Einführung

Viele Lehrbücher beginnen ihre Einführung in die Java-Programmierung mit der Kodierung einer scheinbar einfachen Anwendung:

```
public static void main(String[] args) {
    System.out.println("Hello World!");
}
```

Dieses kleine Programm bewirkt, dass die Zeichenfolge „`Hello World`“ auf dem Bildschirm erscheint.

Wir gehen bewusst anders vor, denn allzu oft vergessen die Programmierer vor der Kodierung ihres Programms, gründlich über die jeweilige Aufgabenstellung nachzudenken und den sozialen, organisatorischen und technischen Kontext, in dem das Anwendungsprogramm später verwendet werden soll, genau zu hinterfragen. Sie wiegen sich sicher in einer vermeintlichen Professionalität, sind aber oft nicht in der Lage, die Fachbegriffe und Abläufe der Anwendungswelt zu verstehen, geschweige denn sich mit ihrer Fachsprache den Auftraggeberinnen und Nutzern, die meist keine IT-Experten sind, verständlich zu machen.

Viele Probleme in der Praxis der Programmentwicklung rühren bekanntermaßen daher, dass Programmentwickler, Auftraggeberinnen und spätere Programmanwender kein ausreichend detailliertes gemeinsames Verständnis der Aufgabe entwickelten. Aus diesem Mangel entstehen unerwünschte Freiheitsgrade bei der Programmierung, die meist zu Enttäuschungen beim Kunden führen.

In dieser Kurseinheit versuchen wir zunächst, Ihnen anhand einer Fallstudie ein intuitives Verständnis der zentralen Begriffe der objektorientierten Programmierung wie Klasse, Objekt, Methode oder Nachricht zu vermitteln. Die Fallstudie dient auch dazu, deutlich machen, dass ein detailliertes Verständnis der gestellten Aufgabe sowie eine präzise Dokumentation notwendige Voraussetzungen für einen gelungenen Programmentwurf sind.

Lernziele

- Die Fachbegriffe eines Anwendungsproblems, ihre Merkmale und typischen Verhaltensweisen verstehen und umfassend beschreiben können.
- Die Anforderungen einer Anwendung erheben und systematisch dokumentieren können.
- Anwendungsfälle, Gegenstände, Beziehungen, Verantwortlichkeiten, Abläufe und andere Phänomene und Erscheinungen von Belang in der Aufgabe erkennen und präzise in der Form eines Fachlexikons und UML-Anwendungsfalldiagrammen darstellen können.
- Die behandelte Aufgabenstellung vervollständigen und auf ähnliche Aufgaben übertragen können.
- Die Begriffe Objekt und Klasse erläutern und gegeneinander abgrenzen sowie auf die Fachbegriffe einer gegebenen Problemstellung anwenden können.
- Verstehen, wie Objekte zusammenwirken.
- Eine Vererbungshierarchie aufstellen und erklären können, wie sich ihre Klassen und das Verhalten dieser Klassen bestimmen lassen.
- Die wesentlichen Merkmale der objektorientierten Programmierung wiedergeben können.
- Die Begriffe Klasse und Objekt und ihre Beziehung zueinander verstehen und anwenden können.
- Die Hauptbestandteile von Klassen und Objekten, nämlich Verhalten und Eigenschaften, erklären und konstruktiv einsetzen können und
- diese in UML-Klassen und Objektdiagrammen darstellen.
- Den Algorithmusbegriff erläutern,
- die grundlegenden Kontrollstrukturen beherrschen,
- sowie ihre Darstellung in UML-Aktivitätsdiagrammen.
- Zudem sollten Sie mit der Übersetzung und Ausführung eines Java-Programms vertraut sein.

1 Von der Aufgabenstellung zum Programm

Mit diesem Kurs wollen wir Ihnen fundierte Kenntnisse von Programmiertechniken, insbesondere der objektorientierten Programmierung, vermitteln. Nach erfolgreichem Abschluss des Kurses sollten Sie in der Lage sein, einfache Systeme in der Programmiersprache Java zu entwickeln und die Qualität überschaubarer Java-Programme zu bewerten.

Da Programme in der Regel dazu dienen, menschliche Tätigkeiten zu unterstützen oder technische Systeme und Anlagen zu überwachen und zu steuern, ist es wichtig, die jeweilige Aufgabenstellung und die Umgebungsbedingungen für das geplante Programm genau zu verstehen.

1.1 Motivation

Um Ihnen den gesamten Entwicklungsprozess von der Aufgabenstellung bis zur Ablieferung eines geprüften Programms anschaulich zu machen, führen wir informelle, aber dennoch methodische Vorgehensweisen ein, die zeigen, wie man:

- eine Aufgabenstellung inhaltlich erfasst und in der Form von Geschäftsvorfällen oder technischen Abläufen, die es zu unterstützen oder zu automatisieren gilt, dokumentiert;
- Geschäftsvorfälle systematisch in einen Programmentwurf überführt und
- derartige Entwurfsdokumente in ausführbare Java-Programme umsetzt.

Bei dieser Vorgehensweise werden Sie grundlegende Programmierkonstrukte wie Abfolge, Schleife, Verzweigung und Rekursion, aber auch wichtige Datenstrukturen wie Liste, Stapel und Baum kennen lernen und benutzen.

Heute gibt es eine Vielzahl unterschiedlicher Programmiersprachen, um Programme zu formulieren. Sie unterscheiden sich jedoch erheblich von den Sprachen, die Menschen benutzen, wenn sie miteinander kommunizieren. Ein entscheidender Unterschied zwischen der Art, wie wir mit einer anderen Person reden und wie der Computer instruiert werden muss, besteht darin, dass Menschen ein großes Maß an Allgemeinwissen und gesundem Menschenverstand besitzen, die es ihnen ermöglichen, auch ungenaue oder unvollständige Aussagen sinnvoll zu interpretieren. Solche Fähigkeiten besitzen Computer nicht. Ihre Anweisungen müssen daher sehr detailliert, Schritt für Schritt und in einer genau festgelegten Sprache, eben der Programmiersprache, angegeben werden.

Allzu oft haben Programmsysteme unvorhergesehene oder unerwünschte Auswirkungen auf die Umgebung, in der sie eingesetzt werden:

- Piloten können bei der Landung den Umkehrschub nicht aktivieren, weil für das Steuerprogramm die Bedingungen einer Bodenberührung nicht erfüllt sind – so geschehen bei der missglückten Landung einer Maschine der Lufthansa in Warschau.
- Mitarbeiter in Organisationen und Unternehmen müssen sich nicht selten an ungewohnte Geschäfts- und Fertigungsabläufe anpassen, weil das neue Programmsystem ohne ausreichende Kenntnis solcher Abläufe geplant wurde.
- Fehlbedienungen werden durch ungewohnte oder unergonomisch gestaltete Bedienoberflächen von Systemen provoziert.

Wir betrachten die Programmierung als eine Ingenieurstätigkeit und legen an sie die gleichen Maßstäbe an wie an andere Technikdisziplinen: Bevor Sie beginnen zu konstruieren, denn Programmieren heißt Konstruieren, müssen Sie

- die Gesetzmäßigkeiten des Anwendungsfelds, die Fachkonzepte und Abläufe verstehen,
- professionelle Arbeitsweisen beherrschen,
- passende Standards kennen (hierzu gehören auch Normen, Richtlinien und Prüfverfahren zur Gestaltung von Benutzungsschnittstellen interaktiver Systeme, die allzu häufig ignoriert werden),
- Qualitäts-, Kosten- und Nutzendenken entwickeln sowie
- empirisches Wissen aufbauen, nutzen und wirksam vermitteln können.

1.2 Softwareentwicklung

Softwareentwicklung

Die Softwareentwicklung umfasst das Entwerfen, Formulieren, Dokumentieren und Überprüfen eines Programms. Sie erschöpft sich demnach nicht im Aufschreiben von Quelltext, sondern schließt eine genaue

- Erhebung der fachlichen Begriffe, Abläufe, Rollen, Gegenstände und Phänomene des jeweiligen Anwendungsbereichs,
- Dokumentation der Anforderungen, die das Programm erfüllen muss,
- Modellierung der Gegenstände und Verfahrensweisen des Anwendungsbereichs und
- umfassende Qualitätsprüfverfahren

mit ein.

Modelle spielen in der Softwareentwicklung eine wichtige Rolle. Für diesen Kurs werden wir die nachfolgende Definition verwenden.

Definition 1.2-1: Modell

Modell
Modellierungssprache

Ein Modell ist die Abstraktion eines Realitätsausschnitts oder eines Systems. Ein Modell wird mit Hilfe einer Modellierungssprache beschrieben.

⌟

Beispiel 1.2-1: Abstraktion in Modellen
Will man beispielsweise ein Modell für ein Universitätsverwaltungssystem entwickeln, so interessiert dabei der Name, der Geburtstag und die Adresse der Studierenden, nicht aber ihre Haarfarbe oder ihre Lieblingsfarbe. Das heißt, in diesem Modell wird von einigen Eigenschaften der Studierenden abstrahiert. ⌟

Eine in der Softwareentwicklung weit verbreitete Modellierungssprache ist die Unified Modeling Language (UML), die wir auch im Rahmen dieses Kurses verwenden werden.

In dieser Kurseinheit werden wir uns mit den einzelnen Schritten von der Idee oder dem Problem hin zum ausführbaren Programm beschäftigen. Dabei unterscheiden wir zwischen Aufgaben, die von Menschen ausgeführt werden müssen, weil sie Kreativität erfordern, und Aktivitäten, die vom Rechner mit Hilfe spezieller Programme, wie zum Beispiel dem Übersetzer (engl. *compiler*), übernommen werden, weil sie mechanischen Charakter haben.

Zunächst wird ein gegebenes Problem untersucht und die an das zu entwickelnde System gestellten Anforderungen identifiziert. Rücksprache mit dem Auftraggeber ist erforderlich, wenn hier Unklarheiten auftreten. In dieser Phase müssen technische Spezialisten mit Experten der Anwendungsdomäne zusammenarbeiten. Hierbei werden zum Beispiel die Anwendungsfälle (Abschnitt 2.3) des zu entwickelnden Systems identifiziert und beschrieben.

Nach der Spezifikation der Anforderungen müssen diese analysiert werden und ein Pflichtenheft erstellt werden. Dieses beschreibt alle wichtigen organisatorischen und technischen Vorgaben (Abschnitt 2.8). Nach der Erstellung des Pflichtenheft wird das System entworfen (Kapitel 3) und das Design schrittweise verfeinert. Dabei werden sowohl die Struktur des Systems als auch sein Verhalten modelliert.

Dieser Entwurf kann dann anschließend entweder von Hand oder zum Teil automatisiert in einer gewünschten Programmiersprache (Kapitel 5) kodiert werden. Dieser Quelltext beschreibt, wie die Software arbeiten soll. Der Quelltext moderner Programmiersprachen ist für einen Menschen lesbar und zugleich geeignet, um automatisch mit Hilfe eines Übersetzers ein ausführbares, also für den Rechner verständliches, Programm zu erzeugen. Das ausführbare Programm ist im Normalfall in einer Maschinensprache verfasst, die für den Menschen nur noch schwer lesbar ist.

Um aber eine gewisse Qualität des ausführbaren Programms zusichern zu können, muss dieses noch getestet werden, bevor es im Anwendungskontext verwendet werden kann. Dafür werden aus der Anforderungsspezifikation, dem Pflichtenheft und dem Entwurf Tests für verschiedene Abstraktionsgrade des Systems festgelegt. Manche dieser Tests müssen manuell durchgeführt werden, andere wiederum können selbst automatisch ausführbare Programme sein. Schlägt ein Tests bei der Ausführung fehl, so muss das entsprechende Element, zum Beispiel der Quelltext,

korrigiert und der Test erneut ausgeführt werden. Es können aber natürlich auch Fehler im Entwurf oder schon in der Anforderungsspezifikation aufgetreten sein.

Mit dem Thema Testen werden wir uns in einer späteren Kurseinheit ausführlicher beschäftigen. Im Rest dieser Kurseinheit werden wir uns mit den verschiedenen Phasen bis hin zum Entwurf beschäftigen und in den nachfolgenden Kurseinheiten mit der Kodierung in Java.

Selbsttestaufgabe 1.2-1:

Ordnen Sie in Abb. 1.2-1 die verschiedenen Artefakte (Rechtecke) und Tätigkeiten (Rechtecke mit abgerundeten Ecken) in den Software-Entwicklungsprozess ein. ◊

1.3 EXKURS: Unified Modeling Language (UML)

Unified Modeling Language (UML)

Die Unified Modeling Language ist eine in der Softwareentwicklung weit verbreitete Modellierungssprache. Sie unterstützt die verschiedenen Phasen des Entwicklungsprozesses. Wir werden im Laufe des Kurses einige Elemente der UML kennen lernen und nutzen.

Entstehung der UML

Seit den 80er Jahren wurden auf verschiedenste Bereiche spezialisierte, objektorientierte Modellierungssprache entwickelt. 1996 rief dann die Object Management Group (OMG), ein Konsortium für Standardisierungen im Bereich der objektorientierten Systementwicklung, auf, Vorschläge für die Spezifikation einer Modellierungssprache einzureichen. Daraufhin reichten 1997 Grady Booch, James Rumbaugh und Ivar Jacobson, die auch vorher schon an eigenen Modellierungssprachen gearbeitet hatten, den Vorschlag für die Unified Modeling Language (UML) ein. Diese wurde in den darauf folgenden Jahren immer weiter bis zur Version 1.5 ergänzt. Da die UML aus verschiedensten Sprachen heraus entstanden ist, ist sie auch sehr vielseitig einsetzbar und ermöglicht die Modellierung verschiedenster Aspekte.

2005 wurde ein Erneuerungsprozess mit der Standardisierung der UML 2 abgeschlossen, die wiederum bis zum Mai 2010 zur Version 2.3 weiterentwickelt wurde. Diese Version werden wir auch in diesem Kurs verwenden.

Was ist die UML?

