss.

- Dies wird durch die Untergrenze $n > 0$ in der Multiplizität ausgedrückt.

Im Beispiel muss dem Bankkunden zur Laufzeit mindestens ein Bankkonto bekannt sein.

- Ebenso muss ein Objekt der Klasse Bankkonto genau einen Bankkunden kennen.
- Sind diese Bedingungen nicht erfüllt, ist das System in keinem erlaubten Zustand.

Der Programmierer, der dieses Modell umsetzt, muss dafür Sorge tragen, dass diese Regeln jederzeit eingehalten werden.



Navigierbarkeit

Bislang waren alle Assoziationen ungerichtet.

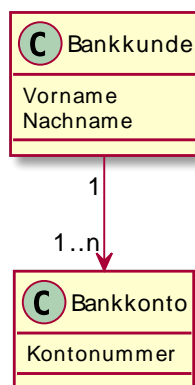
- Jede Seite kennt dann die Objekte der jeweils anderen Seite.

In der Modellierungsphase ist das sicher auch in Ordnung.

- Für die Programmierung erhöht dies aber den Aufwand.
- Die Objekte in den Beziehungen müssen auf beiden Seiten jeweils synchron gehalten werden.

Spätestens vor der Programmierung wird man daher die **Navigierbarkeit** der Assoziation einschränken.

- Man kann dann nur noch von genau einem Objekt zur anderen Seite der Beziehung navigieren.
- Eine solche Einschränkung äußert sich bei der Assoziation durch einen kleinen Pfeil.



9.5 Aggregation und Komposition

Aggregation und Komposition

Zwei speziellere Arten der Assoziation sind die **Aggregation** und die **Komposition**.

- Beide beschreiben eine Teile/Ganzes-Beziehung, die eine baumartige Hierarchie darstellt.
- Für das Ganze (die Wurzel der Hierarchie) wird oft die Bezeichnung Aggregat benutzt.
- Die Einzelteile (die Kindknoten) werden als Komponenten bezeichnet.

Folgende Beispiele werden durch eine Aggregation bzw. eine Komposition abgebildet:

- Eine Mannschaft besteht aus Spielern.
- Ein Haus besteht aus Zimmern.
- Ein Auto besteht aus einem Motor, einem Chassis und Rädern.

Ob die Komponenten unabhängig von dem Aggregat existieren können, entscheidet darüber, ob es sich um eine Aggregation oder um eine Komposition handelt.

Aggregation

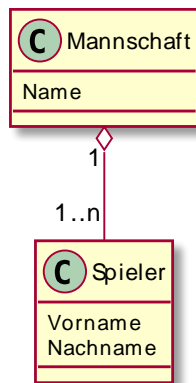
Bei der Aggregation können die Komponenten unabhängig vom Aggregat existieren.

- Die Komponenten können auch Teil eines anderen Aggregats sein.

Ein Beispiel für eine Aggregation sind die Spieler einer Mannschaft.

- Die Spieler sind Teil einer Mannschaft.
- Sie können aber auch ohne die Mannschaft existieren und auch Teil einer anderen Mannschaft sein.

Im Klassendiagramm wird eine Aggregation durch eine ungefüllte Raute an der Aggregatklasse dargestellt.



Komposition

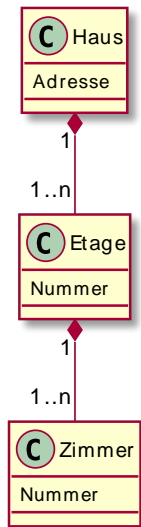
Bei der Komposition können die Komponenten nicht unabhängig vom Aggregat existieren.

- Die Komponenten können auch nur Teil eines einzigen Aggregats sein.
- Wird das Aggregat zerstört, werden auch die Komponenten zerstört.

Ein Beispiel für eine Komposition sind die Etagen eines Hauses.

- Das Haus besteht aus Etagen.
- Die Etagen können aber nicht ohne das Haus existieren.
- Auch kann eine Etage nicht Teil eines weiteren Hauses sein.

Im Klassendiagramm wird eine Komposition durch eine gefüllte Raute an der Aggregatklasse dargestellt.



Vererbung

Auch die Vererbung kann im Klassendiagramm abgebildet werden.

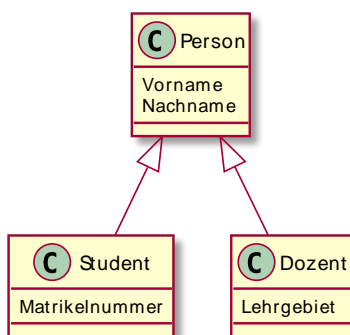
- Leitet eine Kindklasse von einer Basisklasse ab, zeigt ein geschlossener Pfeil von der Kindklasse zur Basisklasse.

Multiplizitäten existieren bei der Vererbung nicht.

- Entsprechend dürfen auch keine an den Vererbungspfeil angetragen werden.

Ebensowenig müssen die geerbten Variablen und Methoden in den Kindklassen aufgeführt werden.

- Durch die Vererbung sind sie automatisch in den Kindklassen vorhanden.



9.6 Modellbildung

Modellbildung

In der Analysephase wird ein erstes Modell des Systems entwickelt.

- Das Klassendiagramm ist dafür ein wichtiges Werkzeug.

Dabei muss die Frage beantwortet werden, welche Klassen und Assoziationen die Anforderungen am besten abdecken.

- Wie viele Objekte sind zur Laufzeit an den Beziehungen beteiligt (Multiplizitäten)?
- Sind Teile/Ganzes-Beziehungen vorhanden (Aggregation oder Komposition)?
- Gibt es Ähnlichkeiten zwischen den Klassen (Vererbung)?

Dieser Prozess wird auch als **Modellbildung**, bzw. **Modellierung** bezeichnet.

- Dabei geht es zunächst gar nicht um Technologie, wie z.B. Datenbanken, Oberflächen usw.
- In der Analysephase geht es lediglich um die **Fachlichkeit** des Systems.

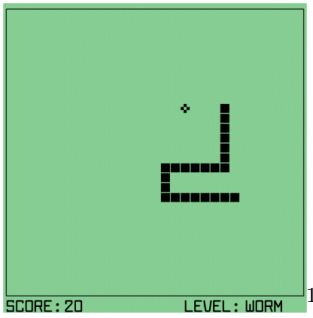
Beispiel

Wir wollen das bekannte Computerspiel **Snake** umsetzen.

- Bei diesem Spiel wird eine Schlange über den Bildschirm gesteuert.
- Die Schlange soll zufällig platzierte Äpfel fressen, um Punkte zu erhalten.
- Stößt die Schlange mit dem Kopf gegen eine Wand oder gegen den eigenen Körper, ist das Spiel verloren.

Die Schlange bewegt sich in eine von vier Richtungen über den Bildschirm.

- Die Schlange besteht aus einzelnen Blöcken, die nach und nach über den Bildschirm ziehen.
- Frisst die Schlange einen Apfel, wird sie dadurch länger.



Vorgehen

Unsere Aufgabe ist es nun, die Fachlichkeit des Spiels in einem Klassendiagramm abzubilden.

- Es müssen geeignete Klassen, Eigenschaften, Assoziationen (...) gefunden werden.
- Wir müssen uns fragen, welche Objekte zur Laufzeit existieren?
- Welche Aufgabe haben sie und wie interagieren sie miteinander?

Begriffe, die in den Anforderungen vorkommen, müssen sich im Modell wiederfinden.

- Im Beispiel ist von Schlange, Körper, Kopf, Punkte usw. die Rede.
- Bei jedem Begriff muss entschieden werden, ob es sich um eine Klasse, um die Eigenschaft einer Klasse, oder um eine Beziehung handelt.
- Zudem kann man damit beginnen, zu überlegen, welche Fähigkeiten (Methoden) die Objekte besitzen müssen.

Bei dieser Modellbildung kann es durchaus zu unterschiedlichen Varianten kommen.

- Es bedarf einiges an Erfahrung, um zu guten Modellen zu kommen.

Modell von Snake

Ein Objekt der Klasse *Spiel* verwaltet die aktuelle Punktezahl.

- Es muss das Spiel starten können und zwischenzeitlich prüfen, ob die Schlange einen Apfel gefressen oder mit einer Wand kollidiert ist.

Auch die Schlange wird als Klasse umgesetzt.

- Sie hat eine Richtung und besteht aus einer Anzahl von Blöcken.

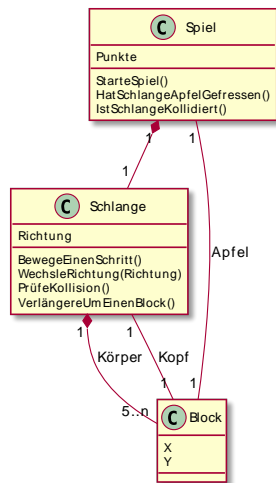
¹Bildquelle: https://www.chip.de/downloads/Snake_18757550.html

9 Analyse

- Einer dieser Blöcke ist der Kopf der Schlange.
- Die Schlange muss einige Fähigkeiten besitzen, z.B. bewegen, oder Richtung wechseln.
- Alle diese Fähigkeiten werden als Methoden umgesetzt.

Das Spiel kennt auch einen Apfel, der als Block umgesetzt ist.

- Ein einzelner Block besitzt eine Position auf dem Bildschirm.



Technische Umsetzung

Ein Klassendiagramm ist ein Bauplan, der bei der Programmierung umgesetzt werden muss.

- Klassen aus dem Modell können 1:1 in C# umgesetzt werden.
- Die Assoziationen werden in Form von Objektreferenzen abgebildet.

Als Beispiel können wir die Klasse *Schlange* umsetzen.

- Die Assoziation auf die Blöcke (Körper) wird als `List<Block>` umgesetzt.
- Der Kopf ist der erste Block im Körper der Schlange.

Lediglich einige technische Details sind noch unklar.

- Wir brauchen eine Idee, wie breit und hoch das Spielfeld (ein Fenster) sein wird.
- Dann müssen wir auch in der Lage sein, Blöcke zu zeichnen.
- Das ist aber nicht Teil der Analyse, sondern des technischen Entwurfs.

Programmcode der Klasse Schlange

```

1 public class Schlange
2 {
3     private int richtung = 0;
4     private List<Block> koerper = new List<Block>();
5
6     public Block Kopf
7     {
8         get { return koerper[0]; }
9     }
10
11    public void WechsleRichtung(int richtung)
12    {
13        this.richtung = richtung;
14    }
15
16    // Die restlichen Methoden können erst implementiert werden,
17    // wenn auch die technische Details (z.B. Breite/Höhe des Spielfelds) klar sind.
18 }

```

9.7 Zusammenfassung und Aufgaben**Zusammenfassung**

Wir haben heute gelernt, ...

- dass wir uns explizit um ein gutes Design von Software bemühen müssen.
- dass ein erstes Modell der statischen Struktur und der elementaren Verarbeitungslogik in der Analysephase entsteht.
- dass wir das Klassendiagramm der UML dazu benutzen, um die statische Struktur des Systems zu beschreiben.
- dass ein Klassendiagramm neben Klassen auch Assoziationen und Vererbung beinhaltet.
- dass Assoziationen der Bauplan für konkrete Objektbeziehungen sind.
- dass Assoziationen durch Multiplizitäten weiter beschrieben werden.
- dass Aggregation und Komposition spezielle Arten der Assoziationen sind.
- wie man aus den Anforderungen ein erstes Modell erzeugt.
- wie man dieses Modell dann technisch auf C# abbildet.

Aufgabe 1

Eine Firma zur Finanzierung von Autokäufen will eine neue Online-Anwendung aufbauen.

Folgende Anforderungen wurden dazu aufgeschrieben:

- Ein Besitzer wird durch einen Namen beschrieben.
- Ein Besitzer besitzt ein oder mehrere Autos welches durch Baujahr und Modell beschrieben werden.
- Eine Bank (Standort) erteilt für ein Auto einen Kredit mit einer Kontonummer, Zinssatz und Anfangsdatum.
- Zu Krediten können Tilgungszahlungen (Datum und Höhe) existieren.
- Ein Besitzer kann eine Person oder eine Firma sein.
- Eine Person hat ein Geschlecht, eine Firma eine Umsatzsteuernummer.

Erstellen Sie das Modell der Anwendung in Form eines UML-Klassendiagramms.

- Bilden Sie jeden Sachverhalt der Anforderungen im Modell ab.

Aufgabe 2

Ein Automobilhersteller will ein neues Navigationsgerät für seine Fahrzeugflotte realisieren.

- Sie sollen das Modell der Anwendung erstellen.

Folgende Anforderungen wurden dazu aufgeschrieben:

- Ein Ort besitzt einen Namen, einen Längen- und einen Breitengrad.
- Eine Verbindung hat eine Länge und eine Durchschnittsgeschwindigkeit.
- Eine Verbindung führt von einem Start-Ort zu einem Ziel-Ort (unterscheidbar!).
- Eine Region besteht aus vielen Orten und Verbindungen, eine Karte besteht aus Regionen.
- Ein Ort kann eine Stadt, eine Sehenswürdigkeit oder eine Tankstelle sein.
- Eine Route besteht aus einer Menge von Verbindungen.

Die Klasse Routenplaner soll die Funktion *BerechneSchnellsteRoute()* besitzen.

- Welche Parameter muss die Funktion bekommen? Welches Ergebnis wird sie liefern?

Aufgabe 3

Erstellen Sie das Modell eines beliebigen Brettspiels, z.B. Mensch-ärgere-dich-nicht.

- Notieren Sie zunächst einige Anforderungen die darlegen, welche Elemente im Spiel überhaupt existieren und wie diese in Beziehung miteinander stehen.

Erstellen Sie dann dazu ein passendes UML-Klassendiagramm.

- Welche Fähigkeiten müssen die Objekte der Klassen zur Laufzeit besitzen?
- Gibt es alternative Modellierungen?
- Warum haben Sie sich für dieses Modell entschieden?

10 Analysemuster

Muster

Hat man einige Softwaresysteme umgesetzt, stellt man fest, dass sich gewisse Strukturen aus Klassen, Schnittstellen und Beziehungen häufig wiederholen.

- Solche Strukturen, die geeignet sind, gewisse Probleme zu lösen, nennt man **Muster**.

Muster sind uncodierte Abstraktionen, man kann sie nicht direkt wie eine Bibliothek nutzen.

- Vielmehr dienen Muster als Vorlage für die Konstruktion einer bestimmten Problemlösung.
- Sie helfen dabei, Entwürfe zu strukturieren und zu kommunizieren und geben Orientierung in komplexen Systemen.
- Muster sind kondensierte Erfahrung, der man sich bedienen kann (aber nicht muss).

Muster spielen in allen Phasen und für alle Aspekte der Softwareentwicklung eine Rolle.

- In der Analyse, im Entwurf, für kleine Details oder große Architekturfragen.
- Entsprechend unterschiedlich ist der Abstraktionsgrad von Mustern.

10.1 Analysemuster

Analysemuster

Muster, die in der Analysephase eingesetzt werden, werden als **Analysemuster** bezeichnet.

- Sie dienen dazu, fachliche Sachverhalte der geschäftlichen Praxis beispielhaft auf objektorientierte Strukturen abzubilden, siehe [3].

10 Analysemuster

- Man spricht auch von den *konzeptuellen Facetten* des Systems bzw. dem *Domänenmodell*.
- Technischen Details spielen in der Analyse bekanntlich zunächst eine untergeordnete Rolle.

Analysemuster befassen sich oft mit Konzepten, die in Unternehmen anzutreffen sind, z.B.:

- Organisationsstrukturen, Mitarbeiter, Kunden, Konten, Produkte, Bestellungen, Mengen- und Zeitangaben, Verträge, usw.
- Es ist alles andere als trivial, all diese Dinge sinnvoll und präzise auf Klassen und Beziehungen abzubilden.

Im Folgenden werden einige Szenarien und dafür hilfreiche Analysemuster vorgestellt.

10.2 Historie, Quantität, Intervall und Koordinator

Gehalt

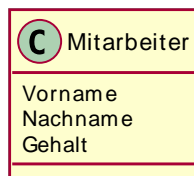
Die Mitarbeiter eines Unternehmens werden in einem System durch die gleichnamige Klasse abgebildet.

- Die Klasse besitzt die Eigenschaft Gehalt.
- Diese Eigenschaft kann zu jedem Zeitpunkt nur genau einen Wert aufnehmen.

Wird das Gehalt durch einen neuen Wert überschrieben, geht der alte Wert verloren.

- Es ist dann nicht nachzuvollziehen, welche unterschiedlichen Gehälter ein Mitarbeiter über die Zeit hinweg erhalten hat.

Nun soll aber der zeitliche Verlauf des Gehalts abgebildet werden können.



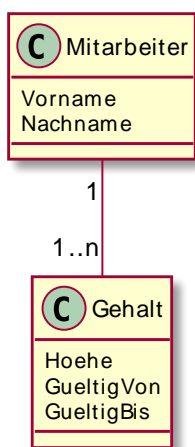
Historie

Dieses Problem wird durch das Analysemuster **Historie** adressiert.

- Die zu historisierende Eigenschaft wird als eigene Klasse extrahiert.
- In unserem Beispiel ist dies die Klasse *Gehalt*.
- Sie erhält die Höhe des Gehalts und den Gültigkeitszeitraum als Eigenschaft.

Über eine Assoziation können dem Mitarbeiter dann mehrere Objekte der Klasse *Gehalt* zugeordnet werden.

- Die Veränderung der Höhe des Gehalts kann dann nachvollzogen werden.
- Die Gültigkeitszeiträume der Gehaltsobjekte dürfen sich nicht überschneiden.



Quantität

Attribute, wie die Höhe des Gehalts, werden häufig über einen elementaren Datentyp (z.B. **double**) abgebildet.

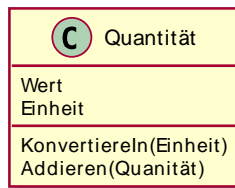
- Eine solche Abbildung ist aber sehr unpräzise.
- Möglicherweise hat das Unternehmen Standorte in unterschiedlichen Ländern mit unterschiedlichen Währungen.

Bei solchen Angaben sollte daher auf die Verwendung eines einfachen Datentyps verzichtet werden, siehe [3].

- Stattdessen sollte ein neuer Typ **Quantität** eingeführt werden
- Ein Objekt dieser Klasse kann dann die Einheit des Werts (einfach oder zusammengesetzt) definieren.

10 Analysemuster

- Auch können Transformationsmethoden und Rechenoperationen korrekt abgebildet werden.



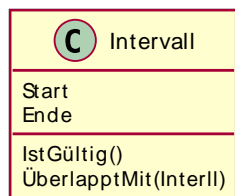
Intervall

Um im Analysemuster Historie die Eigenschaft einer Klasse zu historisieren, wird eine neue Klasse über eine Assoziation verbunden.

- Diese neue Klasse beinhaltet einen Gültigkeitszeitraum, der angibt, in welchem Intervall das Attribut den entsprechenden Wert annimmt.
- Dieser Gültigkeitszeitraum sollte bestenfalls ebenfalls als eigene Klasse abgebildet werden.
- So können entsprechende Prüfmethoden implementiert werden.

In unserem Beispiel handelt es sich bei der Unter- und Obergrenze um Datumswerte.

- Aber auch andere, numerische Werte sind möglich und denkbar.



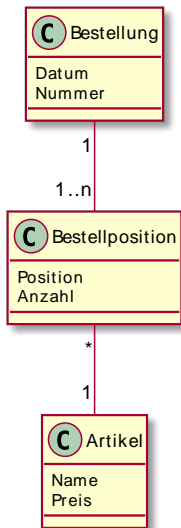
Bestellung

In einem Shop-System müssen Bestellungen abgebildet werden.

- Einer Bestellung sind viele Artikel zugeordnet.
- Bestellung und Artikel besitzen eigene Attribute.
- Aber auch die Zuordnung der Artikel zu der Bestellung besitzt Eigenschaften, z.B. die Bestellmenge oder die Position des Artikels.

Dieses Szenario wird durch das Analysemuster **Liste** bzw. **Koordinator** adressiert.

- Neben der Klasse Bestellung und Artikel wird eine eigene **Assoziationsklasse** *Bestellposition* eingeführt.
- Diese verbindet einen Artikel mit der Bestellung.
- Zudem kann in der Bestellposition die Menge usw. abgebildet werden.



10.3 Reflexive Assoziation und Kompositum

Produktkategorie

In einem Online-Bestellsystem sollen Produkte in Kategorien organisiert werden können.

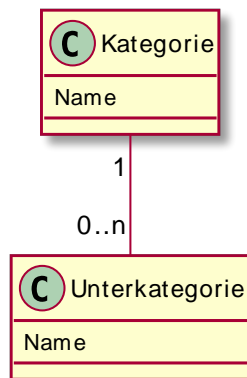
- Eine Produktkategorie besitzt dabei einen Namen.
- Eine Produktkategorie kann Unterkategorien besitzen.

Eine naive Modellierung bildet eine Kategorie und eine Unterkategorie jeweils als eigene Klasse ab.

- Beide Klassen werden über eine Assoziation miteinander verbunden.

Dies ist aber keine gute Modellierung.

- In der Analysephase bilden Klassen fachliche Konzepte ab.
- Es existiert aber kein konzeptueller Unterschied zwischen einer Kategorie und einer Unterkategorie.



Reflexive Assoziation

Eine Assoziation darf auch eine Klasse mit sich selbst verbinden.

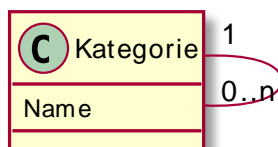
- Eine derartige Assoziation wird **reflexive Assoziation** genannt.