Bei der Modellierung eines Softwaresystems soll von einem bestimmten Realitätsausschnitt abstrahiert werden. Dabei werden die wesentlichen Eigenschaften des Systems durch eine abstrakte Repräsentation, das Modell, beschrieben. Während der Entwicklung eines Softwaresystems entstehen in der Regel mehrere verschiedene Modelle, z. B. das Analyse- sowie das Entwurfsmodell. Die UML basiert auf

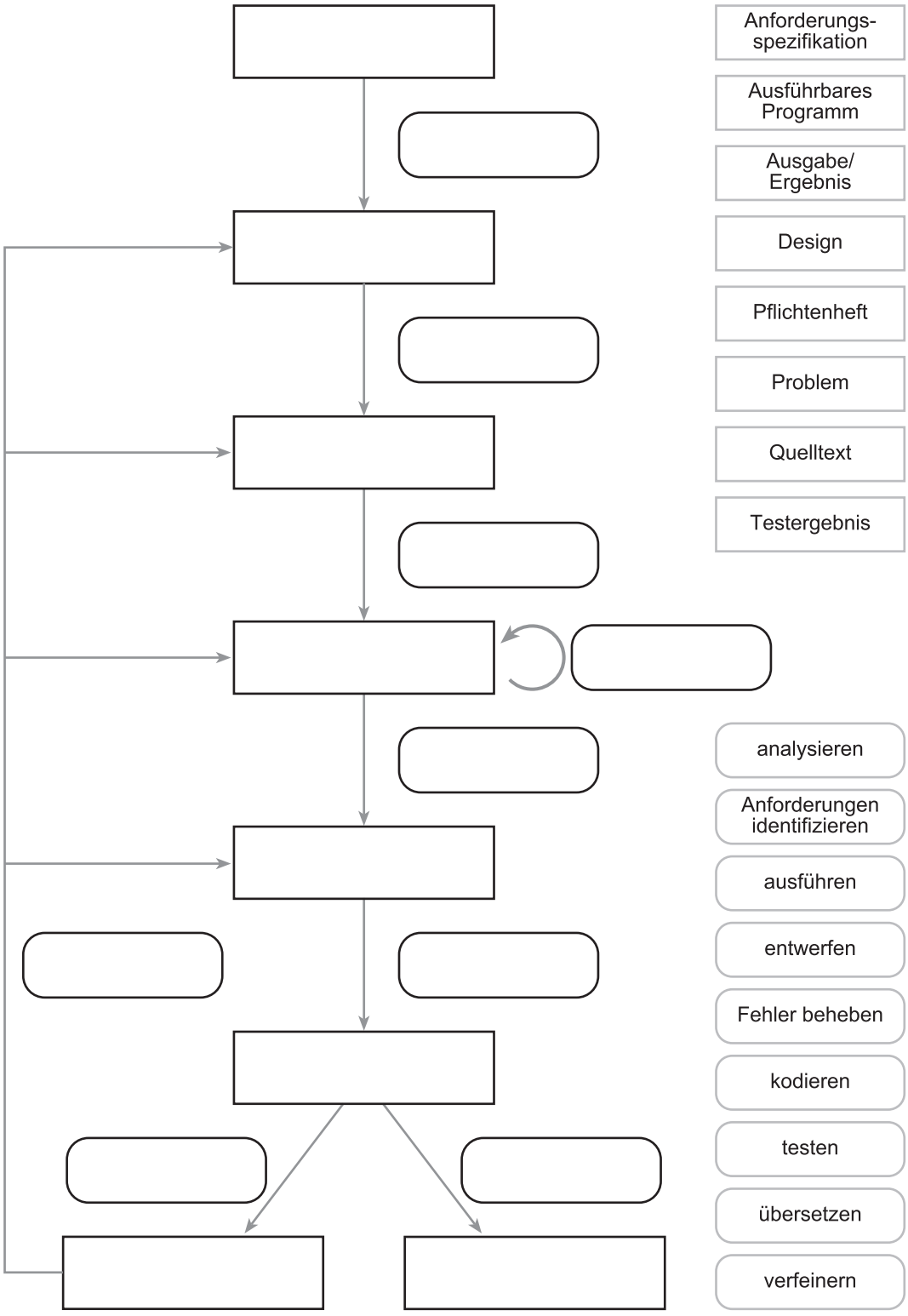

Abb. 1.2-1: Software-Entwicklungsprozess

objektorientierten Konzepten und stellt eine graphische und textuelle Notation für Modelle zur Verfügung und ermöglicht somit das Spezifizieren, Konstruieren, Visualisieren und Dokumentieren eines Systems. Die intuitive graphische Notation soll auch den Austausch und die Diskussion mit den Domänenexperten erleichtern.

Dabei ist zu beachten, dass es sich bei der UML lediglich um eine Notation handelt und nicht um eine Methodik, die beschreiben würde, wie man zu einem Modell des Systems gelangt. Die Vorgehensweise kann somit unabhängig von der Notation festgelegt werden. Jedes Modell beschreibt einen bestimmten Aspekt des Systems und stellt somit eine Sicht auf das System dar. Die UML erzwingt keine bestimmten Vorgehensweisen, keinen Einsatz bestimmter Programmiersprachen oder bestimmter Entwicklungswerkzeuge. Die UML kann auch nicht nur für objektorientierte Modelle genutzt werden, auch wenn sie selbst auf objektorientierten Konzepten basiert.

Die Konzepte der UML sind in Form eines Metamodells festgeschrieben. Das Metamodell ist das Modell, das festlegt, wie UML-Modelle, die von einem Anwender erstellt werden, aussehen können. Dieses Metamodell ist mit einem kleinen Sprachkern der UML beschrieben. Es definiert die graphischen und textuellen Elemente, das Vokabular, die korrekte Form dieser Elemente und deren Beziehungen untereinander, also die Syntax, sowie deren Bedeutung, die Semantik.

Diagramme der UML

Verhaltensdiagramm
Strukturdiagramm

Um die Beschreibung verschiedener Sichten zu ermöglichen, definiert die UML vierzehn Diagrammarten, die wiederum in Verhaltens- und Strukturdiagramme unterteilt werden. Verschiedene Sichten ermöglichen die Modellierung und Betonung verschiedener Aspekte eines Systems. Abb. 1.3-1 zeigt eine Übersicht über die Diagramme und ihre Systematik. Mit den grau markierten Diagrammen werden wir uns im Laufe des Kurses genauer beschäftigen.

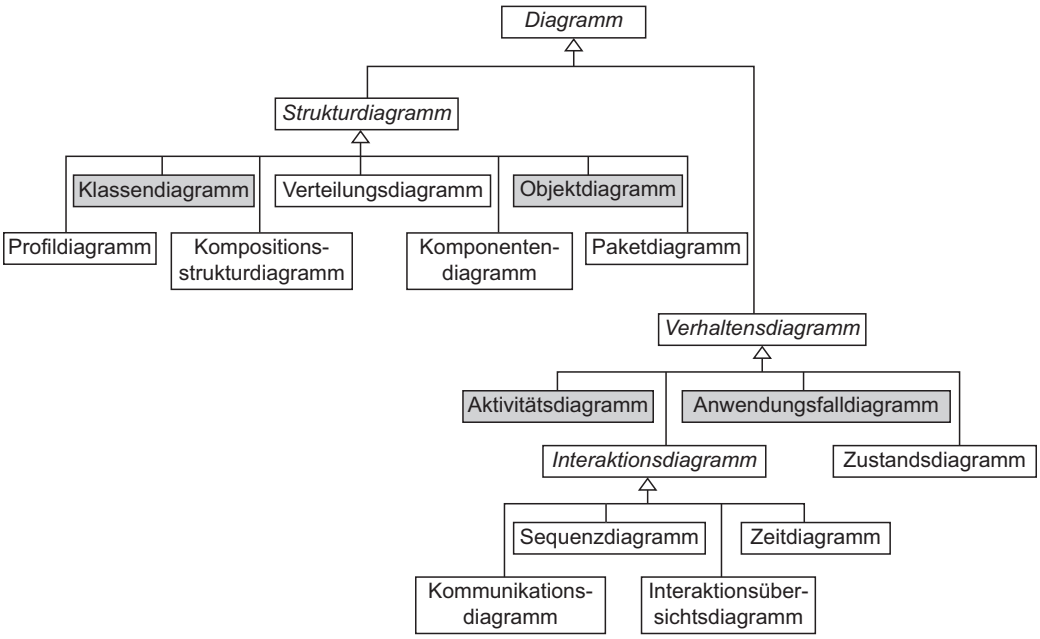

Abb. 1.3-1: Systematik der UML2-Diagrammarten

Die Modellelemente selbst können in Struktur- und Verhaltenselemente unterteilt werden, sowie in Elemente zur Modularisierung und Kommentierung. Diagramme werden in der UML im Normalfall wie in Abb. 1.3-2 dargestellt.

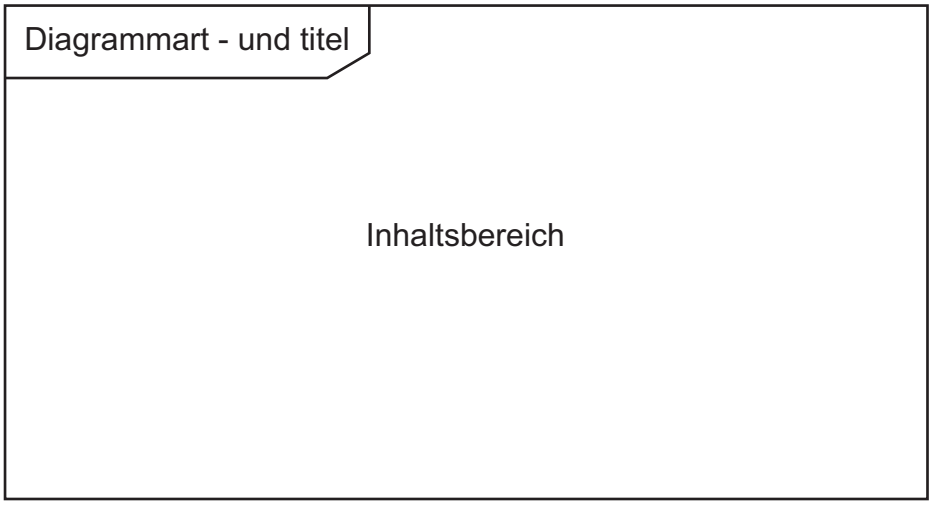

Abb. 1.3-2: Aufbau eines UML-Diagramms

Dabei gibt der Titel links oben den Namen und die Art des Diagramms an. Die Modellelemente selbst werden im Inhaltsbereich angeordnet. Manche Elemente werden auch direkt am Rahmen des Diagramms angeordnet. Beinhaltet das Diagramm keine solchen Elemente, so können Rahmen und Titel auch weggelassen werden.

Weiterführende Literatur

[Hitz05] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, Werner Retschitzegger:
UML @ Work. Objektorientierte Modellierung mit UML 2
Dpunkt Verlag, 3. (aktualisierte) Auflage, 2005

[OMG10] OMG (The Object Management Group):
OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3[6]
OMG, 2010

6 http://www.omg.org/spec/UML/2.3/

2 Anforderungsanalyse

Ausgangspunkt der Programmierung ist die Erschließung ausführlicher Informationen über Aufgaben, Arbeitspraktiken und Entwurfsoptionen aus Sicht der Benutzer eines Programmsystems. Erprobte Vorgehensweisen zur Bewältigung dieser Aufgabe benutzen Fragebögen, Interviewtechniken oder Aufgabenanalysemethoden, um einen möglichst umfassenden Satz an Anforderungen zu erhalten.

Erst wenn die Anforderungen an das zu entwickelnde System und die Umgebungsbedingungen, unter denen es ablaufen soll, verstanden und dokumentiert sind, ist es sinnvoll, mit der Gestaltung des Programmaufbaus im Großen und der Implementierung konkreter Programmeinheiten zu beginnen.

2.1 Fallstudie

Weil es schwierig ist, abstrakte Theorie ohne konkrete Beispiele zu verstehen, wollen wir diese Vorgehensweise an Hand einer einfachen Fallstudie untersuchen.

Nehmen wir dafür als Rahmen den Laden *Adas Blumenland* und stellen uns vor, dass dort bisher ohne Verwaltungssystem gearbeitet wurde. In Zukunft sollen die verschiedenen Geschäftsprozesse wie Verkauf, Lieferung und Bestellungen durch ein Computersystem unterstützt werden. In dem Laden arbeiten neben der Chefin Ada König noch mehrere Verkäufer. Das Sortiment umfasst neben diversen Pflanzen auch verschiedenes Zubehör und Dekorationsartikel.

Zusätzlich zu regulären Kunden gibt es erfasste Premiumkundinnen, die eine Kundennummer besitzen. Sowohl die Verkäufer als auch Frau König bedienen die Kunden im Laden und können Premiumkundinnen erfassen. Alle können direkt im Laden einkaufen und dabei bar oder mit Kreditkarte und bei Beträgen ab 100 Euro auch per Rechnung bezahlen. Wird eine solche Rechnung nicht innerhalb von 14 Tage bezahlt, so wird von der Chefin eine Mahnung versendet. Premiumkundinnen hingegen können auch telefonisch oder im Laden eine Lieferung in Auftrag geben. Bei einer Lieferung ist immer eine Bezahlung per Rechnung möglich. Die Ware wird dann per Kurier kostenlos geliefert werden. Die Rechnung muss vor der Lieferung beglichen worden sein.

Das Lager des Ladens muss auch regelmäßig aufgefüllt werden. Dies kann entweder durch regelmäßige Überprüfung auffallen oder wenn ein Verkäufer auf Grund eines Kundenwunsches nachschaut und feststellt, dass der Artikel nicht in ausreichender Stückzahl vorhanden ist. Die Bestellungen beim Großlieferanten können nicht durch die Verkäufer sondern nur durch die Chefin vorgenommen werden. Ein Verkäufer kann lediglich das frühest mögliche Lieferdatum erfragen. Von der Chefin können auch neue Artikel ins Sortiment aufgenommen oder bisherige entfernt werden.

2.2 Analyse der Anwendungswelt

Jede Beschreibung der Realität hängt von der Wahl der Fachbegriffe sowie von einer vorsichtigen Erläuterung der Phänomene ab, die jeder Begriff bezeichnet. Wichtig ist, dass wir uns vor dem Entwurf eines Programmsystems ein genaues Bild der Wirklichkeit verschaffen und dieses auch mit viel Disziplin umfassend und möglichst eindeutig beschreiben.

Sehen wir uns jetzt die Fallstudie noch einmal genauer an, um:

- die Handlungsträger,
- ihr Zusammenwirken und
- ihre für die Geschäftsabläufe wesentlichen Handlungen zu erkennen,
- die Informationen, die sie austauschen, nachzuvollziehen,
- die Gegenstände, mit denen sie umgehen, sowie
- die Beziehungen, die zwischen diesen Gegenständen bestehen, zu beschreiben und
- erste Ideen zu diskutieren, wie die Geschäftsvorfälle durch Computer unterstützt werden können.

Selbsttestaufgabe 2.2-1:

Untersuchen Sie die Fallstudie nach den gerade genannten Kriterien und überlegen Sie auch, was für Probleme bei den einzelnen Geschäftsprozessen auftreten können.

◊

2.3 Anwendungsfälle und Akteure

Zunächst wollen wir das Verhalten des Systems abstrakt dokumentieren, ohne auf die inneren Strukturen und Abläufe einzugehen. Die Geschäftsvorfälle im Anwendungsfeld werden wir in so genannten Anwendungsfällen beschreiben. Anwendungsfälle erfassen die Funktionalität, die von den zukünftigen Nutzern – in unserem Fall Frau König und ihr Personal – benötigt wird.

Definition 2.3-1: Anwendungsfall

Anwendungsfall

Ein Anwendungsfall (engl. use case*) bezeichnet eine typische Interaktion zwischen einem Akteur und einem System aus der Sicht des Akteurs. Ein Anwendungsfall umfasst Folgen von Aktionen, die reguläre, aber auch vom normalen Verhalten abweichende Abläufe beschreiben.*

Anwendungsfälle erfassen genau die Systemaktionen, die für das Zusammenwirken zwischen dem System und den Akteuren erforderlich sind und dazu beitragen, bestimmte Ziele der Akteure zu erreichen.

Anwendungsfälle werden durch Akteure oder Zeitereignisse angestoßen. ⌟

In Anwendungsfällen treten Anwender und andere Systeme, die mit dem betrachteten System interagieren, immer in bestimmten Rollen auf. In diesen Rollen sind sie mit bestimmten Verantwortlichkeiten und Rechten ausgestattet.

Definition 2.3-2: Akteur

Anwender und externe Systeme, die an Anwendungsfällen teilnehmen, aber selbst nicht Teil des zu realisierenden Systems sind, werden als Akteure (engl. actor*) bezeichnet.* ⌟

Akteur

Systemaktionen, die zum Anwender hin nicht sichtbar sind, werden nicht in Anwendungsfällen erfasst.

Beispiel 2.3-1: Anwendungsfälle

Mögliche Anwendungsfälle in unserer Fallstudie sind zum Beispiel die Bestellung bei einem Großlieferanten, die Erfassung einer neuen Premiumkundin oder der Verkauf beliebig vieler Artikel. Diese Anwendungsfälle werden durch die zuständige Mitarbeiterin oder eine Kundin angestoßen. ⌟

Es ist hier wichtig, zwischen Rollen und konkreten Personen wie Frau König zu unterscheiden, denn ein und dieselbe Person kann z. B. die Rolle des Verkäufers oder des Kuriers übernehmen. Umgekehrt können verschiedene Personen zu verschiedenen Zeiten oder auch zur gleichen Zeit als Verkäufer aktiv sein. Diese Individuen sollen in Anwendungsfällen nicht weiter unterschieden werden, da sie in Bezug auf die betrachteten Geschäftsvorgänge alle die gleiche Rolle spielen.

Akteure nehmen daher immer Bezug auf Rollen und nicht auf konkrete Personen. Rolle

Beispiel 2.3-2: Akteure

Mögliche Akteure in unserer Fallstudie sind Großlieferanten, die Chefin, Verkäufer, Kunden und Premiumkundinnen sowie der Kurier. ⌟

Zusätzlich zu den Anwendungsfällen und den Akteuren muss noch festgelegt werden, welche Akteure an welchen Anwendungsfällen beteiligt sind. Dies geschieht mit Hilfe von Assoziationen.

Definition 2.3-3: Beziehung zwischen Akteur und Anwendungsfall

Die Teilnahme eines Akteurs an einem Anwendungsfall wird durch eine Assoziation zwischen beiden ausgedrückt.

Andere Beziehungen zwischen Akteuren und Anwendungsfällen gibt es nicht. ⌟

Assoziation zwischen Akteur und Anwendungsfall

Anwendungsfälle, Akteure sowie ihre Beziehungen lassen sich übersichtlich mit Hilfe von UML-Anwendungsfalldiagrammen darstellen. Anwendungsfalldiagramme sind dafür gedacht, die ermittelten Anforderungen zu dokumentieren, ohne dabei im Detail auf das Verhalten des künftigen Systems einzugehen.

Definition 2.3-4: UML-Anwendungsfalldiagramm

Anwendungsfalldiagramm

Ein UML-Anwendungsfalldiagramm beschreibt grafisch die Zusammenhänge zwischen einer Menge von Akteuren und Anwendungsfällen. Ein solches Diagramm besteht aus:

- *dem System, dargestellt als Rechteck, wobei der Name des Systems im Rechteck steht;*
- *Anwendungsfällen, die als Ellipsen innerhalb des Systems erscheinen und die Bezeichnung des Anwendungsfalls enthalten;*
- *Akteure, die als Strichmännchen außerhalb des Systems dargestellt werden, wobei der Name des Akteurs unter das Symbol geschrieben wird;*
- *Assoziationen zwischen Akteuren und Anwendungsfällen, die durch Linien dargestellt werden;*
- *Beziehungen zwischen Akteuren und Anwendungsfällen untereinander (siehe auch Abschnitt 2.4 und Abschnitt 2.5).*

⌟

Beispiel 2.3-3: Anwendungsfalldiagramm

Abb. 2.3-1 zeigt das System, einige Anwendungsfälle, Akteure und ihre Assoziationen untereinander.

Die Assoziationen geben an, welche Akteure an welchen Anwendungsfällen beteiligt sind.

⌟

2.4 Beziehung zwischen Akteuren

Zusätzlich zu Beziehungen zwischen Akteuren und Anwendungsfällen gibt es Beziehungen zwischen Akteuren. Beziehungen zwischen Akteuren sind keine Handlungsbeziehungen. Eine Generalisierungsbeziehung drückt die Ähnlichkeit zwischen Rollen aus. So können ein allgemeinerer und ein speziellerer Akteur in einigen Anwendungsfällen beispielsweise die gleiche Rolle einnehmen.

Generalisierungsbeziehung zwischen Akteuren

Eine Generalisierungsbeziehung zwischen zwei Akteuren sagt aus, dass der speziellere Akteur anstatt des allgemeineren an Anwendungsfällen teilnehmen kann. Er kann darüber hinaus aber auch an Anwendungsfällen teilnehmen, an denen der Allgemeinere nicht beteiligt ist.

Eine Generalisierung wird als Pfeil zwischen den Akteuren dargestellt, wobei die unausgefüllte Pfeilspitze zum allgemeineren Akteur zeigt.

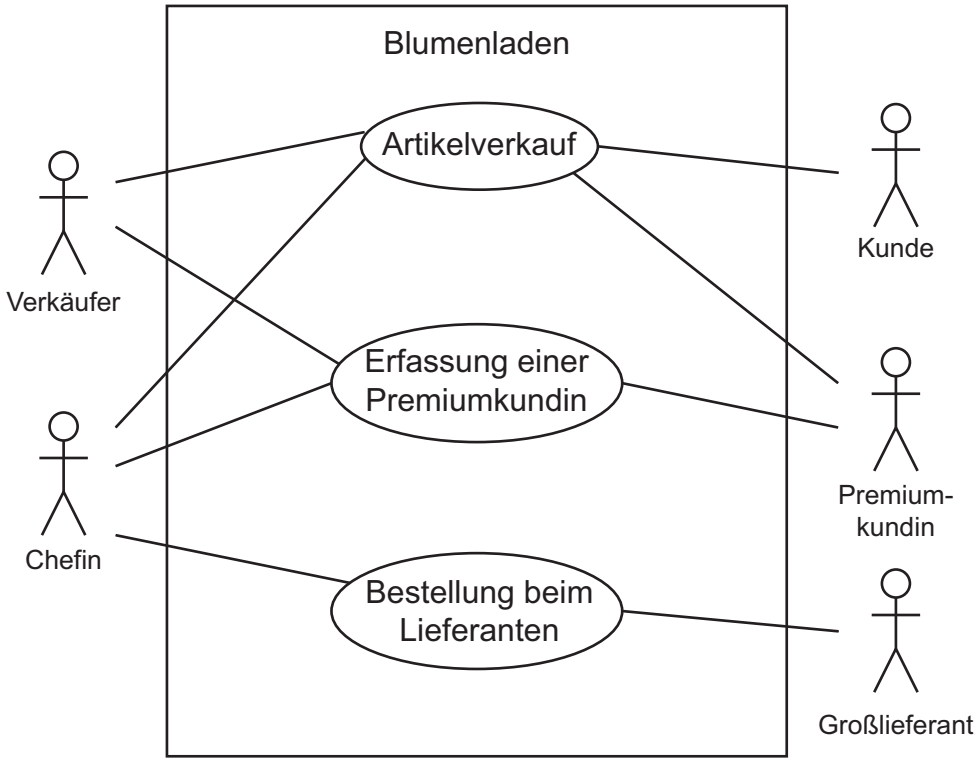

Abb. 2.3-1: Einige Anwendungsfälle, Akteure und ihre Assoziationen

Beispiel 2.4-1:
Wie Abb. 2.4-1 veranschaulicht, kann eine Premiumkundin genauso wie ein Kunde direkt im Laden etwas kaufen, sie kann aber darüber hinaus auch etwas telefonisch bestellen und umsonst nach Hause liefern lassen. Eine Premiumkundin ist somit eine Spezialisierung eines Kunden und ein Kunde eine Generalisierung einer Premiumkundin. Ebenso kann eine Chefin die gleichen Aufgaben wie eine Verkäuferin übernehmen, aber eben auch die Bestellungen beim Lieferanten erledigen. Somit handelt es sich bei einer Chefin um eine Spezialisierung von Verkäuferin. ⌟

2.5 Beziehung zwischen Anwendungsfällen

Sehen wir uns die Anwendungsfälle etwas genauer an, so erkennen wir, dass einzelne Anwendungsfälle Beziehungen zu anderen Anwendungsfällen haben. So kann beispielsweise ein Anwendungsfall das Verhalten eines anderen enthalten.

Es kann aber auch sein, dass bei Anwendungsfällen unter bestimmten Bedingungen ein zusätzliches Verhalten notwendig ist. In diesem Fall wird der Anwendungsfall durch einen anderen erweitert. Dabei ist der erweiterte Anwendungsfall unabhängig vom erweiternden definiert. Zudem kann es wie schon bei den Akteuren auch zwischen Anwendungsfällen Generalisierungsbeziehungen geben.

Wenn ein Anwendungsfall das Verhalten eines anderen mit einschließt, so wird dieses durch eine Enthält-Beziehung ausgedrückt. Dabei zeigt ein gestrichelter Pfeil

Enthält-Beziehung

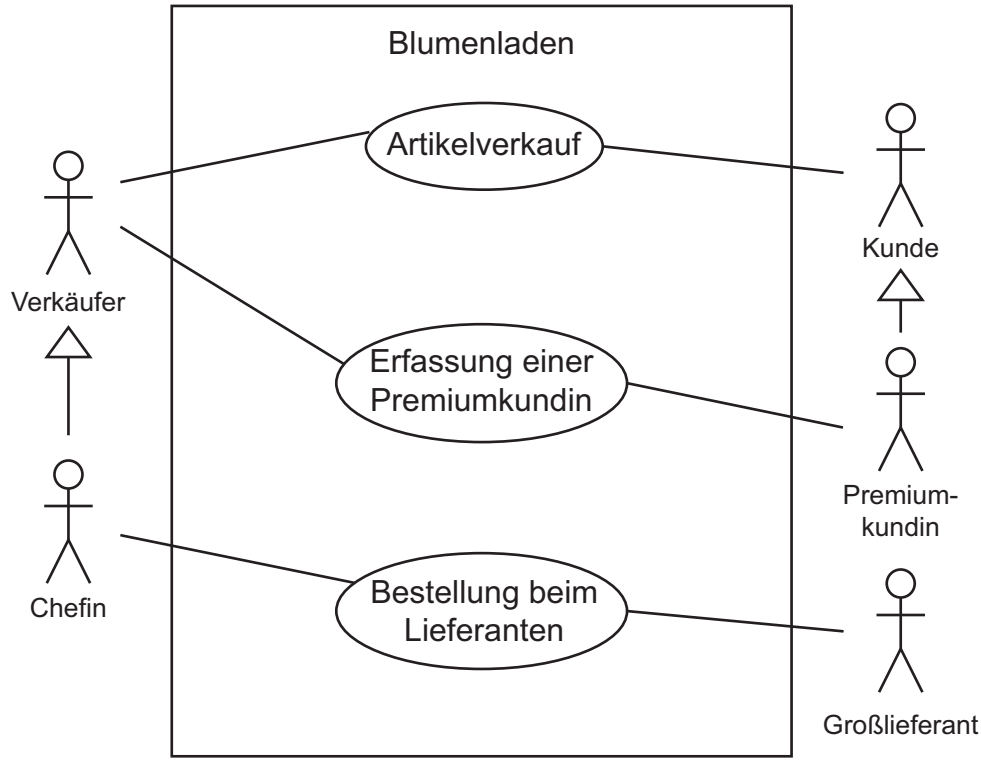

Abb. 2.4-1: Generalisierung und Spezialisierung zwischen Akteuren

mit einer unausgefüllten Spitze auf den Anwendungsfall, der enthalten ist. Der Pfeil ist mit dem Schlüsselwort «include» beschriftet.

«include»

Beispiel 2.5-1: Enthält-Beziehung zwischen Anwendungsfällen
Der Anwendungsfall „Artikelverkauf“ enthält beispielsweise den Anwendungsfall „Bezahlung“.

⌟

Erweitert-Beziehung

Die Erweitert-Beziehung drückt aus, dass ein Anwendungsfall an einer bestimmten Stelle unter bestimmten Bedingungen durch einen anderen erweitert wird. Die Beziehung wird häufig dafür benutzt, optionales Verhalten vom Standardverhalten zu unterscheiden. Dabei zeigt ein gestrichelter Pfeil mit einer unausgefüllten Spitze auf den Anwendungsfall, der erweitert wird. Der Pfeil ist mit dem Schlüsselwort «extend» beschriftet.

«extend»

Beispiel 2.5-2: Erweitert-Beziehung zwischen Anwendungsfällen
Fällt während des Verkaufs eines Artikels auf, dass er gar nicht oder nicht in ausreichender Menge vorhanden ist, so muss eine Nachbestellung erfolgen. Die Bestellung eines Artikels erweitert somit den Verkauf eines Artikels.