So kann das vorherige Szenario viel geschickter abgebildet werden.

- Einem Objekt der Klasse Kategorie können beliebig viele andere Kategorie-Objekte als Unterkategorien zugeordnet werden.

Eine solche Modellierung kann alle möglichen Bäume bzw. Graphen abbilden.

- z.B. Vater/Mutter/Sohn, Verzeichnisstrukturen, usw.



Kompositum

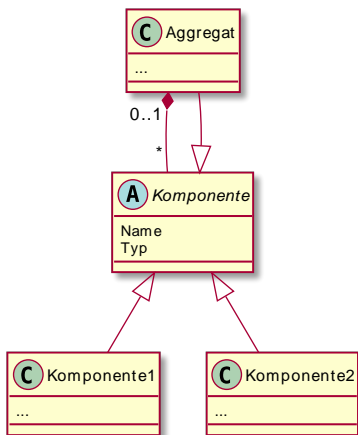
Das Analysemuster **Kompositum** nutzt ebenfalls die reflexive Assoziation als Stilmit-

- Es wird immer dann eingesetzt, wenn ein größeres Ganzes, z.B. eine Maschine oder System, aus Einzelteilen besteht und als Modell abgebildet werden muss.

Die abstrakte Basisklasse abstrahiert dabei von konkreten Komponenten.

- Eine Aggregat besteht aus einer beliebigen Menge von Komponenten.

- Da das Aggregat aber selber eine Komponente ist, kann es auch aus weiteren Aggregaten bestehen.



10.4 Exemplartyp und Vorrat

Autovermietung

Eine Autovermietung hat sehr viele Fahrzeuge im Bestand.

- Die Klasse Fahrzeug besitzt Eigenschaften, wie den Herstellernamen, das Modell, die Fahrgestellnummer und das Kennzeichen.

Diese Modellierung zeigt aber einige Nachteile:

- Die Werte mancher Eigenschaften haben eine hohe Redundanz, z.B. wiederholt sich der Hersteller sehr oft.
- Es kann kein Fahrzeugmodell eingeführt werden, wenn nicht auch schon ein konkretes Fahrzeug dazu im Bestand ist.

Offenbar sollte die Klasse in mehrere Konzepte aufgeteilt werden.



Exemplartyp

Das Analysemuster **Exemplartyp** adressiert diese Situation.

- Die Eigenschaften werden dabei auf zwei Klassen verteilt.

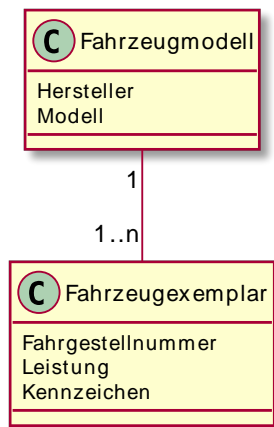
Die Klasse *Fahrzeugmodell* sammelt alle modellspezifischen Eigenschaften.

- Dies ist z.B. der Hersteller und das Modell.

Eine weitere Klasse *Fahrzeugexemplar* repräsentiert die konkreten Fahrzeuge im Bestand.

- Diese trägt Eigenschaften, wie die Fahrgestellnummer und die Leistung.

Dem Fahrzeugmodell können dann viele konkrete Fahrzeugexemplare zugeordnet werden.



Vorrat

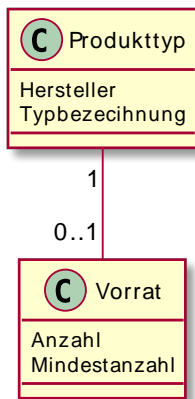
Das Analysemuster Exemplartyp sollte immer dann angewandt werden, wenn die einzelnen Exemplare voneinander unterscheidbar sind.

- z.B. Fahrzeuge, Bücher, ...

Bei einem anonymen **Vorrat** können die Exemplare aber nicht voneinander unterschieden werden.

- z.B. Schrauben, Betriebsstoffe, ...

Der Vorrat wird dann als ein einzelnes Objekt abgebildet.



10.5 Rollen und wechselnde Rollen

Rollen

In einer Beziehung können Objekte gegenüber anderen Objekten **Rollen** einnehmen.

- Eine Person ist gegenüber einem Ort der Bewohner.
- Der Ort ist gegenüber der Person der Wohnort.

Eine Assoziation zwischen Klassen bildet dann die Bedeutung einer Rolle ab.

- Der Rollenname kann zum besseren Verständnis auch an die Enden der Assoziation geschrieben werden.

Objekte einer Klasse können auch mehrere, unterschiedliche Rollen gegenüber einem anderen Objekt einnehmen.

- Eine Person kann in einer Fußballmannschaft der Trainer oder der Torwart sein.

Das Konzept der Rollen ist ebenfalls ein bekanntes Analysemuster.

Beispiel

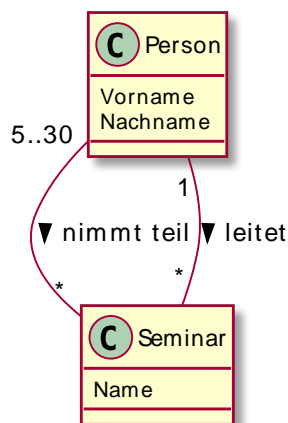
Eine Person kann für ein Seminar der Teilnehmer oder der Dozent sein.

- Beide Rollen werden als jeweils eigene Assoziation zwischen den Klassen Person und Seminar abgebildet.

Der Dozent sollte dabei nicht gleichzeitig auch Teilnehmer sein.

- Diese Bedingung lässt sich mit dem Klassendiagramm allein nicht darstellen.
- Eine solche Bedingung kann aber mithilfe der Object Constraint Language (OCL) ausgedrückt werden, siehe [5].

Die Art und Anzahl der Rollen muss bei dieser Modellierung über die Zeit stabil bleiben.



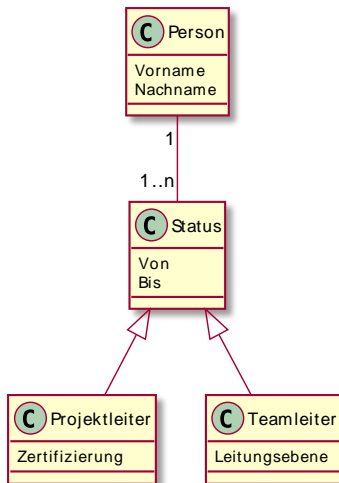
Wechselnde Rollen

Sind die Rollen nicht starr und können sich potenziell ändern, wird das Analysemuster der **wechselnden Rollen** angewandt.

- Im Beispiel kann der Person, ähnlich wie in der Historie, eine zeitlich begrenzte Rolle zugewiesen werden.

Die konkreten Rollen, Team- oder Projektleiter, werden dann von einer gemeinsamen (abstrakten) Basisklasse abgeleitet.

- Auch hier sollten sich die Gültigkeitszeiträume nicht überschneiden.



Kombination von Mustern

Muster sind keine starren Konstrukte, sondern Hilfsmittel bei der Umsetzung von Softwaresystemen.

- Muster helfen dabei, Probleme zu erkennen und diese mit bewährten Strategien zu lösen.
- Alle Muster, egal ob Analyse- oder Entwurfsmuster, können beliebig miteinander kombiniert und an die eigenen Bedürfnisse angepasst werden.

Zum Beispiel kann mit der Historie jede Eigenschaft einer Klasse historisiert werden.

- Dabei kann es sich z.B. auch um die Rolle eines Mitarbeiters (Rollen) oder die Zuordnung einer Komponente eines Aggregats handeln.
- Die Quantität oder das Intervall kann auch leicht mit einer Koordinator-Klasse verbunden werden.
- Im Prinzip gibt es keine Grenzen bei der Kombination der Muster.

10.6 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, ...

- was Muster im Allgemeinen sind und welchen Verwendungszweck sie haben.

10 Analysemuster

- dass Analysemuster dabei helfen, die Fachlichkeit eines Systems in einem Modell abzubilden.
- welche Probleme die Analysemuster Historie, Quantität, Intervall und Koordinator lösen.
- wozu man reflexive Assoziationen nutzen kann und wie dieses Strukturmittel im Muster Kompositum angewandt wird.
- wie die Analysemuster Exemplartyp und Vorrat angewandt werden.
- wie man die Analysemuster Rollen und wechselnde Rollen verwendet.

Aufgabe 1

In den Aufgaben des vorherigen Abschnitts sollten einige Informationsmodelle in Form von UML-Klassendiagrammen erstellt werden.

- Untersuchen Sie die Modelle auf die Anwendbarkeit von Analysemustern.
- Wo ist es sinnvoll, Analysemuster anzuwenden?

Aufgabe 2

Überlegen Sie, wie das Informationsmodell eines Streaminganbieters wie Spotify oder Apple Music aussehen könnte.

- Welche Muster finden hier Verwendung?

11 Entwurf

11.1 Analyse vs Entwurf

Analyse

In den letzten Abschnitten haben wir uns mit der **Analysephase** der Softwareentwicklung befasst.

- Das Ziel dieser Phase ist es, ein erstes **Informationsmodell** einer Anwendung zu erstellen.
- Zudem wird meist auch die elementare **Verarbeitungslogik** festgelegt.

Beide Facetten (Struktur und Dynamik) können mithilfe der UML modelliert werden.

- Das Informationsmodell wird dabei meist als Klassendiagramm erstellt.
- Für die Verarbeitungslogik werden z.B. Aktivitäts- oder Sequenzdiagramme genutzt.

In der Analysephase geht es hauptsächlich um die **Fachlichkeit** des Systems.

- Die entstehenden Modelle sollen die Frage beantworten, was genau das System leisten soll und welche Strukturen dazu nötig sind.
- Technische Details zur Umsetzung (ob als Web-Anwendung, oder als App) werden dabei zunächst (soweit es geht) ausgeblendet.

Entwurf

In der **Entwurfsphase** werden die Ergebnisse der Analyse aufgegriffen und erweitert.

- Der Entwurf befasst sich nun mit allen technischen Details, z.B.:
- um welche Art von Anwendungssystem es sich handeln soll (Web, App, Desktop, ...),
- wie die Benutzeroberfläche gestaltet wird,
- wie und wo Daten gespeichert werden,

- wie Benutzer authentifiziert und autorisiert werden, ...

Durch die Festlegung all dieser Details entsteht letztendlich die **Softwarearchitektur**.

- Meistens wird der Begriff der Softwarearchitektur so definiert, dass es sich dabei um die fundamentale Struktur/Organisation des Softwaresystems handelt.
- Martin Fowler, ein anerkannter Experte, bezeichnet Softwarearchitektur eher als die Summe aller wichtigen und schwer änderbaren Entscheidungen¹.

Qualität des Entwurfs

In dieser Veranstaltung soll es nicht primär um spezielle Technologien gehen.