⌟

Generalisierungsbeziehung zwischen Anwendungsfällen

Eine Generalisierungsbeziehung zwischen Anwendungsfällen drückt aus, dass der speziellere Anwendungsfall das Verhalten des allgemeineren übernimmt oder anpassen kann. Er besitzt implizit auch die Beziehung zu allen Akteuren des allgemeineren Anwendungsfalls.

Beispiel 2.5-3:
Bisher hatten wir nur den allgemeinen Anwendungsfall Bezahlung. Es sollen jedoch laut Fallstudie Barbezahlung, Bezahlung mit Kreditkarte oder per Rechnung möglich sein. Diese Anwendungsfälle können wir als Spezialisierung von Bezahlung modellieren.

So kann ein Anwendungsfall jedes Bezahlverfahren zulassen. Dieser hat dann eine Enthält-Beziehung zum Anwendungsfall Bezahlung, ein anderer wiederum lässt zum Beispiel nur Barbezahlung zu und besitzt deswegen nur eine Enthält-Beziehung zum Anwendungsfall Barbezahlung.

Da der Anwendungsfall Bezahlung eine Assoziation zum Akteur Kunde hat, besitzen auch alle spezielleren Anwendungsfälle implizit diese Assoziation.

⌟

Abb. 2.5-1 veranschaulicht die verschiedenen Beziehungen zwischen Anwendungsfällen.

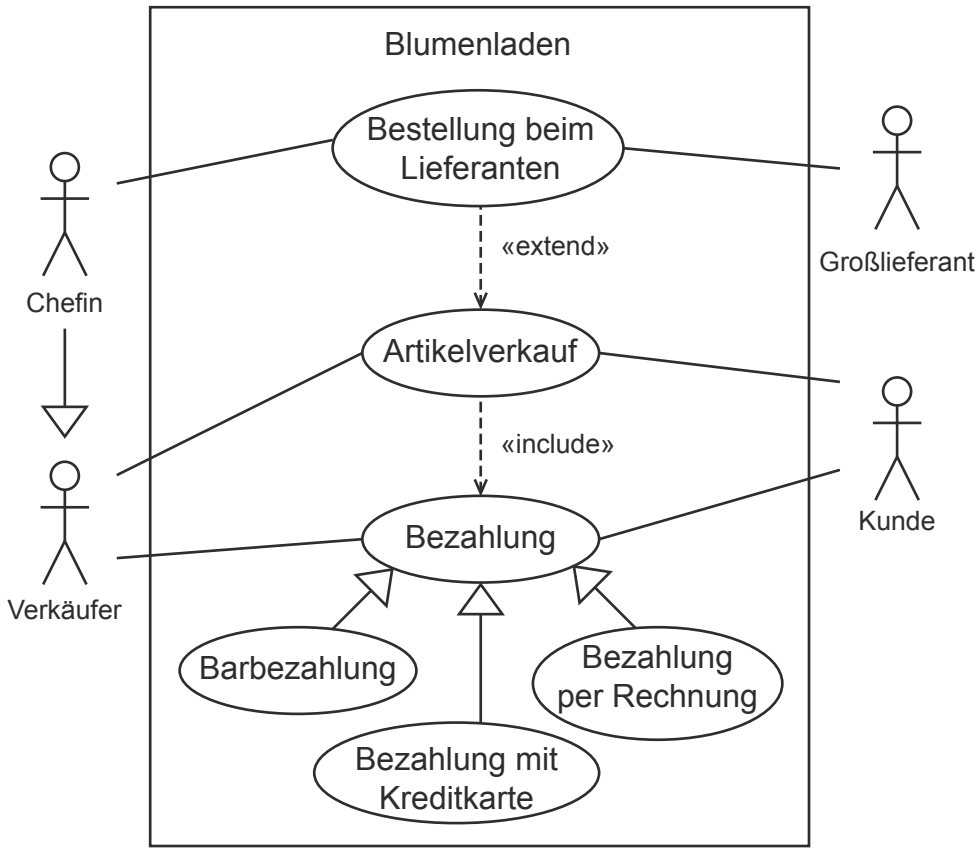

Abb. 2.5-1: Beziehungen zwischen Anwendungsfällen

Selbsttestaufgabe 2.5-1:

Erstellen Sie ein möglichst umfassendes Anwendungsfalldiagramm für die Fallstudie.

◊

2.6 Anwendungsfallbeschreibungen

Der Informationsgehalt der Anwendungsfalldiagramme ist offensichtlich zu gering, um die wesentlichen Aspekte von Anwendungsfällen zu dokumentieren. In diesem Abschnitt führen wir deshalb eine ergänzende Beschreibungstechnik ein.

Um einen gezielten Programmentwurf anzufertigen, müssen die Abläufe, von denen jeder Anwendungsfall abstrahiert, mögliche Abweichungen von Standardabläufen und die Bedingungen, unter denen ein normaler oder abweichender Ablauf angestoßen wird, festgelegt werden.

Definition 2.6-1: Szenario

Szenario

Wir nennen eine einzelne Abfolge von Aktionen, d. h. einen Ablaufpfad in einem Anwendungsfall, ein Szenario.

Ein Szenario, das die Sicht der Anwender wiedergibt, fasst das zu entwickelnde System als Blackbox auf. Die innere Arbeitsweise des Systems bleibt den Anwendern verborgen, von Interesse ist allein das externe Verhalten. Das Augenmerk liegt bei dieser Perspektive auf typischen Interaktionen zwischen Benutzern und dem Anwendungssystem.

⌟

Szenarien spielen in der objektorientierten Programmierung eine wichtige Rolle. Sie tauchen in verschiedenen Entwicklungssituationen auf, etwa bei der Aufstellung der Anforderungen oder bei der detaillierten Bestimmung der Verhaltensspezifikation eines neuen oder eines zu verändernden Anwendungssystems.

Auf einer detaillierteren Betrachtungsebene, zum Beispiel während des Entwurfs, benutzt man Szenarien dazu, die Interaktion zwischen Softwarekomponenten darzustellen. Hier liegt das Augenmerk auf der Beschreibung der Art und Weise, wie die jeweilige Aufgabe unter Verwendung von Arbeitsmitteln und Arbeitsgegenständen erledigt werden kann [Züllighoven98].

Eine tabellarische Darstellung der Szenarien, die alle möglichen Interaktionen zwischen einem Anwendungsfall und den mit ihm kommunizierenden Akteuren beschreiben, ergänzt deshalb die grafische Darstellung eines Anwendungsfalls.

Definition 2.6-2: Beschreibungsschema für Anwendungsfälle

Beschreibungsschema

Für die Beschreibung der in Anwendungsfällen auftretenden Interaktionen werden wir das in Tab. 2.6-1 angegebene Schema verwenden.

Tab. 2.6-1: Beschreibungsschema für Anwendungsfälle

Im folgenden Schema sind nur an den nicht fett angegebenen Stellen entsprechende Eingaben vorzunehmen			
Anwendungsfall		Bezeichnung	
Kurzbeschreibung		Wesentliche Funktionen sowie Bedeutung für den Anwender	
Beteiligte Akteure		Kürzel . . .	Bezeichnung . . .
Vorbedingung		Systemzustand, der für erfolgreiche Ausführung erforderlich ist	
Nachbedingung		Systemzustand, der nach erfolgreicher Ausführung vorliegt	
Auslöser		Ereignisse oder Aktionen, die den Anwendungsfall auslösen, z. B. zeitliche Ereignisse oder auch Ausnahmefälle in anderen Anwendungsfällen	
Standardszenario			
Nr.	Akteur	Kurzbeschreibung der einzelnen Schritte eines erfolgreichen Ablaufs, der entweder mit der auslösenden Aktion oder dem Eintreten der Vorbedingung beginnt und mit dem gewünschten Ergebnis oder dem Eintritt der erwarteten Nachbedingung endet; dabei sollte immer ein Tätigkeitswort verwendet werden	
. . .	. . .	. . .	
Alternative Szenarien			
Nr.	Akteur	z. B. Redefinition oder Alternativen für Schritte im Standardszenario	
. . .	. . .	. . .	
Erweiterungen			
Nr.	Akteur	Erweiterung von Schritten im Standardablauf	
. . .	. . .	. . .	
Fehlersituationen			
Nr.	Akteur	Situationen, die bei diesem Schritt zu einer nicht erfolgreichen Ausführung führen. Beschreibung des Systemzustands nach einer nicht erfolgreichen Ausführung	
. . .	. . .	. . .	

Der schematische Aufriss in Tab. 2.6-1 wird uns helfen, keine wesentlichen Anteile einer Anwendungsfallbeschreibung zu vergessen und die Beschreibungen zu vereinheitlichen.

Beispiel 2.6-1: Anwendungsfall „Bestellung eines Premiumkunden“
Eine konkrete Anwendungsfallbeschreibung für den Anwendungsfall „Bestellung eines Premiumkunden“ ist in Tab. 2.6-2 angegeben. Wir ergänzen dabei den Akteur Kurier, da wir nicht davon ausgehen, dass die Lieferung durch einen Verkäufer erfolgen muss.

Tab. 2.6-2: Anwendungsfall „Bestellung eines Premiumkunden“

Anwendungsfall		Bestellung eines Premiumkunden	
Kurzbeschreibung		Ein Premiumkunde kann ein Bestellung aufgeben und diese dann kostenlos nach Hause liefern lassen. Die Bestellung kann telefonisch oder persönlich erfolgen. Es sind sowohl Barbezahlung als auch Bezahlung mit Kreditkarte oder per Rechnung möglich.	
Beteiligte Akteure		Vk Pk Ku	Verkäufer Premiumkunde Kurier
Vorbedingung		Der bestellte Artikel muss auf Lager oder rechtzeitig bestellbar sein. Der Kunde ist ein Premiumkunde.	
Nachbedingung		Der Artikel wurde an den Kunden geliefert und der entsprechende Betrag bezahlt.	
Auslöser		Anruf eines Kunden oder Besuch des Kunden im Laden	
Standardszenario			
1.	Pk	Der Kunde nennt die gewünschten Artikel persönlich im Laden.	
2.	Vk	Der Verkäufer stellt fest, dass die Artikel in ausreichender Menge auf Lager sind.	
3.	Pk	Der Premiumkunde nennt gültige Kundennummer, einen Liefertermin und Adresse.	
4.	Vk	Verkäufer erfasst die Daten und berechnet den Preis.	
5.	Pk	Der Kunde bezahlt bar (→ Anwendungsfall Barbezahlung).	
6.	Ku	Der Kurier liefert die gewünschten Artikel am definierten Datum.	
Alternative Szenarien			
zu 1.	Pk	Der Kunde nennt die gewünschten Artikel telefonisch.	
zu 5.	Pk	Der Kunde nennt seine Kreditkartennummer am Telefon. (→ Anwendungsfall Bezahlung mit Kreditkarte)	
zu 5.	Pk	Der Kunde legt seine Kreditkarten vor. (→ Anwendungsfall Bezahlung mit Kreditkarte)	

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite		
zu 5.	Pk	Der Kunde zahlt bar beim Eintreffen der Lieferung. (→ Anwendungsfall Barbezahlung)
zu 5.	Pk	Der Kunde bezahlt die Rechnung vor der Lieferung. (→ Anwendungsfall Bezahlung per Rechnung)
Erweiterungen		
zu 2.	Vk	Artikel sind nicht in ausreichender Menge vorhanden, können aber rechtzeitig geliefert werden; die Chefin muss eine Bestellung veranlassen (→ Anwendungsfall Bestellung beim Lieferanten)
Fehlersituationen		
zu 2.	Vk	Artikel ist nicht in ausreichender Menge vorhanden, kann auch nicht rechtzeitig geliefert werden; Anwendungsfall wird abgebrochen, aber Bestellung trotzdem vorgenommen (→ Anwendungsfall Bestellung beim Lieferanten)
zu 3.	Pk	Kunde nennt ungültige Kundennummer; Anwendungsfall wird abgebrochen
zu 5.	Pk	Der Kunde bezahlt nicht rechtzeitig; Lieferung findet nicht statt bzw. wird im Falle einer fehlgeschlagenen Bezahlung vor Ort wieder mitgenommen

⌟

Folgende Fragen sollte man sich stellen, wenn man die Abläufe für einen Anwendungsfall erhebt:

- Warum benutzt ein Akteur das System?
- Welche Art der Antwort erwartet der Akteur von einer Aktion?
- Was muss der Akteur tun, um das System benutzen zu können?
- Welche Information muss der Akteur dem System übermitteln?
- Welche Information erwartet der Akteur als Antwort vom System?

Selbsttestaufgabe 2.6-1:

Erstellen Sie eine Anwendungsfallbeschreibung unter Verwendung des Schemas in Tab. 2.6-1 für den Anwendungsfall „Verkauf“.

◊

2.7 Datenlexikon

In den bisherigen Diagrammen verwendeten wir zahlreiche Fachbegriffe aus dem Anwendungsfeld, die nur vage bestimmt sind. Wir haben z. B. noch nicht darüber geredet, welche Daten ein Eintrag für einen Premiumkunden enthalten soll.

Ein Datenlexikon ist eine nach festen Regeln aufgebaute präzise Bestimmung aller Datenlexikon

Datenelemente, die für das Anwendungsgebiet wichtig sind. Das Datenlexikon trägt dazu bei, Missverständnisse sowie unterschiedliche Interpretationen von Auftraggeberinnen und Entwicklern zu vermeiden. Tab. 2.7-1 gibt die in einem Datenlexikon üblichen Symbole und Schreibweisen an. Kommentare können dafür benutzt werden, umgangssprachlich Definitionsbereiche oder Maßeinheiten anzugeben.

Tab. 2.7-1: Schreibweise im Datenlexikon

=	setzt sich zusammen aus
+	Folge
|	Auswahl
{ }	beliebige Wiederholung
( )	optionale Angabe
**	Kommentar

Nachfolgend ist der Ausschnitt des Datenlexikons für den Eintrag eines Premiumkunden gezeigt (wenn auch aus Vereinfachungsgründen nicht ganz realitätsgetreu).

```
Zeichenkette = Buchstabe + {(Buchstabe | -)}
GanzeZahl = Ziffer + {Ziffer}
Premiumkunde = Vorname + Nachname + Anschrift + Kundennummer
    Vorname = Zeichenkette
    Nachname = Zeichenkette
    Anschrift = Straße + Hausnummer + PLZ + Ort
    Straße = Zeichenkette
    Ort = Zeichenkette
    Hausnummer = GanzeZahl + (Buchstabe)
    PLZ = 00001 | ... | 99999
    Kundennummer = GanzeZahl
```

2.8 Pflichtenheft

Pflichtenheft

In einem professionellen Entwicklungsprojekt wird üblicherweise zwischen Auftraggeberinnen und Softwareentwicklern ein sog. Pflichtenheft aufgestellt. Im Pflichtenheft sind alle wichtigen organisatorischen und technischen Vorgaben zur Erstellung der Software zusammengefasst. Das Pflichtenheft bildet die Grundlage für die technische Realisierung der Software. Das Pflichtenheft dient damit zwei Hauptzwecken:

- Es muss so abgefasst sein, dass es vom Auftraggeber und den Spezialisten in den Fachabteilungen, die die Software später einsetzen sollen, verstanden wird.
- Die erfasste Information muss ebenso für die Softwareentwickler hilfreich sein.

Der angestrebte Sollzustand ist der Hauptbestandteil des Pflichtenheftes. Der Istzustand soll nur dann in das Pflichtenheft mit aufgenommen werden, wenn er zur Verdeutlichung des Sollzustandes beiträgt.

3 Einführung in die Objektorientierung

Objektorientierte Entwurfstechniken und Programmiersprachen werden von vielen Fachleuten aus Hochschule und Praxis bevorzugt, weil sie fachliche und informationstechnische Begriffe vereinheitlichen. Die Begriffe der Anwendung dienen hierbei als Grundlage zur Modellierung einer programmtechnischen Lösung. Das Anwendungsmodell kann ohne gedanklichen Bruch in ein objektorientiertes Programm überführt werden.

Zunächst werden wir uns mit den Konzepten der Objektorientierung vertraut machen, anschließend die Anwendungswelt auf diese Konzepte abbilden und die entstandenen Modelle mithilfe der UML notieren.

Das zentrale Element der Objektorientierung ist, wie der Name schon sagt, das Objekt. Als Beispiele für die verschiedenen Konzepte werden wir Elemente unserer Fallstudie aus Abschnitt 2.1 heranziehen.

3.1 Objekte und Klassen

Definition 3.1-1: Objekt

Objekt

Die objektorientierte Programmierung bezeichnet die Abbilder konkreter individuell unterscheidbarer Gegenstände und Träger individueller Rollen in Entwurfsdokumenten und Programmen als Objekte.

⌟

Beispiel 3.1-1:
Beispiele für Objekte sind in unserer Fallstudie die Kundin Anna Müller, der Kunde Jens Meier oder die Rechnung mit der Nummer 20022.

⌟

Bemerkung 3.1-1:
Wichtig ist, dass es sich bei Objekten um individuelle, identifizierbare Abbilder handelt.

⌟

Bei der Modellbildung vereinfachen wir die Komplexität realer Objekte durch die Konzentration auf wesentliche Eigenschaften und Fähigkeiten. Mit Fähigkeiten meinen wir mögliche Formen des Umgangs mit den Objekten.

Nun wäre es sehr mühsam, wollte man alle Objekte eines Anwendungsbereichs in ihren Eigenschaften und Fähigkeiten einzeln angeben und programmieren. Es sind einfach zu viele, und zahlreiche Objekte weisen auch Ähnlichkeiten auf, die wir bei der Abstraktion nutzen können.

Definition 3.1-2: Klasse

Eine Klasse beschreibt in der objektorientierten Programmierung einen Bauplan für viele ähnliche, aber individuell unterscheidbare Objekte. Klasse

Klassen werden durch Substantive bezeichnet. ⌟

Bemerkung 3.1-2:
Jedes Objekt ist eine Ausprägung einer bestimmtem Klasse. Wir bezeichnen Objekte auch als Exemplare. ⌟

Beispiel 3.1-2:
Mögliche Klassen sind `Kunde`, `Rechnung` *oder auch* `Blume`. *Dabei wären Anna Müller und Jens Meier Objekte der Klasse* `Kunde` *und die Rechnung mit der Nummer 20022 ein Exemplar der Klasse* `Rechnung`. ⌟

Klassen beschreiben die Eigenschaften, die alle Exemplare dieser Klasse besitzen, und auch deren Verhalten und Fähigkeiten.

Definition 3.1-3: Attribut

Die Eigenschaften, die alle Exemplare einer Klasse besitzen, werden durch Attribute dargestellt. Attribut

Attribute werden durch Substantive bezeichnet. ⌟

Beispiel 3.1-3:
Attribute der Klasse `Kunde` *sind* `name` *und* `adresse`. *Eine Rechnung besitzt beispielsweise die Attribute* `rechnungsnummer`, `betrag` *und* `istBezahlt`. ⌟

Attribute beschreiben lediglich, welche Eigenschaften ein Exemplar (Objekt) einer Klasse hat, jedoch nicht welchen konkreten Wert diese Eigenschaft bei dem gegebenen Exemplar besitzt.

Definition 3.1-4: Attributwert

Ein Attributwert bezeichnet den Wert, den ein Exemplar zu einem bestimmen Attribut besitzt. Attributwert

⌟

Beispiel 3.1-4:
Ein Beispiel für einen Attributwert des Attributs `name` *der Klasse* `Kunde` *ist* `Anna Müller`. *Die Nummer* `20022` *ist ein möglicher Attributwert für das Attribut* `rechnungsnummer` *der Klasse* `Rechnung`. ⌟

Bemerkung 3.1-3: Zustand eines Objekts

Zustand

Durch die Attributwerte eines Objekts wird sein innerer Zustand festgelegt. ⌟

Definition 3.1-5: Methode

Methode

Das gemeinsame Verhalten aller Objekte einer Klasse wird durch die Methoden der Klasse bestimmt. Jede Methode beschreibt eine Fähigkeit oder mögliche Form des Umgangs mit einem Objekt der Klasse.

Methoden werden durch Verben bezeichnet und in der Regel durch ein nachstehendes Klammerpaar „`(`“ und „`)`“ abgeschlossen. ⌟

Beispiel 3.1-5:

`ändereAdresse()` *ist eine mögliche Methode für die Klasse Kunde.* `markiereAlsBezahlt()` *ist eine mögliche Methode für die Klasse Rechnung.* ⌟

3.2 Beziehungen

Objekte können Beziehungen zu Objekten der gleichen oder anderer Klassen haben.

Definition 3.2-1: Assoziation und Verknüpfung

Assoziation zwischen Klassen
Verknüpfung

Eine Assoziation zwischen zwei Klassen drückt aus, dass es zwischen Exemplaren dieser Klassen eine Beziehung geben kann. Eine Assoziation besitzt einen Namen. Die konkrete Beziehung zwischen zwei Exemplaren wird Verknüpfung genannt. Zu jeder Verknüpfung zwischen zwei Exemplaren muss es immer eine zugehörige Assoziation zwischen den beiden Klassen geben. ⌟

Beispiel 3.2-1:

Jede Rechnung wird für einen Kunden ausgestellt. Somit hat ein Exemplar der Klasse `Rechnung` *eine Verknüpfung zu einem Objekt der Klasse* `Kunde`*. Zwischen den beiden Klassen* `Kunde` *und* `Rechnung` *existiert die Assoziation* `rechnungsempfaenger`*. Die Rechnung 20022 kann somit eine Verknüpfung zum Kunden Jens Meier besitzen.* ⌟

Wie bei Akteuren und Anwendungsfällen gibt es auch bei Klassen Generalisierung und Spezialisierung. Generalisierung/Spezialisierung ist eins der zentralen Konzepte in der Objektorientierung.

In der Realität kennen wir viele solche Beziehungen. So können wir beispielsweise allgemein von Fahrzeugen sprechen. Allerdings ist es in machen Situtationen notwendig, zwischen PKWs und LKWs zu unterscheiden. PKWs und LKWs sind also spezielle Fahrzeuge. Fahrzeuge können außerdem in Land- und Wasserfahrzeuge

unterschieden werden. Wenn wir das Modell noch weiter ergänzen, erhalten wir eine ganze Hierarchie (siehe Abb. 3.2-1). Solche Hierarchien treffen wir auch häufig in der Biologie an, wenn Tiere oder Pflanzen in einer bestimmten Systematik erfasst werden.

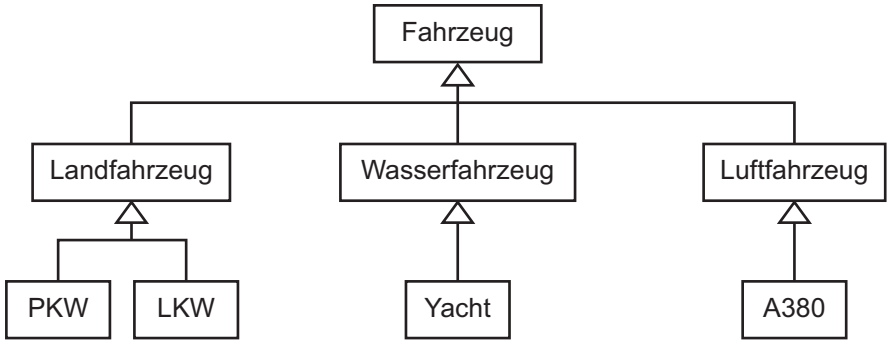

Abb. 3.2-1: Eine Klassenhierarchie für verschiedene Fahrzeugtypen. Die unausgefüllten Pfeilspitzen zeigen jeweils zur allgemeineren Klasse.

Durch solche Spezialisierungen drücken wir aus, dass Klassen gemeinsame Eigenschaften und Verhalten aufweisen. Wir sprechen dann davon, dass die speziellere Klasse die Eigenschaften und das Verhalten der allgemeineren erbt. Dabei kann die speziellere Klasse immer noch neue Eigenschaften und Verhalten hinzufügen.

Manchmal ist es nötig, dass das Verhalten der spezielleren Klassen angepasst wird. Wenn das Verhalten verändert wird, dann sprechen wir davon, dass das Verhalten bzw. die Methode überschrieben wird.

Eine Klasse B kann nur dann eine Spezialisierung der Klasse A sein, wenn gilt, dass jedes Objekt der Klasse B ein A ist.

Beispiel 3.2-2:
Jeder PKW ist ein Fahrzeug, umgekehrt gilt dies jedoch nicht. Genauso wenig gilt die Aussage, dass jeder PKW ein LKW ist.

⌟

Definition 3.2-2: Generalisierung

Generalisierungsbeziehung zwischen Klassen

Eine Generalisierung zwischen zwei Klassen drückt aus, dass die speziellere Klasse die Eigenschaften, das Verhalten und die Beziehungen der Allgemeineren erbt und dabei erweitern oder überschreiben kann. Ein Exemplar der spezielleren Klasse kann überall dort verwendet werden, wo ein Exemplar der allgemeineren Klasse verwendet werden kann.

⌟

Bemerkung 3.2-1:

Oberklasse
Unterklasse
Vererbung

Die allgemeinere Klasse wird in der Regel als Oberklasse und die speziellere als Unterklasse bezeichnet. Der Umstand, dass die Unterklasse von der Oberklasse Eigenschaften und Verhalten übernimmt, wird auch Vererbung genannt.

⌟

Selbsttestaufgabe 3.2-1:

Versuchen Sie, aus der folgenden Beschreibung eine Klassenhierarchie für die im Blumenladen verkauften Artikel zu erstellen:

Der Blumenladen verkauft sowohl Pflanzen als auch Zubehör. Beim Zubehör gibt es unter anderem Blumentöpfe, Vasen, diverse Dekorationsartikel, Grußkarten sowie Blumenerde und Dünger.

Bei den Pflanzen wird zwischen Topfpflanzen, also Außenpflanzen, Zimmerpflanzen und Kakteen, und Schnittblumen unterschieden. Zudem werden auch Sträuße verkauft.

(Attribute und Methoden müssen Sie nicht berücksichtigen.) ◊

In der realen Welt kommt es vor, dass eine Klasse die Spezialisierung mehrerer Klassen ist. So ist ein Amphibienfahrzeug beispielsweise sowohl ein Land- als auch ein Wasserfahrzeug. Die Klasse Amphibienfahrzeug ist somit eine Spezialisierung der Klasse Land- und der Klasse Wasserfahrzeug. In einem solchen Fall spricht man von Mehrfachvererbung.

Definition 3.2-3: Mehrfachvererbung

Mehrfachvererbung

Wenn eine Klasse die Spezialisierung mehrerer anderer Klassen ist, so nennen wir dies Mehrfachvererbung. ⌟

Beispiel 3.2-3:
Wir haben die Hierarchie aus Abb. 3.2-1 um einige Klassen erweitert, bei denen Mehrfachvererbung nötig ist.

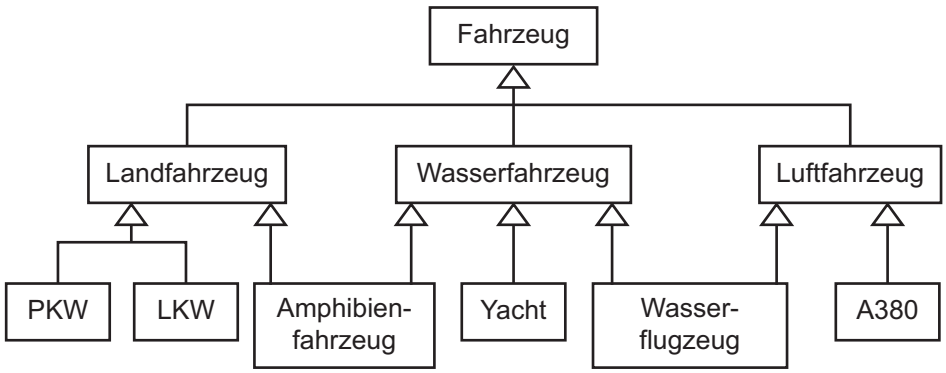

Abb. 3.2-2: Eine Klassenhierarchie für verschiedene Fahrzeugtypen mit Mehrfachvererbung

⌟

Selbsttestaufgabe 3.2-2:

Ordnen Sie die folgenden Begriffe den verschiedenen Konzepten, wie Klasse, Objekt, Attribut, Attributwert und Methode, zu:

Rose, Rechnung, passeBeschreibungAn, Artikel, Preis, Rechnungsnummer, Farbe, überprüfeLagerBestand, Hanna Meier, „Zur Ecke 17, 12345 Irgendwo“, Kunde, Rot, Name, Schnittblume, 31221, ändereName, Adresse

◊

3.3 Kommunikation

In einem objektorientierten Programm existieren viele verschiedene Objekte. Damit das Programm das gewünschte Ergebnis erreichen kann, ist es nötig, dass die Objekte zusammenarbeiten.