- Wir wollen uns hier keine speziellen Frameworks ansehen, die z.B. bei der Entwicklung von Web-Anwendungen oder Apps für mobile Endgeräte wichtig sind.

Wir wollen aber lernen, wie man zu einem gelungenen, objektorientierten Entwurf kommt.

- Dabei muss entschieden werden, wie die fachlichen und technischen Anforderungen auf Klassen, Schnittstellen und Beziehungen abgebildet werden sollen.
- In der Realität existieren dafür beliebig viele Varianten.
- Welche Entwurfsvariante ist aber besser, welche schlechter geeignet?

Der Entwurf muss nach bestimmten Qualitätskriterien bewertet werden.

- Hochwertige Software ist aber nicht nur funktional, effizient und zuverlässig.

Evolvierbarkeit

Da sich die Umgebung von Software über die Zeit ändert, muss langlebige Software kontinuierlich angepasst werden.

- Die Software muss anpassbar, weiterentwickelbar, evolvierbar sein.

Manche Entwürfe zeigen beim Versuch der Weiterentwicklung aber eine gewisse *Alterung*.

- Das sind Eigenschaften, die auch alternde Lebewesen besitzen, siehe auch [13, S. 2-6]:
- Starrheit, Zerbrechlichkeit, Unbeweglichkeit, Zähigkeit.

¹<https://martinfowler.com/architecture/>

Robert C. Martin spricht dann davon, dass Software verrottet, vermodert (siehe [6]).

- Verrottende Software entwickelt dann gewisse Gerüche (*engl. smells*).
- Eine ganze Reihe von Werken befassen sich mit sog. Code Smells und ihrer Vermeidung, siehe z.B. [18].

11.2 SOLID

Kohäsion und Kopplung

Ob ein Entwurf evolvierbar ist, wird insbesondere auch durch die strukturelle Aufteilung auf Klassen und Schnittstellen beeinflusst.

- Die Qualität des Entwurfs schlägt sich dabei in zwei Eigenschaften nieder: **Kohäsion** und **Kopplung**.

Kohäsion meint, wie gut Programmelemente eine einzelne logische Aufgabe abdecken.

- In einem System mit starker Kohäsion ist z.B. eine einzelne Klasse für genau eine Aufgabe zuständig.

Kopplung dagegen ist ein Gradmesser, wie stark die Abhängigkeiten zwischen Klassen sind.

- In einem System mit geringer Kopplung können einzelne Klassen leicht gegen andere Varianten ausgetauscht werden.
- Eine Änderung an einer Klasse führt auch nicht zwangsläufig dazu, dass an vielen anderen Klassen Änderungen vorgenommen werden müssen.

SOLID

Eine Reihe von **Entwurfsprinzipien** zielen darauf ab, die Kohäsion zu erhöhen und die Kopplung zu reduzieren.

- Solche Prinzipien sind keine starren Regeln, sondern Empfehlungen.
- Sie helfen aber dabei, qualitativ hochwertige Entwürfe zu erstellen.

Ein bekannter Satz solcher Prinzipien ist unter der Abkürzung **SOLID** bekannt, siehe [4].

- Die einzelnen Buchstaben dieses Begriffs stehen für:
- **S**: Single Responsibility Principle (SRP)

- **O**: Open Closed Principle (OCP)
- **L**: Liskov Substitution Principle (LSP)
- **I**: Interface Segregation Principle (ISP)
- **D**: Dependency Inversion Principle (DIP)

Wir werden uns im Folgenden alle diese Prinzipien ansehen.

11.3 Prinzip der eindeutigen Verantwortlichkeit

Single Responsibility Principle

Das **Prinzip der eindeutigen Verantwortung** (engl. Single Responsibility Principle) zielt direkt auf die Erhöhung der Kohäsion ab.

- Es besagt, dass Programmelemente nur eine Anforderung eines Akteurs umsetzen sollen.
- Unterschiedliche Aufgaben und Aspekte sollen getrennt voneinander implementiert werden (engl. separation of concerns).
- Anders gesagt, darf es nur einen Grund geben, ein Programmelement ändern zu müssen, siehe [6, S. 114-119].

Aus diesem Prinzip lassen sich viele weitere Heuristiken direkt oder indirekt ableiten, z.B.:

- Redundanzen im Programmcode müssen vermieden werden (engl. Don't repeat yourself).
- Eine Methode sollte entweder andere Methoden integrieren, oder selber Logik implementieren, aber nicht beides.
- Eine Methode sollte entweder ein Kommando oder eine Abfrage umsetzen, aber nicht beides (engl. Command-Query-Separation).

CSV-Daten

Wir wollen uns an einem Beispiel ansehen, wie man dieses Prinzip umsetzen kann.

- In einem Unternehmen werden alle Verkäufe in sog. CSV-Dateien abgelegt.
- CSV-Dateien sind Text-Dateien, die tabellarische Daten beinhalten.
- Die Spalten sind durch ein Semikolon voneinander getrennt.
- Die erste Zeile beinhaltet die Spaltenüberschriften.

Eine solche Datei könnte wie folgt aussehen:

```
1 Produkt;Preis;Anzahl
2 SSD Festplatte;48,9;3
3 Monitor;129,3;5
4 Tastatur;10,49;3
5 Maus;62,99;7
```

Unsere Aufgabe ist es nun, ein Programm zu entwickeln, welches die Datei einliest und den Gesamtumsatz berechnet und ausgibt.

Umsatzleser

Eine erste, sehr einfache Lösung könnte wie folgt aussehen:

```
1 class UmsatzLeser
2 {
3     public static void Main()
4     {
5         string dateiname = "Daten.csv";
6         var reader = new StreamReader(dateiname);
7         reader.ReadLine();
8
9         while (!reader.EndOfStream)
10        {
11            string line = reader.ReadLine();
12            var parts = line.Split(';');
13            var preis = Convert.ToDouble(parts[1]);
14            var anzahl = Convert.ToDouble(parts[2]);
15            gesamtsatz += anzahl * preis;
16        }
17        reader.Close();
18    }
19 }
```

Bewertung

Diese Lösung funktioniert zwar, ist aber kein wirklich guter Entwurf.

- Wir haben in der Klasse *UmsatzLeser* zu viele unterschiedliche Dinge getan.
- Die Klasse liest eine Datenquelle (Datei), die Daten werden in das gewünschte Format transformiert und Berechnungen werden angestellt.

Keiner der einzelnen Aspekte kann aktuell eigenständig getestet werden.

- Die Transformation der Daten oder die Berechnung funktioniert nicht ohne eine CSV-Datei.

Wir wollen daher den Entwurf verbessern.

- Dazu verteilen wir die einzelnen Aspekte auf mehrere eigene Klassen und Methoden.

Nach der Änderung

Wir erstellen einige neue Klassen:

- Ein Objekt der Klasse *SalesPosition* bildet genau eine Zeile der Datei ab.
- Die Klasse *CsvParser* transformiert die Datei in eine Menge solcher Objekte.
- Die Klasse *SalesCalculator* berechnet dann den Umsatz aus den Objekten.

Die Lösung sieht danach wie folgt aus:

```
1 string[] file_content = File.ReadAllLines("Data.csv");
2 IEnumerable<SalesPosition> sales = CsvParser.Parse(file_content);
3 double turnover = SalesCalculator.GetTurnover(sales);
4 Console.WriteLine(turnover);
```

Alle diese Klassen können unabhängig voneinander genutzt und getestet werden.

- Die Kohäsion der neuen Klassen ist viel höher, als in der ersten Lösung.
- Die Anwendung ist deutlich anpass-, portier- und wartbarer als vorher.

11.4 Prinzip der Offen- und Verschlussenheit

Open Closed Principle

Das Prinzip der **Offen- und Verschlussenheit** (engl. **Open Closed Principle**) ist das nächste Entwurfsprinzip, dass wir uns ansehen wollen.

- Das Prinzip besagt, dass Programmelemente für Modifikationen verschlossen sein sollten.
- Für Erweiterungen sollten Sie aber offen sein, siehe [6, S. 120-132].

Den bestehenden Programmcode einer Klasse zu verändern ist eine Modifikation.

- Eine Modifikation ändert das Verhalten von bestehenden Systemen.
- Entsprechend müssen Modifikationen sehr vorsichtig vorgenommen werden.

Eine Erweiterung jedoch fügt dem System zusätzliches Verhalten hinzu.

- Dies kann z.B. durch Ableiten von einer bestehenden Klasse bzw. Implementieren einer Schnittstelle geschehen.
- Es muss dann nur diese neue Klasse getestet werden, nicht jedoch der bestehende Programmcode.

Beispiel

Ein Beispiel für das Open Closed Principle haben wir bereits gesehen.

- In unserem Tic Tac Toe Spiel haben wir eine abstrakte Basisklasse *Spieler* definiert.

```

1  abstract class Spieler
2  {
3      public char Spielstein { get; set; } 4
4      public Spieler(char spielstein)
5      {
6          Spielstein = spielstein;
7      }
8
9      public abstract void Ziehe(Spielfeld feld);
10 }
```

Die Klasse gibt das Verhalten vor, das eine Klasse anbieten muss, um als Spieler zu gelten.

- Daraus haben wir dann zwei unterschiedliche Arten von Spielern abgeleitet.
- Eine Klasse *ComputerSpieler* und einen *MenschlichenSpieler*.

Netzwerkspieler

Wir können nun eine weitere Art von Spieler einführen, den *Netzwerkspieler*.

- Diese könnte dann den Zug von einem entfernten Rechner empfangen.
- Wenn wir die neue Klasse von der Klasse *Spieler* ableiten, müssen wir lediglich die Methode *Ziehe()* entsprechend implementieren.

Wir können unser System dadurch ganz einfach um weitere Funktionalität erweitern.

- Die abstrakte Klasse *Spieler* abstrahiert von den konkreten Spielerarten.
- Mithilfe der Polymorphie realisieren wir eine Art Steckdose, in der wir unterschiedliche Arten von Spielern einstecken können.

Wir erweitern dadurch die Anwendung um neue Funktionalität, ohne bestehende Klassen verändern zu müssen.

11.5 Liskovsches Substitutionsprinzip

Netzwerkspieler

Betrachten wir nun einen ersten Entwurf unserer Klasse *Netzwerkspieler*:

```
1 public class Netzwerkspieler : Spieler
2 {
3     public bool IstVerbunden { get; private set; }
4
5     public void Verbinden() {
6         // Hier wird die Verbindung aufgebaut
7         IstVerbunden = true;
8     }
9
10    public override void Ziehe(Spielfeld feld) {
11        if (!IstVerbunden)
12            throw new Exception("Verbindung ist nicht aufgebaut!");
13
14        // Nun wird der Zug vom Server geholt...
15    }
16 }
```

Liskov Substitution Principle

Bei dieser Implementierung gibt es leider ein Problem.

- Bevor ein Zug ausgeführt werden kann, muss mit der Methode *Verbinden()* zunächst eine Verbindung zu einem Server aufgebaut worden sein.

Ein Objekt der Klasse *Spiel* macht dies natürlich nicht.

- Das Spiel kennt die konkrete Implementierung der abstrakten Klasse *Spiel* nicht und weiß nichts über die Methode *Verbinde()*.
- Der Netzwerkspieler kann daher nicht ohne Weiteres in unserem Spiel eingesetzt werden.

Das sog. **Liskovsche Substitutionsprinzip** (engl. **Liskov Substitution Principle**) befasst sich genau mit dieser Problematik.

- Es beschreibt, wie Kindklassen aufgebaut sein müssen, damit sie in Systemen als gleichwertiger Ersatz von Basisklassen genutzt werden können.

Offenbar wurde in unserem ersten Entwurf gegen dieses Prinzip verstoßen.

Design by Contract

Der Netzwerkspieler funktioniert nur dann vernünftig, wenn auf dem Objekt vorher einmal *Verbinden()* aufgerufen wurde.

- Eine solche Forderung wird auch als **Vorbedingung** bezeichnet.

Der Begriff der Vorbedingung stammt aus dem Konzept des **Design by Contract**.

- Damit eine Methode vernünftig arbeiten kann, müssen alle Vorbedingungen erfüllt sein.
- Bei der Methode *Ziehe()* ist das die Vorbedingung, dass die Verbindung aufgebaut wurde.

Nach der Erfüllung einer Aufgabe sind dann i.d.R. verschiedene **Nachbedingungen** erfüllt.

- Für die Methode *Ziehe()* gilt, dass auf dem Spielfeld nach dem Zug ein Spielstein mehr liegen muss als vorher.

Zudem müssen durchgängig gewisse Grundannahmen (sog. **Invarianten**) gelten.

- So bleibt die Dimension des Spielfelds durchgängig gleich (3x3).

Problemlösung

Ob in einer Vererbungshierarchie das Liskovsche Substitutionsprinzip eingehalten werden kann, entscheidet sich anhand folgender Regeln:

- Die Kindklasse darf die **Vorbedingungen** der Basisklasse höchstens **abschwächen**.
- Die Kindklasse darf die **Nachbedingungen** der Basisklasse höchstens **verstärken**.

In unserem Fall wurden die Vorbedingungen der Kindklasse Netzwerkspieler allerdings erhöht.

- Entsprechend konnten wir den *Netzwerkspieler* nicht einfach in unser Spiel integrieren.

Wir können das Problem aber lösen.

- Die Vorbedingung der aufgebauten Verbindung muss immer dann erfüllt sein, wenn wir die Methode *Ziehe()* aufrufen.
- Wir können die Verbindung z.B. bereits im Konstruktor der Klasse aufbauen, oder bei jedem Zug prüfen, ob die Verbindung besteht und diese im Zweifel aufbauen.

11.6 Schnittstellen Segregationsprinzip

Abhängigkeiten

Abhängigkeiten entstehen dadurch, dass sich Objekte unterschiedlicher Klassen in einem Geflecht gegenseitig nutzen (Client-Server-Prinzip).

- Diese gegenseitige Nutzung von Objekten ist ein Kernprinzip der Objektorientierung.
- Das TicTacToe-Spiel kann z.B. ohne Spieler nicht funktionieren.

Viele Abhängigkeiten (ein hoher Grad der Kopplung) können aber zu einer deutlich schlechteren Anpassbarkeit und Wartbarkeit eines Systems führen.

- Selbst kleine Änderungen können dann einen erheblichen Arbeitsaufwand erzeugen.

In manchen Systemen kommt man an dem Punkt an, dass Änderungen unüberschaubare Effekte mit sich bringen.

- Es darf möglichst nichts mehr geändert werden.

Beispiel

In einem vorhergehenden Abschnitt haben wir eine Anwendung entwickelt, um einen Sportverein verwalten zu können.

```

1  class Sportverein
2  {
3      private Mitglied vorsitzender;
4      private List<Mitglied> mitglieder = new List<Mitglied>();
5
6      // Hier fehlt einiges an Programmcode...
7
8      public IEnumerable<Mitglied> AlleMitglieder()
9      {
10         return mitglieder;
11     }
12 }

```

Ein Objekt der Klasse *Sportverein* verwaltet einen Vorsitzenden und mehrere Mitglieder.

- Die Mitglieder werden in einer Objektvariablen vom Typ `List<Mitglied>` verwaltet.

List oder IEnumerable

In der Methode *AlleMitglieder()* liefern wir als Ergebnis ein Objekt vom Typ `IEnumerable<Mitglied>` zurück, nicht die Liste selbst.

- Dies hat insbesondere etwas mit **Abhängigkeiten** zwischen Klassen zu tun.

Die Schnittstelle *IEnumerable* abstrahiert von allen möglichen Arten von Sammlungen.

- Dadurch könnte in der Klasse *MitgliederListe* die Verwaltung der Mitglieder z.B. auf ein Array umgestellt werden.
- Der Nutzer der Klasse würde dies nicht bemerken.

Abhängigkeiten sollten möglichst nur zu **Abstraktionen** (z.B. Schnittstellen), nicht aber zu konkreten Klassen aufgebaut werden.

- Variablen sollte keine Referenz auf eine konkrete Klasse halten, lediglich zu Abstraktionen.
- Klassen sollten nicht von konkreten Klassen ableiten.

Interface Segregation Principle

Schnittstellen sind das wichtigste Mittel, um von konkreten Klassen zu abstrahieren.

- Schnittstellen sollten dabei so schlank wie möglich sein.

Ein Modul sollte über eine Schnittstelle nur diejenigen Methoden präsentiert bekommen, die es für eine bestimmte Aufgabe auch wirklich braucht.

- Aber eben nicht mehr.

Diese Forderung ist auch als das **Schnittstellen-Segregationsprinzips** (engl. **Interface Segregation Principle, ISP**) bekannt, siehe [6, S. 166].

- Zu schwergewichtige Schnittstellen sollten in mehrere Schnittstellen aufgetrennt werden.
- Dieses Prinzip ist eng verwandt mit dem Prinzip der eindeutigen Verantwortung.
- Auch Abstraktionen (Schnittstellen) sollten dieser eindeutigen Verantwortung gerecht werden.

11.7 Abhängigkeits-Umkehrprinzip

Dependency Inversion Principle

In unserem TicTacToe-Spiel benutzt das Spiel unterschiedliche Arten von Spielern.

- Es entsteht eine Abhängigkeitsbeziehung zwischen diesen Elementen.
- Die Klasse *Spiel* kann darin als das übergeordnete Modul bezeichnet werden.

Das **Abhängigkeitsumkehr-Prinzip** (engl. **Dependency Inversion Principle, DIP**) stellt in einer solchen Beziehung die folgenden Regeln auf, siehe [6, S. 151-161]:

- Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen.
- Beide sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängen.
- Details sollten von Abstraktionen abhängen.

Robert C. Martin bezeichnet die Einhaltung dieses Prinzips auch als das Markenzeichen guten objektorientierten Designs, siehe [6, S. 161].

Dependency Inversion Principle

Die Abhängigkeit zwischen Spiel und Spieler haben wir bereits über eine Abstraktion geregelt.

- Wir haben gesehen, dass dazu entweder eine Schnittstelle *ISpieler* oder eine abstrakte Basisklasse eingesetzt werden kann.

Das Abhängigkeitsumkehr-Prinzip benennt dabei auch, wer für die Definition dieser Abstraktion verantwortlich sein sollte.

- Die höhere Ebene (das Spiel) sollte diese Abstraktion vorgeben.
- Das Spiel sollte definieren, was es sich unter einem Spieler vorstellt.
- Die Spieler müssen diese Vorgabe dann einhalten.

Würden Spiel und die Spieler in unterschiedlichen Bibliotheken implementiert, hätte die Spieler-Bibliothek dann eine Abhängigkeit zu der Spiel-Bibliothek.

- Nicht anders herum.
- Dadurch entsteht auch der Name des Prinzips: Abhängigkeiten-Umkehrprinzip.

11.8 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, ...

- was die Aufgabe des objektorientierten Entwurfs ist.
- welche Qualitätskriterien einen guten von einem schlechten Entwurf unterscheiden.
- was mit Kohäsion und Kopplung gemeint ist.
- welche Entwurfsprinzipien dabei helfen, die Qualität eines Entwurfs zu verbessern.
- wofür der Begriff SOLID steht und wie die einzelnen Prinzipien darin umgesetzt werden.

Aufgaben

In einem Unternehmen sind Informationen zu allen Mitarbeitern in einer Text-Datei abgelegt:

```
1 last_name, first_name, date_of_birth, email
2 Doe, John, 1982/10/08, john.doe@foobar.com
3 Ann, Mary, 1975/09/11, mary.ann@foobar.com
```

Schreiben Sie eine Konsolen-Anwendung in C#, die täglich eine E-Mail mit Geburtstagsgrüßen an diejenigen Mitarbeiter verschickt, die an diesem Tag Geburtstag haben.

- Berücksichtigen Sie dabei die SOLID Entwurfsprinzipien.

Erweitern Sie das Programm so, dass diejenigen Mitarbeiter, die an einem 29. Februar Geburtstag haben und es diesen Tag im Jahr nicht gibt, die Mail am Tag zuvor erhalten.

- Berücksichtigen Sie dabei das Prinzip der Offen- und Verschllossenheit.

12 Entwurfsmuster

12.1 Entwurfsmuster

Entwurfsmuster

Die in der Entwurfsphase eingesetzten Muster werden als **Entwurfsmuster** bezeichnet.

- Sie zielen darauf ab, die Architektur der Anwendung positiv zu beeinflussen, so dass die Kohäsion steigt und die Koppelung sinkt.

Für große Verbreitung von objektorientierten Entwurfsmustern hat dabei das Buch *Design Patterns - Elements of Reusable Object-Oriented Software* gesorgt, siehe: [2].

- Die Autoren (Erich Gamma, Richard Helms, Ralph Johnson und John Vlissides) werden heute auch als die *Gang-of-Four* (GoF) bezeichnet.

In dem Buch werden 23 Muster in drei unterschiedliche Kategorien unterschieden:

- Die **erzeugende Muster** befassen sich mit der Erzeugung von Objekten aus Klassen.
- **Strukturmuster** fassen Objekte zu gewissen statischen Strukturen zusammen.
- **Verhaltensmuster** beschreiben, wie Objekte durch Zusammenwirken ein bestimmtes Verhalten erzeugen können.

12.2 Fabrik und Einzelstück

Objekterzeugung

Objekte werden mit Hilfe des **new** Operators aus den Klassen erzeugt.

- Dazu wird dann der Konstruktor der Klasse aufgerufen, der das Objekt initialisieren kann.
- Dabei kann es aber zu Problemen kommen, die ein Konstruktor allein nicht lösen kann:

Ein Konstruktor kann nur genau den Typ erzeugen, der durch die Klasse festgelegt ist.