Nachricht

Die Zusammenarbeit von Objekten läuft mit Hilfe von Nachrichten ab. Ein Objekt kann eine Nachricht an ein anderes Objekt oder gelegentlich auch sich selbst verschicken. Eine solche Nachricht enthält die Information, welche Methode ausgeführt werden soll, und wenn nötig noch weitere Informationen, die das empfangende Objekt benötigt, um die Nachricht zu verarbeiten. Das versendende Objekt wartet in der sequenziellen Programmierung so lange, bis das empfangende Objekt eine Antwort sendet, und setzt erst dann seine Arbeit fort.

Beispiel 3.3-1:
Stellen wir uns vor, eine Kundin will einen Strauß Blumen kaufen. Durch die Eingabe des Verkäufers wird das für einen Verkauf zuständige Objekt zunächst eine Nachricht an das Lagerobjekt verschicken. Das Lagerobjekt wird aufgefordert zu überprüfen, ob alle benötigten Blumen in ausreichender Anzahl verfügbar sind, und, wenn dies der Fall ist, die benötigten Blumen aus dem Lager zu entnehmen. Anschließend wird ein neues Rechnungsobjekt erzeugt. Dieses erhält die Nachricht, den Blumenstrauß als Rechnungsposten hinzuzufügen. Sodann kann das Rechnungsobjekt nach dem endgültigen Preis gefragt werden und nach erfolgreicher Bezahlung darüber informiert werden.

⌟

Bemerkung 3.3-1: Sequenzielle Programmierung
In diesem Kurs werden wir uns nur mit sequenzieller Programmierung beschäftigen. In der sequenziellen Programmierung wird zu jedem Zeitpunkt des Programms nur eine Aktion ausgeführt. In der nebenläufigen oder parallelen Programmierung können hingegen mehrere Aktionen gleichzeitig ausgeführt werden.

⌟

3.4 Objektorientierte Analyse und Entwurf

Wir haben gelernt, dass wir Gegenstände, die wir als gleichartig sehen, zu Klassen und somit zu einem Begriff zusammenfassen können. Die Klassenbildung ist meist nicht eindeutig. Sie hängt vom Verständnis der beteiligten Personen ab.

Jede Beschreibung der Realität hängt von der Wahl der Fachbegriffe und einer vorsichtigen Erläuterung der Phänomene ab, die jeder Begriff bezeichnet. Wichtig ist, dass wir uns vor dem Entwurf eines Programmsystems ein genaues Bild der Wirklichkeit erschaffen und dieses auch mit viel Disziplin umfassend und möglichst eindeutig beschreiben.

Unsere Wahrnehmung der Wirklichkeit müssen wir nachvollziehbar mit den schematischen und formalen Beschreibungen, die ein Programm ausmachen, in Beziehung setzen. Nur so können wir sicher sein, dass die Auswirkungen des Programms die gewünschten Resultate liefern. In jeder Anwendungswelt werden Begriffe benutzt, die von vorne herein nur selten von Entwicklern voll verstanden werden.

Beispiel 3.4-1: Ungenügende Modellanalyse
Ein Unfall einer Lufthansa-Maschine beim Landeanflug konnte eindeutig auf die unvollständige Modellierung der Phänomene der Wirklichkeit zurückgeführt werden:

Das Bremssystem dieser Maschine war so ausgelegt, dass der Umkehrschub, der ein Flugzeug nach der Landung stark abbremst, erst eingeschaltet werden kann, wenn die Maschine tatsächlich gelandet ist. Dadurch sollte ein versehentliches Einschalten des Umkehrschubs in der Luft verhindert werden. Allerdings hatten die Ingenieure den Zustand des Gelandetseins so festgelegt, dass beide Fahrwerke mit einem vorgegebenen Anpressdruck auf der Fahrbahn aufliegen müssen.

Da an jenem Tag starker Regen und strenge Scherwinde herrschten, erreichte die Maschine trotz Bodenkontakt nicht den vorgesehenen Anpressdruck. Der Umkehrschub konnte nicht rechtzeitig aktiviert werden, und die Maschine rollte mit hoher Geschwindigkeit über die Landebahn hinaus.

⌟

Um systematisch aus einer Aufgabenbeschreibung die Kandidaten für die verschiedenen Elemente zu gewinnen, müssen wir uns fragen:

- Welche dinglichen und abstrakten Gegenstände werden bearbeitet, verändert oder tauchen in Kommunikationssituationen auf?
- Welche Eigenschaften zeichnen diese Gegenstände aus?
- Welche Beziehungen bestehen zwischen ihnen?
- Wie wird mit ihnen umgegangen?
- Welche Rollen treten auf, und für welche Handlungen sind sie verantwortlich?

Bei der Substantivanalyse werden Substantive als mögliche Kandidaten für Klassen, Attribute oder Assoziationen und Verben als Kandidaten für Methoden identifiziert. Konkrete Attributwerte oder Objekte interessieren bei der Erstellung des Klassenmodells eher weniger.

Substantivanalyse

Beispiel 3.4-2:
Betrachten wir die Fallstudie, so finden wir unter anderem uninteressante Substantive und konkrete Namen wie Zukunft, Geschäftsprozesse, Adas Blumenland und Ada König. Wir stoßen jedoch auch auf Klassenkandidaten wie Pflanze, Dekorationsartikel und Chefin.

⌟

Selbsttestaufgabe 3.4-1:

Identifizieren Sie analog zu Beispiel 3.4-2 weitere Kandidaten für Klassen, Attribute und Methoden aus der Beschreibung der Fallstudie. Überlegen Sie auch, wo vermutlich Informationen fehlen, die noch nachträglich erfragt werden müssen. ◊

3.5 UML-Klassendiagramm

Klassen, Objekte und ihre Beziehungen können mit UML-Klassendiagrammen beschrieben werden, deren Elemente wir im Folgenden vorstellen.

Definition 3.5-1: UML-Klassendiagramm

Ein UML-Klassendiagramm beschreibt grafisch die Attribute, Methoden sowie Assoziationen und Generalisierungen zwischen Klassen.

UML-Klassendiagramm

⌟

Eine Klasse wird wie in Abb. 3.5-1 in einem UML-Klassendiagramm als unausgefülltes Rechteck dargestellt, wobei der Klassenname in das Rechteck geschrieben wird.

Abb. 3.5-1: Minimale Darstellung einer Klasse

Um die Attribute einer Klasse in einem UML-Klassendiagramm darzustellen, wird dem Rechteck ein neuer Abschnitt hinzugefügt. In diesem Abschnitt wird ein Attribut pro Zeile aufgezählt. Das Attribut muss mindestens einen Namen aufweisen. Hat eine Klasse keine Attribute oder sind sie in dem aktuellen Diagramm nicht von Interesse, so kann wie in Abb. 3.5-2 der gesamte Abschnitt entfallen. Werden nicht alle Attribute im Diagramm erwähnt, so kann das Vorhandensein weiterer Attribute mit drei Punkten am Ende des Abschnitts angedeutet werden.

Attribute im UML-Klassendiagramm

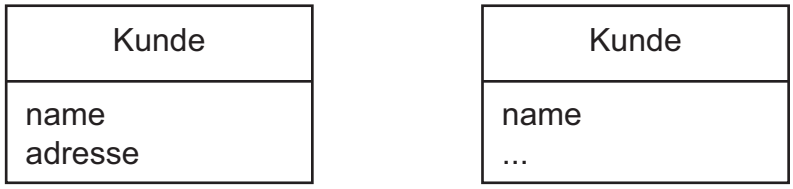

Abb. 3.5-2: Darstellung aller Attribute einer Klasse (links) und eines Teils der Attribute (rechts)

Methoden im UML-Klassendiagramm

Ebenso kann optional ein Abschnitt für Methoden eingefügt werden. Dieser wird unterhalb des Abschnitts für Attribute ergänzt. Dabei wird auch eine Methode pro Zeile aufgezählt. Es muss mindestens der Name (inklusive des Klammerpaares) genannt werden. Wie schon bei den Attributen kann das Vorhandensein weiterer Methoden mit drei Punkten am Ende des Abschnitts angedeutet werden.

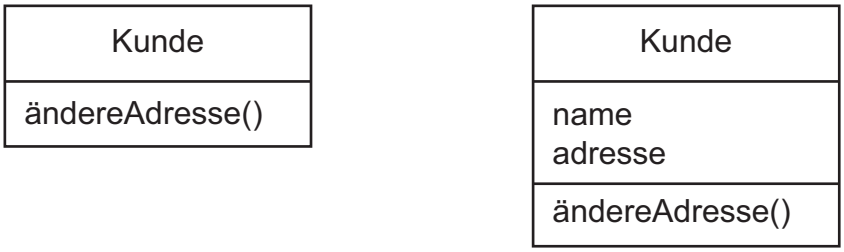

Abb. 3.5-3: Darstellung aller Methoden einer Klasse ohne Attribute (links) und einer Klasse mit Attributen (rechts)

Bemerkung 3.5-1:
Weitere Eigenschaften von Attributen und Methoden und deren Notation in UML werden wir erst später im Kurs kennen lernen.

⌟

Assoziation im UML-Klassendiagramm

Eine Assoziation zwischen zwei Klassen wird als einfache Linie zwischen den Klassen angedeutet. Der Name der Assoziation wird an die Linie geschrieben. Solange keine einfachen Pfeilspitzen an den Enden angegeben oder die Enden mit einem X gekennzeichnet werden, ist die Navigierbarkeit der Assoziation undefiniert. Die Navigierbarkeit gibt an, welches der beteiligten Objekte einen Verweis auf das jeweils andere besitzt.

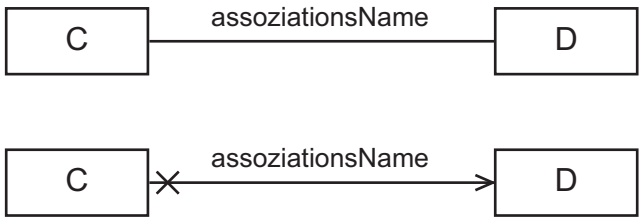

Abb. 3.5-4: Darstellung einer Assoziation mit undefinierter Navigierbarkeit (oben) und einer Assoziation, bei der nur die Objekte der Klasse C die Objekte der Klasse D kennen und nicht umgekehrt (unten)

Oft will man bei Assoziationen ausdrücken, mit wie vielen Objekten einer anderen Klasse ein Objekt in Beziehung stehen kann bzw. muss. Beispielsweise muss eine Rechnung immer für genau eine Kundin ausgestellt worden sein. Es kann aber durchaus mehrere Rechnungen für eine Kundin geben. Solche Aussagen werden in der UML durch Multiplizitäten dargestellt.

Multiplizitäten

Um auszudrücken, dass eine Rechnung für genau einen Kunden ausgestellt wird, steht an dem Ende der Assoziation bei der Klasse `Kunde` eine `1`. Die Aussage, dass es zu einem Kunden mehrere Rechnungen geben kann, wird durch ein `*` an dem anderen Ende der Assoziation dargestellt. Ein `*` steht dabei für beliebig viele.

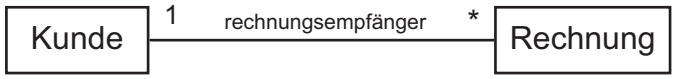

Abb. 3.5-5: Multiplizitäten zwischen Kunden und Rechnungen

Multiplizität `Y` in Abb. 3.5-6 legt fest, mit wie vielen Objekten der Klasse `B` ein Objekt der Klasse `A` eine Beziehung `beispielBeziehung` hat.

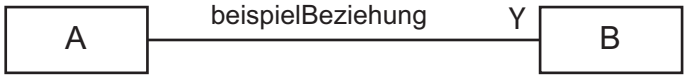

Abb. 3.5-6: Multiplizitäten zwischen Klassen

Eine gültige Multiplizität kann entweder eine ganze positive Zahl oder `*` sein. Wenn ein Bereich angegeben werden soll, so wird dieser mit `untereGrenze..obereGrenze` dargestellt. Dabei kann `obereGrenze` auch wieder `*` sein.

Eine Generalisierungsbeziehung wird mit unausgefüllten Pfeilspitzen dargestellt, wobei die Pfeilspitze zur allgemeineren Klasse zeigt.

Generalisierungsbeziehung in der UML

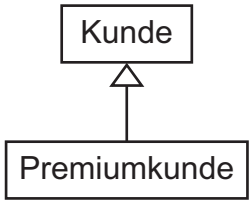

Abb. 3.5-7: Generalisierungsbeziehung zwischen den Klassen `Kunde` und `Premiumkunde`

Selbsttestaufgabe 3.5-1:

Erstellen Sie eine Klassenhierarchie in Form eines UML-Klassendiagramms für geometrische Figuren. Die Hierarchie sollte mindestens die folgenden Figuren umfassen:

Polygon, Dreieck, Viereck, Fünfeck, Quadrat, Trapez, Parallelogramm, Raute, Sehnenviereck, Rechteck, Kreis, Kegelschnitt und Ellipse.

Sie können gerne noch weitere Figuren hinzufügen. Versuchen Sie auch Attribute, Methoden und Assoziationen zu identifizieren.

◊

3.6 UML-Objektdiagramm

UML-Objektdiagramm

Um konkrete Situationen beschreiben zu können, ist es manchmal hilfreich Objekte darzustellen. In einem UML-Objektdiagramm wird ein Objekt durch ein Rechteck dargestellt. In dem Rechteck stehen, durch einen Doppelpunkt getrennt und unterstrichen, der Name und die Klasse des Objekts. Hat das Objekt keinen speziellen Namen, so kann dieser auch entfallen. Wenn gewünscht, kann auch noch ein Abschnitt für die Attributwerte ergänzt werden.

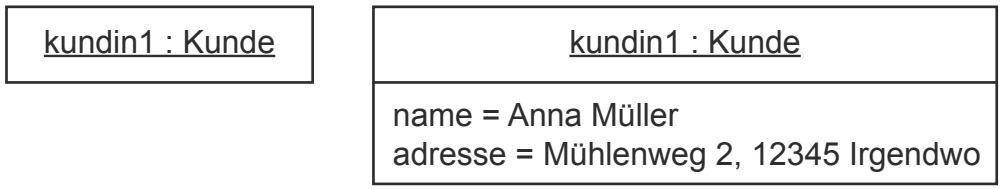

Abb. 3.6-1: Darstellung des Objekts `kundin1` in vereinfachter Form (links) und mit Attributwerten (rechts)

Verknüpfungen zwischen Objekten werden auf dieselbe Art dargestellt wie Assoziationen zwischen Klassen.