- Ein Konstruktor der Klasse *Person* kann nur ein *Person*-Objekt erzeugen.
- Mitunter soll aber in Abhängigkeit der übergebenen Parameter ein Subtyp geliefert werden, z.B. *Student*.

Die Erzeugung eines Objekts kann zudem recht kompliziert sein.

- Diese Mechanik dann im Konstruktor abzubilden, verstößt gegen das Prinzip der einzigen Verantwortung.

Fabrik

Diese Probleme können durch das Entwurfsmuster **Fabrik (engl. Factory)** gelöst werden.

- Die Fabrik ist ein erzeugendes Muster, es befasst sich mit der Erzeugung von Objekten.

Eine Fabrik wird durch eine eigene Klasse abgebildet.

- Der Name der Klasse sollte das Wort *Fabrik* im Namen tragen, z.B. *PersonFactory*.
- Eine solche Fabrikklasse bietet eine oder mehrere, meist statische Methoden an, die Objekte erzeugen und zurückliefern.

Der Rückgabetyt dieser Methoden ist oft eine Abstraktion.

- Also eine Schnittstelle oder eine (abstrakte) Basisklasse.
- Dadurch können unterschiedliche Typen zurückgeliefert werden.

Beispiel

Im folgenden Beispiel soll aus einem übergebenen Text ein passendes Objekt erzeugt werden.

```
1 class PersonFactory
2 {
3     public static Person GetPerson(string data)
4     {
5         var elements = data.Split(';');
6         if (elements.Count == 3)
7             return new Student(elements[0], elements[1], Convert.ToInt32(elements[2]));
8         else
9             return new Person(elements[0], elements[1]);
```

```

10     }
11 }

```

Je nachdem, wie die Parameter beschaffen sind, werden unterschiedliche Typen erzeugt:

- `var obj1 = PersonFactory.GetPerson("Achim;Schäfer");`
- `var obj2 = PersonFactory.GetPerson("Elvira;Üztürk;1234567");`
- In diesem Fall ist `obj1` eine Person, `obj2` jedoch ein Student.

Einzelstück

Das **Einzelstück** (*engl. Singleton*) ist ebenfalls ein erzeugendes Entwurfsmuster.

- Es sorgt dafür, dass zur Laufzeit einer Anwendung nur ein einziges Objekt einer bestimmten Klasse existieren kann.
- Es nutzt dazu eine Fabrikmethode, liefert aber immer dasselbe Objekt zurück.
- Das Einzelstück ist in vielen Szenarien hilfreich, z.B. beim Logging oder einer Datenbankverbindung.

Um sicherzustellen, dass nur ein einziges Objekt einer Klasse existieren kann, darf der Konstruktor der Klasse nicht öffentlich sein.

- Das einzige Exemplar der Klasse wird dann über eine statische Fabrikmethode beschafft.

Beispiel

```

1  class DatabaseConnectionAsSingleton
2  {
3      private static DatabaseConnectionAsSingleton instance = null;
4
5      private DatabaseConnectionAsSingleton()
6      {
7          // Hier kann das Objekt noch weiter initialisiert werden
8      }
9
10     public static DatabaseConnectionAsSingleton Instance
11     {
12         get
13         {
14             if (instance == null)
15                 instance = new DatabaseConnectionAsSingleton();
16
17             return instance;

```

```
18     }  
19 }  
20 }
```

Nutzung des Einzelstücks

Ein Objekt der Klasse *DatabaseConnectionAsSingleton* kann nun nicht mehr mit dem Operator `new` erzeugt werden.

- Der Compiler würde wegen des privaten Konstruktors einen Fehler melden.
- Das einzige Objekt der Klasse wird über die Fabrikmethode *Instance* beschafft:
- `var instance = DatabaseConnectionAsSingleton.Instance;`
- Auf dem Objekt *instance* können dann alle öffentlichen Methoden aufgerufen werden.

Achtung: Die Nutzung eines Einzelstücks verbirgt Abhängigkeit im Code.

- Abhängigkeiten sollten aber immer explizit sein.
- Hängt *A* von *B* ab, sollte man *B* an *A* übergeben müssen.
- Entsprechend wird davor gewarnt, das Einzelstück übermäßig einzusetzen.
- Stattdessen sollte besser *Dependency Injection* eingesetzt werden.

12.3 Iterator

Iterator

Das Entwurfsmuster **Iterator** ist ein Beispiel eines Verhaltensmusters.

- Der Iterator wird im Zusammenhang mit Datenstrukturen eingesetzt.
- Bekanntlich können Datenstrukturen ihre Daten auf ganz unterschiedliche Weise organisieren, z.B. in einem Array, als einfach-verkettete Liste, als Baum, ...

Der Iterator abstrahiert von dieser Organisation.

- Egal, wie die Daten abgelegt sind, mit dem Iterator kann man über die Elemente iterieren.

Im .Net-Framework wird der Iterator über die Schnittstelle *IEnumerator* abgebildet.

- Die Klasse, welche diese Schnittstelle implementiert, muss die Eigenschaft *Current* und die Methoden *MoveNext()* und *Reset()* anbieten.

Beispiel

Zu unserer einfach-verketteten Liste können wir nun eine eigene Iterator-Klasse konstruieren.

- Die Klasse *LinkedListEnumerator* implementiert die Schnittstelle *IEnumerator*.
- Sie erhält im Konstruktor ein Objekt einer einfach-verketteten List.
- Die wichtigste Methode *MoveNext()* schiebt eine Art Zeiger-Objekt durch die Liste.

```

1 public class LinkedListEnumerator : IEnumerator
2 {
3     private LinkedList list;
4     private ListNode current;
5
6     // Der Rest fehlt hier
7
8     public bool MoveNext()
9     {
10         current = current != null ? current.next : null;
11         return current != null;
12     }
13 }

```

IEnumerable

Datenstrukturen, zu denen ein Iterator existiert, implementieren im .Net-Framework die Schnittstelle *IEnumerable*.

- Die einzige Methode *GetEnumerator()* liefert dann einen entsprechenden *Iterator* zurück

```

1 public IEnumerator GetEnumerator()
2 {
3     return new LinkedListEnumerator(this);
4 }

```

So ausgestattet kann die einfach-verkettete Liste dann auch in einer **foreach**-Schleife genutzt werden:

```
1 var list = new LinkedList();
2 list.pushBack(1);
3 list.pushBack(2);
4
5 foreach (object o in list)
6     Console.WriteLine(o);
```

12.4 Strategie und Besucher

TicTacToe

Erinnern wir uns an unser TicTacToe-Spiel.

- Unser Entwurf sollte offen für Erweiterungen, aber verschlossen für Änderungen sein.
- Daher haben wir mit einer abstrakten Basisklasse bzw. einer Schnittstelle eine Abstraktion für die unterschiedlichen Arten von Spielern geschaffen.
- So können zur Laufzeit dann unterschiedliche Arten von Spielern gegeneinander antreten.
- Zudem können neue Arten von Spielern auch nachträglich noch hinzugefügt werden, ohne das Spiel selbst verändern zu müssen.

Diese Herangehensweise ähnelt dem Entwurfsmuster **Strategie (engl. strategy)**, siehe [14].

- Dieses Verhaltensmuster sorgt dafür, dass in einem Kontext ein Algorithmus durch einen anderen austauschbar wird.

Routenplaner

Stellen wir uns vor wir müssten einen Routenplaner erstellen.

- Der Routenplaner kann aber unterschiedliche Routen berechnen, z.B. die schnellste oder die kürzeste Route.

Wir müssen also unterschiedliche Strategien vorgeben können.

- Über eine Schnittstelle definieren wir eine Abstraktion dieser Strategien.

```

1 public interface IRouteStrategy
2 {
3     Route GetRoute();
4 }

```

Wir können dann unterschiedliche solcher Strategien implementieren.

- z.B. die Klasse *FastestRoutingStrategy* bzw. die *ShortestRoutingStrategy*.

Strategien anwenden

Der Routenplaner bekommt genau eine konkrete Strategie übergeben.

- Er nutzt diese, um die Route zu berechnen:

```

1 var fastest = new FastestRoutingStrategy();
2 var planer = new Routeplaner(fastest);
3 var route = planer.ExecuteStrategy();
4 Console.WriteLine(route.Path + " --> " + route.Duration);
5
6 // Eine andere Routing-Strategie erzeugen:
7 planer.Strategy = new ShortestRoutingStrategy();
8 route = planer.ExecuteStrategy();
9 Console.WriteLine(route.Path + " --> " + route.Duration);

```

Dokument

Stellen wir uns vor, wir müssten die Struktur eines Textdokuments auf Klassen abbilden.

- Ein Dokument besteht aus eine Menge von einzelnen Bestandteilen.
- Jeder Bestandteil kann z.B. normaler Text, Text in Fettdruck, oder ein Hyperlink sein.

Eine solche Struktur könnte z.B. wie folgt aussehen (siehe [20]):

```

1 public abstract class DocumentPart {
2     public string Text { get; set; }
3 }
4
5 public class PlainText : DocumentPart { }
6 public class BoldText : DocumentPart { }
7 public class Hyperlink : DocumentPart {
8     public string Url { get; set; }
9 }
10

```

```
11 public class Document {  
12     public List<DocumentPart> Parts { get; private set; } = new List<DocumentPart>();  
13 }
```

Dokumentenformate

Mit Hilfe dieser objektorientierten Klassenstruktur können nun Dokumente als Objekte abgebildet werden:

```
1 var doc = new Document();  
2 doc.Parts.Add(new BoldText("Überschrift"));  
3 doc.Parts.Add(new PlainText("Dies ist ein normaler Textabschnitt"));  
4 doc.Parts.Add(new Hyperlink("Dies ist ein Link", "https://www.hshl.de"));
```

Diese Datenstruktur wollen wir oft in unterschiedlichen Formaten ausgeben können.

- z.B. als Html- oder als Latex-Dokument.

Ein einfache Herangehensweise wäre dann, alle Klassen mit entsprechenden Methoden auszustatten, z.B. *ToHtml()*:

```
1 // So könnte z.B. die Methode ToString() für die Klasse Hyperlink aussehen:  
2 public string ToHtml() {  
3     return "<a href=\" + docPart.Url + \">\" + docPart.Text + "</a>";  
4 }
```

Besucher

Für jedes neue Datenformat müssten wir die bestehenden Klassen *Document*, *PlainText*, *BoldText* und *Hyperlink* um zusätzliche Methoden erweitern.

- Ein Entwurf sollte aber für Modifikationen verschlossen, für Erweiterung aber offen sein (Prinzip der Offen- und Verschlossenheit).

Zusätzliche Dokumentenformate sollten also als eigene Klassen implementiert werden.

- Dazu bietet sich das Entwurfsmuster **Besucher** (engl. **Visitor**) an.
- Dieses Verhaltensmuster dient der Kapselung von Operationen, die auf Elementen einer Objektstruktur angewandt werden sollen.