Selbsttestaufgabe 3.6-1:

Stellen Sie die folgende Situation in Form eines UML-Objektdiagramms dar.

Die Rechnung über 70 Euro mit der Nummer 1234 ist am 17.8.1998 für die Kundin Hanna Schneider ausgestellt worden. Hanna Schneider wohnt im Tulpenweg 7 in 56789 Beispielstadt.

◊

4 Einführung in die Algorithmik

Bisher haben wir uns damit beschäftigt, die Gegenstände der Anwendungsdomäne sowie ihre Struktur und Beziehungen untereinander zu beschreiben. Dabei haben wir auch modelliert, welches Verhalten die einzelnen Klassen aufweisen sollen. Jedoch haben wir das Verhalten bisher nicht beschrieben.

Die Beschreibung von bestimmten Verfahren zur Lösung gegebener Probleme oder Aufgaben ist auch in Bereichen außerhalb der Informatik nötig. Typische Beispiele dafür sind Kochrezepte, Baupläne für ein Flugzeug oder Bootsmodell, die Bedienungsanleitung zur Programmierung eines Videorekorders, Autoreparaturanweisungen oder Fahranweisungen, um von Ort A nach B zu kommen. Solche Verfahrensbeschreibungen richten sich in der Regel an Menschen und sind deswegen nicht so präzise verfasst wie es Beschreibungen sein müssen, die ein Rechner versteht.

In diesem Kapitel werden wir uns mit der Beschreibung von Verhalten in der Programmierung beschäftigen. Dazu lernen wir zunächst den Begriff des Algorithmus kennen und anschließend die Beschreibung von Verhalten mit UML-Aktivitätsdiagrammen.

4.1 Algorithmen

In der Algorithmik wird versucht, für Problemklassen möglichst gute Lösungen zu finden. Dabei ist wichtig, dass man Lösungen nicht für ein konkretes Problem mit konkreten Werten finden will, sondern für alle Probleme dieser Klasse.

Beispiel 4.1-1: Problem vs. Problemklasse

Problemklasse

Will man die Zahlenfolge 6, 32, 1, 64, 12 sortieren, so handelt es sich dabei um ein konkretes Problem, für das man auch einfach eine Lösung benennen kann. Die dazugehörige Problemklasse ist jedoch die Sortierung einer beliebigen Zahlenfolge mit einer endlichen Anzahl an Elementen. Dafür kann man keine konkrete Lösung angeben, sondern lediglich ein Verfahren beschreiben, das alle Probleme dieser Art löst. Man könnte das Problem noch allgemeiner formulieren, wenn man nicht nur Zahlenfolgen, sondern Folgen mit beliebigen Elementen mit einer eindeutigen Ordnung sortieren will.

⌟

Algorithmen sind in allen Zweigen der Informatik von Bedeutung. Algorithmen wurden nach Ibn Mûsâ Al-Chwârismî benannt, einem persisch-arabischen Autor, der im frühen 9. Jahrhundert eine Arbeit über Arithmetik und Algebra verfasste. Dieses Werk wurde ins Lateinische übersetzt und „algorismus“ genannt. Abb. 4.1-1 zeigt den Mathematiker, der in Bagdad lebte und dort im „Haus der Weisheit“ arbeitete.

Abb. 4.1-1: Al-Chwârismî

Der Begriff des Algorithmus kann auf verschiedene Arten definiert werden. Wir werden im Rahmen des Kurses die folgende Definition verwenden:

Definition 4.1-1: Algorithmus

Algorithmus

Ein Algorithmus ist eine wohldefinierte Verfahrensbeschreibung, die aus einem oder mehreren Eingabewerten einen oder mehrere Ausgabewerte mit bestimmten Eigenschaften produziert. Ein Algorithmus löst dabei immer eine Klasse von Problemen. ⌟

Ein Algorithmus im Sinne von Definition 4.1-1 hat die folgenden Eigenschaften:

Terminierung
- Terminierung: Ein Algorithmus muss bei allen möglichen Eingaben nach endlich vielen Schritten beendet sein und ein Ergebnis liefern.

Finitheit
- Finitheit: Die Beschreibung des Algorithmus muss endlich sein, ebenso wie der zur Ausführung benötigte Speicher.

Effektivität
- Effektivität: Alle Schritte eines Algorithmus müssen eindeutig und in einer endlichen Zeit ausführbar sein.

Die folgenden beiden Eigenschaften werden häufig und so auch im Rahmen dieses Kurses gefordert:

Determiniertheit
- Determiniertheit: Ein Algorithmus liefert bei den gleichen Eingaben immer das gleiche Ergebnis.

Determinismus
- Determinismus: Bei einer Ausführung ist zu jedem Zeitpunkt eindeutig festgelegt, welcher Schritt als nächstes ausgeführt wird.

Bemerkung 4.1-1: Nichtdeterministische Algorithmen

Zusätzlich zu den deterministischen Algorithmen gibt es auch nichtdeterministische. Diese werden wir jedoch im Kurs nicht weiter betrachten. ⌟

Bemerkung 4.1-2: effektiv vs. effizient

Ein Algorithmus muss effektiv sein, d. h. in einer endlichen Zeit das Problem lösen. Häufig sind jedoch effiziente Algorithmen gesucht, die das Problem zum Beispiel in relativ kurzer Zeit oder mit wenig Speicher lösen.

⌟

Selbsttestaufgabe 4.1-1:

Versuchen Sie sich den Unterschied zwischen Determiniertheit und Determinismus zu verdeutlichen.

◊

Bemerkung 4.1-3: Verfahrensbeschreibungen aus anderen Bereichen

Verfahrensbeschreibungen aus anderen Bereichen als der Informatik erfüllen oft nicht alle Eigenschaften, die wir von einem Algorithmus verlangen.

⌟

Beispiel 4.1-2: Algorithmus von Euklid

Algorithmus von Euklid

Noch viel früher in der Zeitrechnung als Al-Chwârismî entwarf im 3. Jahrhundert v. Ch. der Mathematiker Euklid aus Alexandria in Ägypten in seinem Buch „Die Elemente“ das noch heute gebräuchliche Verfahren zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

Der nachfolgende Algorithmus bestimmt den größten gemeinsamen Teiler der positiven ganzen Zahlen a und b.

1. *Wenn b > a ist, vertausche a und b.*
2. *Sei c der Rest der Division von a durch b.*
3. *Wenn c = 0 ist, dann ist der Algorithmus zu Ende und b das Ergebnis.*
4. *a wird der Wert von b zugewiesen und b der Wert von c.*
5. *Fahre mit Schritt 2 fort.*

⌟

Selbsttestaufgabe 4.1-2:

Führen Sie den Algorithmus mit den folgenden Eingaben aus. Wie oft wird jeder Schritt ausgeführt und was ist das jeweilige Ergebnis?

a) a = 21, b = 6

b) a = 78, b = 273

c) a = 13, b = 31

d) a = 64, b = 34

◊

Selbsttestaufgabe 4.1-3:

Wir haben gefordert, dass ein Algorithmus terminiert. Zeigen Sie, dass der Algorithmus von Euklid terminiert.

◊

Bisher haben wir uns mit den Eigenschaften beschäftigt, die ein Algorithmus zwingend haben muss. Im Folgenden werden wir einige weitere wichtige Eigenschaften von Algorithmen kennen lernen.

Korrektheit

Ein Algorithmus ist korrekt, wenn er sich bei der Ausführung so verhält, wie wir dies bei der Formulierung beabsichtigten. In kritischen Anwendungsbereichen wie der Flugsicherung, der Medizin oder der Prozesssteuerung können durch inkorrekte Algorithmen verursachte Fehlfunktionen katastrophale Schäden anrichten. Ein fehlerhafter Algorithmus in einem Chipentwurf kann erhebliche Kosten für den verantwortlichen Hersteller nach sich ziehen, wenn der Fehler erst nach Auslieferung der Prozessoren entdeckt wird.

Unsere Bestimmung des Begriffs Korrektheit ist zwar intuitiv verständlich aber auch sehr unpräzise. Korrektheit ist ein relativer Begriff, weil Korrektheit immer nur in Bezug auf eine explizite Vorgabe objektiv überprüft werden kann. In der Praxis versucht man die Korrektheit eines Algorithmus häufig durch Testen nachzuweisen. Beim Testen von Programmen ist das Vergleichsobjekt die jeweilige Testspezifikation. In einer späteren Kurseinheit werden wir verschiedene Testverfahren kennen lernen, um Klassen zu testen.

Im Allgemeinen kann man mit Tests jedoch nur die Anwesenheit von Fehlern, nicht aber deren Abwesenheit nachweisen. Die Anzahl der Tests ist immer begrenzt, so dass Testen nur eine Auswahl aller möglichen Abläufe eines gängigen Programms erfassen kann. Nur wenn der Zustandsraum eines sequenziellen Programms erschöpfend abgesucht werden kann, liefert Testen den erwünschten Nachweis. In kritischen Anwendungen wendet man deshalb mathematisch strenge Beweistechniken an. Hier wird der Korrektheitsbeweis immer in Bezug auf eine präzise Spezifikation dessen, was der Algorithmus tun soll, geführt.

Die verschiedenen Ingenieursdisziplinen, insbesondere die Informatik, entwickelten eine Vielzahl von Spezifikationstechniken, um Eigenschaften von Algorithmen und Programmen mit mathematischer Strenge zu formulieren, sowie Beweiswerkzeuge, um Softwareentwickler bei der Führung eines Korrektheitsbeweises zu unterstützen. Wir werden solche Techniken und Werkzeuge in diesem Kurs nicht weiter betrachten, sondern nur mathematische oder informationstechnische Schreibweisen anwenden, um Algorithmen eindeutig zu spezifizieren.

Beispiel 4.1-3: Spezifikation des Sortierproblems
Die Aufgabe eines Sortieralgorithmus besteht darin, eine gegebene Folge von Werten

$$\langle d_1, d_2, d_3, \ldots, d_n \rangle$$

gemäß einer vorgegebenen Ordnung „$\leq$“ in eine permutierte Folge:

$$\langle d'_1, d'_2, d'_3, \ldots, d'_n \rangle$$

zu bringen, sodass gilt:

$$d'_1 \leq d'_2 \leq d'_3 \leq \cdots \leq d'_n$$

Die Ausgangsreihe ist eine Permutation der Eingangsreihe. ⌟

Bemerkung 4.1-4: Permutation
Eine Permutation ist eine Umordnung einer Folge von Elementen in einer Weise, dass kein Element hinzugefügt wird oder verloren geht und auch kein Element verändert wird. Die Anzahl der Permutationen einer Folge mit n Elementen ergibt sich zu n! (Fakultät). ⌟

Komplexität

Die Komplexität eines Algorithmus befasst sich mit dem Mindestaufwand, den ein Algorithmus zur Lösung einer Aufgabe benötigt. Die Komplexitätstheorie untersucht sowohl den Rechenaufwand als auch den Speicherplatzbedarf. Beim Rechenaufwand ist nicht die effektive Ausführungszeit auf einem Rechner von Belang, sondern die Zahl der Bearbeitungsschritte in Abhängigkeit von der jeweiligen Eingangsgröße. Auf die Details der Komplexitätsbewertung von Algorithmen können wir im Rahmen dieses Kurse jedoch nicht weiter eingehen.

Robustheit

Es gibt noch eine Reihe weiterer Eigenschaften von Algorithmen und Programmen, wie Robustheit, Anpassbarkeit, Wiederverwendbarkeit, die je nach Anwendungsfall mehr oder weniger von Bedeutung sind. Unter Robustheit verstehen wir die Eigenschaft eines Algorithmus, auch in ungewöhnlichen Situationen definiert zu arbeiten, d. h. fehlerhafte und widersprüchliche Eingabedaten abzuweisen und auch umfangreiche Eingabedaten abzuhandeln. Der Aspekt Robustheit ergänzt den Aspekt Korrektheit, denn die Korrektheit betrifft das Verhalten eines Algorithmus im Rahmen der durch die Spezifikation erfassten Fälle und Bedingungen. Robustheit charakterisiert dagegen das Verhalten bei nicht spezifizierten Bedingungen.

Anpassbarkeit

Anpassbarkeit ist ein Maß für den Aufwand, mit dem ein Algorithmus an eine geänderte Spezifikation angepasst werden kann. Diese Eigenschaft ist vor allem für große Programmsysteme von erheblicher Bedeutung, weil die Auswirkungen von Änderungen an einer Stelle – anders als in kleinen Programmen – kaum zu überschauen sind.

Wiederverwendbarkeit

Die Wiederverwendbarkeit drückt aus, wie einfach ein Algorithmus in vielen ver-

schiedenen Anwendungen ohne Änderung benutzt werden kann.

Wir wissen nun, worum es sich bei Algorithmen handelt und welche Eigenschaften sie besitzen können bzw. müssen. Im nächsten Abschnitt werden wir uns genauer mit der Beschreibung von Algorithmen bzw. Verfahrensbeschreibungen beschäftigen.

Selbsttestaufgabe 4.1-4:

Versuchen Sie einen Algorithmus zu entwickeln, der überprüft, ob es sich bei einer gegebenen positiven ganzen Zahl a um eine Primzahl handelt. ◊

4.2 Verhaltensbeschreibung und Kontrollstrukturen

Kontrollstrukturen

UML-Aktivitätsdiagramm

Wir werden im Folgenden die grundlegenden Kontrollstrukturen für Verfahrensbeschreibungen kennen lernen sowie deren Darstellung in Form von UML-Aktivitätsdiagrammen. Die behandelten Kontrollstrukturen sind in den meisten imperativen und objektorientierten Programmiersprachen ähnlich. Mit der Umsetzung der Kontrollstrukturen in Java werden wir uns in der nächsten Kurseinheit beschäftigen. Bevor wir uns mit den einzelnen Kontrollstrukturen befassen, lernen wir die nötigen Grundlagen des UML-Aktivitätsdiagramms kennen.

Definition 4.2-1: Aktivität

Aktivität

In der UML ist eine Aktivität ein benanntes Verhalten. Ein Aktivitätsdiagramm spezifiziert eine solche Aktivität. ⌟

Bemerkung 4.2-1:
Eine solche Aktivität kann zum Beispiel das Verhalten einer einzelnen Methode widerspiegeln, aber auch grobgranularer ganze Geschäftsprozesse beschreiben. Im Rahmen des Kurses werden wir Aktivitäten für die Beschreibung einzelner Methoden verwenden. ⌟

Eine Aktivität wird durch ein Rechteck mit abgerundeten Ecken dargestellt. In der linken oberen Ecke steht der Name (siehe Abb. 4.2-1).