Für jedes Datenformat wird dabei eine Besucherklasse eingeführt, die eine bestimmte Schnittstelle implementiert.

- Die Elemente der Datenstruktur, in unserem Fall die Dokumentenbestandteile, können von diesen Besucherobjekten dann besucht werden.

IDocumentConverterVisitor

Die Schnittstelle *IDocumentConverterVisitor* definiert, welche Methoden eine Besucherklasse implementieren muss, um die Datenstruktur in ein konkretes Format zu überführen:

```

1 public interface IDocumentConverterVisitor
2 {
3     void Visit(PlainText docPart);
4     void Visit(BoldText docPart);
5     void Visit(Hyperlink docPart);
6 }

```

Ein konkreter Besucher implementiert nun diese Schnittstelle:

```

1 public class HtmlDocumentConverter : IDocumentConverterVisitor
2 {
3     // Der Rest fehlt hier...
4
5     public void Visit(Hyperlink docPart)
6     {
7         output.Append($"<a href=\"{docPart.Url}\">{docPart.Text}</a>");
8     }
9 }

```

Besucher empfangen

Die Elemente der Dokumentenstruktur müssen nun in der Lage sein, einen Besucher zu empfangen.

- Dazu bieten sie die Methode *Accept* an, die als Parameter ein Objekt vom Typ *IDocumentConverterVisitor* empfängt.

```

1 public class Document
2 {
3     public List<DocumentPart> Parts { get; private set; } = new List<DocumentPart>();
4
5     public void Accept(IDocumentConverterVisitor visitor)
6     {
7         foreach (var part in Parts)
8             part.Accept(visitor);
9     }
10 }

```

Besucher benutzen

Der *HtmlDocumentConverter* kann die Dokumentenstruktur nun besuchen und dies in ein Html-Dokument konvertieren.

```
1 var doc = new Document();
2 doc.Parts.Add(new BoldText("Überschrift"));
3 doc.Parts.Add(new PlainText("Dies ist ein normaler Textabschnitt"));
4 doc.Parts.Add(new Hyperlink("Dies ist ein Link", "https://www.hshl.de"));
5
6 var html = new HtmlDocumentConverter();
7 doc.Accept(html);
8 Console.WriteLine("Als Html:\n" + html.ToString() + "\n");
```

Es können zudem leicht neue Klassen eingeführt werden, welche die Schnittstelle *IDocumentConverterVisitor* implementieren.

- Diese bilden dann zusätzliche Formate ab.
- Die bestehenden Klassen der Dokumentenstruktur müssen dazu nicht verändert werden.

12.5 Beobachter

Ereignisse

Objekte stellen anderen Objekten ihre Dienste über Methoden zur Verfügung.

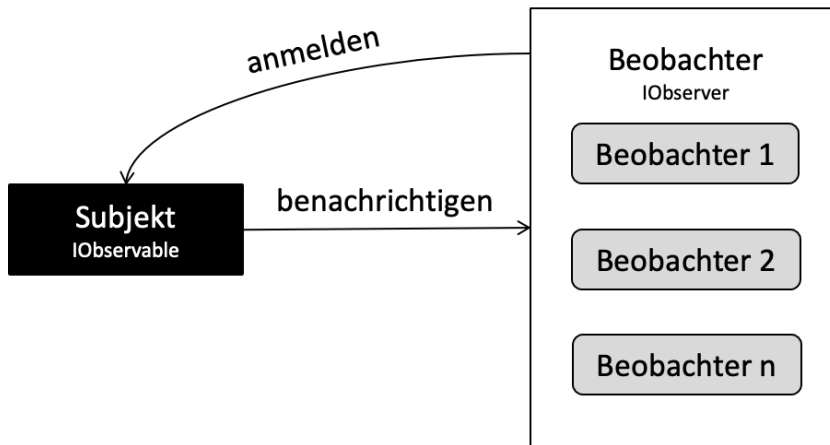
- Dies ähnelt einer klassischen Client-Server-Beziehung.
- Die Interaktion geht dabei immer vom Client-Objekt aus.

Objekte verwalten aber auch ihren eigenen inneren Zustand.

- z.B. der Kontostand eines Bankkontos, oder ob ein Button geklickt wurde.
- Der Zustand kann sich im Prinzip jederzeit ändern.
- Ist ein Client an diesen Änderungen interessiert, ist es wenig performant, den Server immer wieder zu fragen, ob der Zustand sich geändert hat (polling).

Die Zustandsänderung eines Objekts kann man sich als **Ereignis** vorstellen.

- In vielen Situationen ist es hilfreich, wenn Objekte über solche Ereignisse informieren können.



Beobachter

Das Entwurfsmuster **Beobachter** (engl. **Observer**) ist ein Verhaltensmuster.

- Es sorgt dafür, dass Objekte eine Zustandsänderung in Form von Ereignissen melden können.

An dem Muster sind zwei Arten von Objekten beteiligt.

- Das Objekt, welches Ereignisse melden kann, wird als **Subjekt** bezeichnet.
- Ein Subjekt implementiert die Schnittstelle *IObservable*.
- Sie ermöglicht es den Beobachtern, sich für die Benachrichtigung an- und abzumelden.

Ein **Beobachter** wiederum implementiert die Schnittstelle *IObserver*.

- Sie zwingt den Beobachter dazu, eine Methode zu implementieren, die das Subjekt zur Benachrichtigung aufrufen kann.

Beobachter

IObservable und IObserver

Die Schnittstelle *IObservable* muss durch das zu beobachtende Subjekt implementiert werden:

```
1 interface IObservable
2 {
3     void RegistriereBeobachter(IObserver b);
4     void EntferneBeobachter(IObserver b);
5     void BenachrichtigeAlleBeobachter();
6 }
```

Die Schnittstelle *IObserver* wird durch alle Beobachter implementiert:

```
1 interface IObserver
2 {
3     void AenderungIstEingetreten(IObservable quelle);
4 }
```

Bankkonto 1/2

Die Änderungen an einem Bankkonto sollen beobachtbar sein:

```
1 public class Bankkonto : IObservable
2 {
3     private double kontostand;
4     private List<IObserver> beobachter = new List<IObserver>();
5
6     public void RegistriereBeobachter(IObserver b)
7     {
8         beobachter.Add(b);
9     }
10
11     public void EntferneBeobachter(IObserver b)
12     {
13         beobachter.Remove(b);
14     }
15
16     public void BenachrichtigeAlleBeobachter()
17     {
18         foreach (var b in beobachter)
19             b.AenderungIstEingetreten(this);
20 }
```

Bankkonto 2/2

```
1 public double Kontostand
2 {
3     get { return kontostand; }
4 }
5
6 public void Einzahlen(double betrag)
```

```

7      {
8          kontostand += betrag;
9          BenachrichtigeAlleBeobachter();
10     }
11
12     public void Auszahlen(double betrag)
13     {
14         kontostand -= betrag;
15         BenachrichtigeAlleBeobachter();
16     }
17 }

```

Bank

Die Bank hat die Fähigkeit, Änderungen zu beobachten:

```

1 public class Bank : IObservable
2 {
3     public void AenderungIstEingetreten(IObservable quelle)
4     {
5         var konto = quelle as Bankkonto;
6
7         if (konto != null)
8         {
9             Console.WriteLine("Eine Änderung an einem Bankkonto ist eingetreten!");
10            Console.WriteLine("Neuer Kontostand: " + konto.Kontostand);
11        }
12    }
13 }

```

Zusammenspiel

Wir können nun an Objekten der Klasse Bankkonto die Bank als Beobachter anmelden.

- Tritt eine Änderung an einem Konto ein, wird die Bank darüber informiert.

```

1 static void Main(string[] args)
2 {
3     var bank = new Bank();
4     var konto1 = new Bankkonto();
5     var konto2 = new Bankkonto();
6
7     konto1.RegistrierteBeobachter(bank);
8     konto2.RegistrierteBeobachter(bank);
9
10    // Hier wird nun die Bank informiert
11    konto1.Einzahlen(1000);

```

```
12     konto2.Auszahlen(1000);  
13 }
```

Sonstige hilfreiche Muster

Es existieren noch viele weitere Muster, die sich im Alltag der Softwareentwicklung als sehr hilfreich erweisen.

- Ein Profi sollte sich alle Muster der Gang-of-Four einmal genauer angesehen haben.
- z.B. Facade, Proxy, Zustand, Strategie, ...

Neben den Analyse- und Entwurfsmustern existieren zudem auch Architekturmuster.

- Diese helfen dabei, geeignete Strukturen aus Komponenten bzw. Schichten zu schaffen.
- z.B. Model-View-Controller (MVC) oder Model-View-Viewmodel (MVVM).
- Solche Schichten widmen sich dann bestimmten Aufgaben in einem Softwaresystem, z.B. der Benutzerführung (engl. user interface), oder der Datenspeicherung.

Innerhalb dieser Schichten spielen dann auch speziellere Muster eine Rolle.

- z.B. bei der Datenspeicherung: Datentransferobjekt, Repository, ...

12.6 Zusammenfassung und Aufgaben

Zusammenfassung

Wir haben heute gelernt, dass...

- auch in der Entwurfsphase eine ganze Reihe von Mustern hilfreich sind.
- dass bei den Entwurfsmustern zwischen Erzeugungs-, Struktur- und Verhaltensmustern unterschieden wird.
- dass die Fabrik und das Einzelstück Erzeugungsmuster sind.
- wozu der Iterator genutzt wird.
- wie man Datenstrukturen mithilfe des Besuchermusters um zusätzliche Operationen erweitern kann.
- wie Ereignisse mithilfe des Beobachtermusters umgesetzt werden können.

Aufgabe 1

Im Begleitprojekt der Veranstaltung *Algorithmen und Datenstrukturen* ist die Klasse *ArrayList* zu finden¹.

- Erstellen Sie eine geeignete Iterator-Klasse zu dieser Datenstruktur.
- Sorgen Sie dafür, dass die Klasse *ArrayList* die Schnittstelle *IEnumerable* implementiert.

Aufgabe 2

Erstellen Sie eine neue Klasse *ObservableArrayList*, die Sie von der *ArrayList* aus Aufgabe 1 durch Vererbung ableiten.

- Die neue Klasse soll die Schnittstelle *IObservable* implementieren.
- Sobald die Liste geändert wird, z.B. wenn ein neues Element eingefügt wird, sollen die Beobachter über die Änderung informiert werden.
- Realisieren Sie Unit-Tests, um die neue Funktionalität zu teste.

¹<https://github.com/LosWochos76/AUD>

13 Zusammenfassung und Wiederholung

13.1 Grundsätze der Objektorientierten Programmierung

Die Objektorientierte Programmierung

In den letzten Wochen haben wir uns mit einem neuen Programmierparadigma befasst.

- Der **Objektorientierten Programmierung**.

Im Vergleich zu reinen imperativen Programmiersprachen (z.B. C) verändert die objektorientierte Programmierung die Strukturierung von Softwaresystemen.

- Objekte bilden die Grundbausteine objektorientierter Softwaresysteme.
- Objekte verbinden Daten und Operationen zu einer Einheit.
- Objekte prägen Beziehungen untereinander aus, um gemeinsam Aufgaben zu lösen.

Die Objektorientierung macht sich vier Konzepte zu Nutze:

- Datenkapselung, Abstraktion, Vererbung und Polymorphie.

Dadurch kann im Entwurf die Kohäsion erhöht und die Kopplung reduziert werden.

- Ein wichtiger Beitrag, um wartbare und evolvierbare Software zu erhalten.

Datenkapselung

Über Objektvariablen verwalten Objekte ihre eigenen Daten.

- Diese Daten repräsentieren den inneren Zustand des Objekts.
- Ein Objekt schützt diesen inneren Zustand über Methoden vor unerlaubtem Zugriff.

Dieses Vorgehen ist als **Datenkapselung** oder **Geheimnisprinzip** bekannt.

- Objekte grenzen sich gegenüber anderen Objekten ab (Module).

Dieses Vorgehen hat verschiedene Vorteile:

- Da nur die öffentliche Schnittstelle betrachtet werden muss, steigt die Übersichtlichkeit.
- Unerwünschte Interaktionen können vermieden werden (weniger Bugs).
- Ein Objekt kann durch ein Objekt mit einem anderen inneren Aufbau ersetzt werden, sofern sich die öffentliche Schnittstelle nicht ändert.

Abstraktion

Programmierer lieben **Abstraktionen**.

- Eine höhere Programmiersprache wie C ist bereits eine solche Abstraktion.
- Wir müssen uns nicht mehr mit den Details eines konkreten Prozessors herum-schlagen.
- In C können wir z.B. Funktionen einführen, um spezifische Fälle zu parametrisieren.

In der Objektorientierten Programmierung erhalten wir viele weitere Stilmittel, um Abstraktionen herstellen zu können.

- Mit **Klassen** und **Schnittstellen** schaffen wir neue Datentypen.

Eine Klasse dient als Vorlage für gleichartige Objekte.

- Sie definiert die Struktur und das Verhalten seiner Instanzen.

Schnittstellen hingegen definieren gemeinsame Funktionalität über mehrere Klassen hinweg.

Vererbung

Objekte können zur Laufzeit über Objektreferenzen Beziehungen untereinander aufbauen.

- Dadurch können beliebig komplexe Strukturen im Rechner abgebildet werden.

Darüber hinaus erlaubt es die **Vererbung**, eine **Ähnlichkeitsbeziehung** zwischen Klassen und Schnittstellen auszudrücken.

- In einer Vererbungshierarchie gibt das Basiselement (eine Klasse oder eine Schnittstelle) Eigenschaften an die Kindelemente weiter.

Bei Klassen sind dies Eigenschaften und Methodenimplementierungen.

- Bei Schnittstellen werden *lediglich* Methodensignaturen weiter gegeben.

Die Kindelemente können die geerbten Konzepte erweitern oder auch ändern.

Polymorphie

Unter bestimmten Bedingungen können unterschiedliche Objekte als Ersatz füreinander genutzt werden.

- Eine solche *Ersetzbarkeit* kann auf zwei Arten garantiert werden:

Kindklassen bieten (mindestens) dieselben Methoden an, wie die Basisklasse.

- Die Objekte der Kindklasse können also als Ersatz der Objekte der Basisklasse genutzt werden.

Implementieren Klassen eine gemeinsame Schnittstelle, besitzen diese einen Satz gleichartiger Methoden.

- Dort, wo lediglich diese Methoden erwartet werden, können also die Objekte aller implementierender Klassen genutzt werden.

In beiden Fällen können Objekte aber unterschiedliches Verhalten zeigen.

- Dies wird als **polymorphes Verhalten** bezeichnet.

13.2 Überblick über die Veranstaltung

Kapitel

All diese Konzepte wurden in den letzten Kapiteln behandelt.

- Darüber hinaus wurde für C# auch gezeigt, wie diese in einer gängigen und modernen objektorientierten Programmiersprache angewandt werden.

Der Lernstoff war in die folgenden Kapitel aufgeteilt:

1. C# für C-Programmierer
2. Klassen und Objekte
3. Objekte zur Laufzeit
4. Schnittstellen und Aufzählungen
5. Objektbeziehungen
6. Vererbung und Polymorphie
7. Analyse und Analysemuster

8. Entwurf und Entwurfsmuster

Lernziele

Der erfolgreiche Teilnehmer der Veranstaltung, ist in der Lage, ...

- Klassen in C# zu implementieren.
- Klassen mit geeigneten Eigenschaften und Methoden auszustatten.
- Objekte zu erzeugen und zu verwalten.
- Objektbeziehungen aufzubauen.
- komplexe, objektorientierte Systeme so zu entwerfen, dass sie verschiedenen Qualitätskriterien genüge tragen.

Diese Fähigkeiten können natürlich nur dann entwickelt werden, wenn auch die Übungsaufgaben erfolgreich bearbeitet wurden.

13.3 Prüfung und Credits

Prüfung und Credits

Die Prüfungsform ist eine Klausur.

- Wenn nicht anders angekündigt, wird die Klausurzeit 3h betragen.

Der Inhalt der Klausur ist eine Kombination der beiden Veranstaltungen **Objektorientierte Programmierung** und **Algorithmen und Datenstrukturen**.

- Der gesamte Lehrstoff ist klausurrelevant!
- Die Klausuraufgaben werden zu den Übungsaufgaben sehr ähnlich sein.

Bei erfolgreicher Teilnahme werde in ISD 8 ECTS Punkte vergeben.

- In ETR 6 ECTS Punkte.

Als Hilfsmittel ist eine einzelne DIN A4 Seite mit eigenen Notizen erlaubt.

- Doppelseitig beschrieben.

13.4 Abschluss

Zu guter Letzt...

Vielen Dank für Ihre Aufmerksamkeit!

- Ich hoffe, ich konnte Ihnen etwas beibringen!
- Ich hoffe, das Ganze hat auch ein wenig Spaß gemacht!

Viel Glück bei der Klausur!

- Ich hoffe aber, dass Sie das Glück nicht brauchen!

Gute Erholung in der vorlesungsfreien Zeit!

- Wir sehen uns (möglicherweise) im nächsten Semester wieder!
- Da lernen wir (in ISD) neue tolle Dinge!

Literatur

- [1] Douglas Martin. *Book Design: A Practical Introduction*. 1. Dez. 1990.
- [2] Erich Gamma u. a. *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed., Reprint Edition. Reading, Mass: Prentice Hall, 1994. 395 S. ISBN: 978-0-201-63361-0.
- [3] Martin Fowler. *Analysis Patterns, Reusable Object Models*. 1. Aufl. Menlo Park, Calif: Addison-Wesley Longman, Amsterdam, 1996. 357 S. ISBN: 978-0-201-89542-1.
- [4] Robert C. Martin. *Design Principles and Design Patterns*. 2000. URL: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf.
- [5] Jos Warmer und Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2. Aufl. Boston, Mass. Munich: Addison-Wesley Professional, 27. Aug. 2003. 236 S. ISBN: 978-0-321-17936-4.
- [6] Micah Martin und Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. 1. Aufl. Pearson, 20. Juli 2006. 691 S.
- [7] Robert Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. Edition. Upper Saddle River, NJ: Prentice Hall, 1. Aug. 2008. 464 S. ISBN: 978-0-13-235088-4.
- [8] Helmut Balzert u. a. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3. Aufl. 2009 Edition. Heidelberg: Spektrum Akademischer Verlag, 17. Sep. 2009. 642 S. ISBN: 978-3-8274-1705-3.
- [9] Helmut Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl. 2012 Edition. Heidelberg: Spektrum Akademischer Verlag, 13. Sep. 2011. 614 S. ISBN: 978-3-8274-1706-0.
- [10] Helmut Balzert und Heide Balzert. *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. 2. Aufl. 2004 Edition. Heidelberg: Spektrum Akademischer Verlag, 26. Okt. 2011. 592 S. ISBN: 978-3-8274-2903-2.
- [11] Bernhard Rumpe. *Modellierung mit UML: Sprache, Konzepte und Methodik*. 2. Aufl. 2011 Edition. Berlin: Springer, 19. Aug. 2011. 304 S. ISBN: 978-3-642-22412-6.
- [12] Guido Walz, Frank Zeilfelder und Thomas Rießinger. *Brückenkurs Mathematik: für Studieneinsteiger aller Disziplinen*. 3. Auflage. SpringerLink Bücher. Heidelberg: Spektrum Akademischer Verlag, 2011. 392 S. ISBN: 978-3-8274-2764-9. DOI: 10.1007/978-3-8274-2764-9.

Literatur

- [13] Joachim Goll. *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*. 2., aktualisierte Aufl. 2014 Edition. Wiesbaden: Springer Vieweg, 2014. 432 S. ISBN: 978-3-658-05531-8.
- [14] Tiago Martins. *Strategy Pattern - C#*. 10. Nov. 2020. URL: <https://medium.com/@martinstm/strategy-pattern-c-24b8ca1e4a8> (besucht am 17.11.2021).
- [15] Christian Silberbauer. *Einstieg in Java und OOP: Grundelemente, Objektorientierung, Design-Patterns und Aspektororientierung*. 2., akt. u. erw. Aufl. 2020 Edition. Springer Vieweg, 13. Aug. 2020. 168 S. ISBN: 978-3-662-61308-5.
- [16] Christian Ullenboom. *Java ist auch eine Insel: Das Standardwerk für Programmierer. Über 1.000 Seiten Java-Wissen. Mit vielen Beispielen und Übungen, aktuell zu Java 14*. 15. Edition. Rheinwerk Computing, 25. Juni 2020. 1246 S. ISBN: 978-3-8362-7737-2.
- [17] Bernhard Lahres, Gregor Rayman und Stefan Strich. *Objektorientierte Programmierung: Das umfassende Handbuch. Die Prinzipien guter Objektorientierung auf den Punkt erklärt*. 5. Edition. Rheinwerk Computing, 26. Feb. 2021. 688 S. ISBN: 978-3-8362-8317-5.
- [18] Uwe Post. *Besser coden: Best Practices für Clean Code. Das ideale Buch für die professionelle Softwareentwicklung*. 2. Aufl. Bonn: Rheinwerk Computing, 31. Aug. 2021. ISBN: 978-3-8362-8492-9.
- [19] Bill Wagner. *.NET-Dokumentation*. URL: <https://docs.microsoft.com/de-de/dotnet/> (besucht am 24.03.2021).
- [20] Sebastian Krysmanski. *The Visitor-Pattern Explained*. URL: <https://manski.net/2013/05/the-visitor-pattern-explained/>.