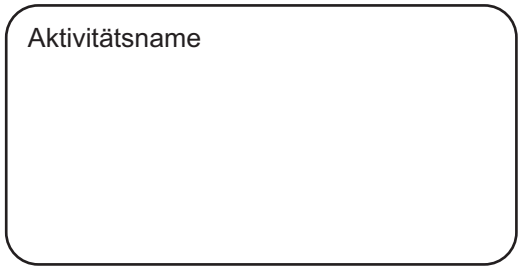

Abb. 4.2-1: Darstellung einer Aktivität in einem UML-Aktivitätsdiagramm

Innerhalb der Aktivität befinden sich die verschiedenen atomaren Aktionen und Kontrollknoten, die die Ausführungsreihenfolge der Aktionen festlegen. Diese sind mit gerichteten Kanten miteinander verbunden.

Aktion
Kontrollknoten

Eine Aktion wird ebenfalls durch ein Rechteck mit abgerundeten Ecken dargestellt. Sie kann beispielsweise für den Aufruf einer Methode stehen, in diesem Fall wird entweder der Methodenname in die Aktion geschrieben oder das Verhalten anderweitig beschrieben. Die Beschreibung kann umgangssprachlich sein, wie zum Beispiel „Adresse erfassen“, oder auch formaler Natur in Form von Berechnungen „c = a + b“. Später werden wir zur Beschreibung solcher Aktionen auch Java-Quelltext verwenden.

Im einfachsten Fall besteht eine Aktivität aus einer Sequenz mehrerer Aktionen. Dafür benötigen wir noch den Initial- und den Aktivitätsendknoten. Der Erste von beiden kennzeichnet den Beginn des Ablaufs und der Zweite markiert das Ende der Ausführung (siehe Abb. 4.2-2).

Sequenz
Initialknoten
Aktivitätsendknoten

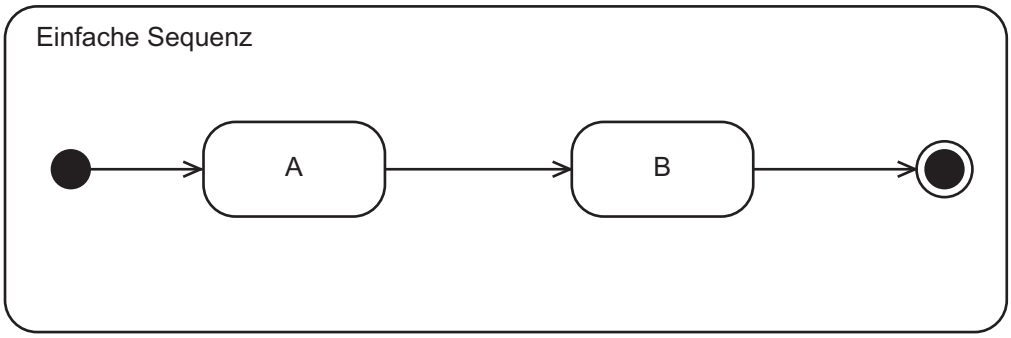

Abb. 4.2-2: Ein Aktivitätsdiagramm mit einer einfachen Sequenz von Aktionen mit dem Initialknoten (links) und dem Aktivitätsendknoten (rechts)

Mit der Sequenz, also der Hintereinanderausführung einer endlichen Anzahl von Aktionen, haben wir ein zentrales Element der Verfahrensbeschreibung kennen gelernt. Die Ausführung der nachfolgenden Aktion wird erst gestartet, wenn der Vorgänger vollständig ausgeführt wurde.

Um auch komplexere Algorithmen oder Verfahren beschreiben zu können, brauchen wir weitere Kontrollstrukturen, wie zum Beispiel eine Verzweigung. Bei Euklids Algorithmus trat eine solche Verzweigung zum Beispiel in Schritt 1 auf.

Verzweigung

Ein Entscheidungsknoten wird durch eine Raute dargestellt und hat einen Ein- und beliebig viele Ausgänge. Wird bei der Ausführung ein solcher Knoten erreicht so muss das Entscheidungsverhalten ausgeführt werden. Das auszuwertende Verhalten wird dabei an den Entscheidungsknoten, wie in Abb. 4.2-3 dargestellt, annotiert. Nach der Auswertung wird überprüft, an welchem Ausgang die in eckigen Klammern stehende Überwachungsbedingung erfüllt ist. Dem Pfad an diesem Ausgang wird dann gefolgt.

Entscheidungsknoten
Überwachungsbedingung

Will man einen Ausgang haben, der alle sonstigen Fälle abdeckt, so schreibt man als Überwachungsbedingung `[else]`.

Vereinigungsknoten

Damit nach einer Verzweigung die Pfade auch wieder zusammengeführt werden können, gibt es Vereinigungsknoten. Diese werden ebenso durch eine Raute dargestellt, besitzen aber mehrere Eingänge und nur einen Ausgang. Ein Entscheidungs- und ein Vereinigungsknoten können auch verschmolzen werden, so dass der Knoten mehrere Ein- und Ausgänge besitzen kann.

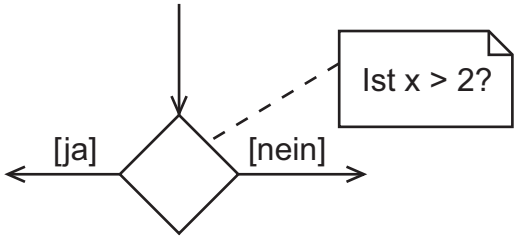

Abb. 4.2-3: Entscheidung mit annotierten Entscheidungsverhalten

Selbsttestaufgabe 4.2-1:

Versuchen Sie die folgende Verhaltensbeschreibung in ein Aktivitätsdiagramm zu überführen:

Beträgt der Preis mehr als 100 Euro, so wird dem Kunden ein Rabatt von 3 % gewährt. Bei allen Preisen muss zum Schluss noch die Mehrwertsteuer von 19 % hinzugerechnet werden.

◊

Wiederholung

Bei vielen Verhaltens- oder Algorithmusbeschreibungen müssen bestimmte Schritte wiederholt werden. So muss bei Euklids Algorithmus nach Schritt 5 beispielsweise wieder Schritt 2 ausgeführt werden. In einem solchen Fall weist die Kante auf einen Vorgängerknoten. Solche Kanten werden im Normalfall als Rückwärtskanten bezeichnet. Durch eine Rückwärtskante entsteht eine Schleife. In Abb. 4.2-4 gehören alle grau gefärbten Aktionen und Kontrollknoten zu der einzigen Schleife.

Rückwärtskante
Schleife

In der Regel soll der Kontrollfluss auch wieder aus einer solchen Schleife herausführen können, so dass ein Entscheidungsknoten innerhalb bzw. am Rand der Schleife notwendig ist. In diesem Fall spricht man von bedingten Wiederholungen oder Schleifen. In einer Aktivität können mehrere disjunkte Schleifen auftreten, aber auch ineinander Enthaltene.

Wir haben die grundlegenden Kontrollstrukturen, Sequenz, Verzweigung und Wiederholung für Verfahrensbeschreibungen sowie deren Darstellung in Form von UML-Aktivitätsdiagrammen kennen gelernt. Diese Strukturen sind der imperativen und objektorientierten Programmierung gemein und unabhängig von der verwendeten Programmiersprache.

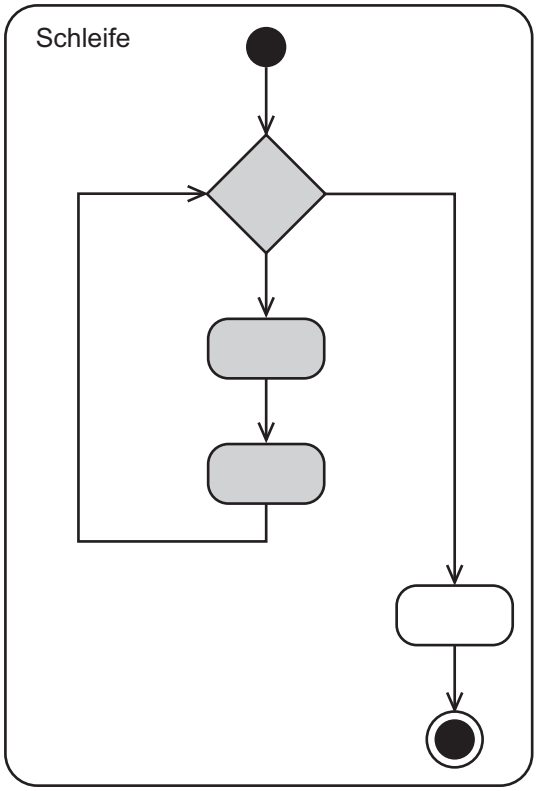

Abb. 4.2-4: Ein Aktivitätsdiagramm mit einer enthaltenen Schleife (grau markiert)

Selbsttestaufgabe 4.2-2:

Versuchen Sie nun, den Algorithmus von Euklid aus Beispiel 4.1-2 in Form eines Aktivitätsdiagramms darzustellen. Besitzt der Algorithmus eine oder mehrere Schleifen? Wenn ja: welche Aktionen und Kontrollknoten gehören dazu? ◊

Selbsttestaufgabe 4.2-3:

Erstellen Sie das Aktivitätsdiagramm zum Primzahl-Algorithmus aus Selbsttestaufgabe 4.1-4. ◊

5 Programmiersprachen

Computer können natürliche Sprachen, die wir Menschen benutzen, heute und auf absehbare Zeit nicht ausreichend verstehen und Anweisungen in natürlicher Sprache nicht ausführen. Deshalb wurden schon in der Frühzeit der Computer eigene Programmiersprachen entwickelt, die viel einfacher aufgebaut sind als natürliche Sprachen wie Deutsch, Englisch oder Chinesisch.

Programmiersprachen sind Kunstsprachen zur Darstellung von Computerprogrammen. Sie sind in ihrer Syntax und Semantik wesentlich einfacher gestaltet als natürliche Sprachen. Die Syntax einer Sprache bestimmt den Aufbau korrekter Ausdrücke einer Sprache und die Semantik die Bedeutung der Ausdrücke.

Programmiersprachen liefern uns den Vorrat an elementaren und in ihrer Bedeutung genau festgelegten Ausdrücken, den wir benötigen, um eine Verfahrensbeschreibung oder ein schematisches Lösungsverfahren für eine bestimmte Aufgabenstellung zu formulieren. Solche aus endlich vielen, ausführbaren elementaren Verarbeitungsschritten bestehenden Beschreibungen sind die Programme.

Statt der im Computer intern verarbeiteten Binärwerte werden in Programmiersprachen häufig verwendete Zahlen und Zeichen symbolisch angegeben. Ihrem Anwendungszweck entsprechend bieten sie Befehle, Steueranweisungen und andere Sprachkonstrukte an, die das Formulieren von Programmen für numerische Berechnungen, für die betriebliche Datenverarbeitung, für die Systemprogrammierung u. a. m. erleichtern. Zugleich abstrahieren sie von den elementaren Operationen des Rechners wie Zugriffe auf Speicherzellen oder logische und arithmetische Operationen.

5.1 EXKURS: Entwicklungsgeschichte von Programmiersprachen

Die ersten höheren Programmiersprachen wie Fortran, Algol oder Pascal unterstützen einen sog. imperativen oder prozeduralen Programmierstil, der sich an den Abläufen (engl. *processes*) und Vorgehensweisen (engl. *procedure*) des Anwendungsgebiets (engl. *application domain*) orientiert. Programme werden bei der imperativen Programmierung entlang der Systemabläufe strukturiert.

Bei der prozeduralen Programmierung orientiert man sich beim Entwurf eines Programms an den Prozeduren, die beschreiben, wie bestimmte Aufgaben ausgeführt werden. Diese Prozeduren tauschen untereinander Daten aus und verändern ihre gemeinsamen globalen Datenstrukturen. Der Aufbau der Datenstrukturen des Programms ist allen Prozeduren zugänglich. Die Strukturierung von Daten, die von den Prozeduren manipuliert werden, ist jedoch dem Entwurf von Prozeduren nachgeordnet.

Ein Nachteil dieses Programmieransatzes ergibt sich aus dieser ungleichen Behandlung von Prozeduren und Datenstrukturen. Sobald nämlich auf Grund veränderter Anforderungen vorhandene Datenstrukturen geändert werden müssen, ist im Extremfall eine Überarbeitung aller Prozeduren, die auf diesen Datenstrukturen aufbauen, fällig. Dies schränkt die Erweiterbarkeit prozeduraler Programme wegen der Notwendigkeit, Änderungen über das gesamte Programm hinweg nachzuvollziehen, erheblich ein.

Ein Computerprogramm, das z. B. in einer Bibliothek eingesetzt werden soll, besteht aus Prozeduren, die wesentliche Anteile der Aufgaben Ausleihe, Rückgabe, Reservierung, Katalogisieren und Beschaffen von Büchern in den Rechner verlagern.

Die Entwicklung der Sprache Fortran war ein entscheidender Abstraktionsschritt, der von vielen Details der Maschinensprachen wie Registerbenutzung und Speicherverwaltung absah und die Aufmerksamkeit des Programmierers auf algebraische Operationen hin lenkte.

Algol und Pascal gingen in der Abstraktion einen Schritt weiter, indem sie auch heute noch wesentliche Kontrollstrukturen einführten. Diese sind auch Bestandteil der Sprache Java.

Etwas neuere Sprachen wie Modula oder Ada stellen die Datenabstraktion und die Zerlegung eines Programms in modulare Einheiten in den Vordergrund der Anwendungsprogrammierung.

Hierbei werden die Datenstrukturen einer Anwendung in abgrenzbare Einheiten, sog. Module (im Fall von Modula) oder Pakete (bei Ada), so eingekapselt, dass Veränderungen an den Datenstrukturen ausschließlich über definierte Zugriffsoperationen erfolgen können. Die Module operieren nicht mehr auf einem gemeinsamen Datenbestand, sondern besitzen ihre eigenen Datenstrukturen, deren Aufbau nach außen verborgen bleibt. Erweiterungen werden so vereinfacht, und die Freiheitsgrade für Implementierungs- und Optimierungsentscheidungen werden größer. Dieser Konstruktionsansatz wurde von David Parnas im Jahr 1972 unter der Bezeichnung „Geheimnisprinzip“ eingeführt und liegt auch dem Prinzip der Datenabstraktion zu Grunde.

Geheimnisprinzip

Bei der datenstrukturorientierten Programmierung konstruierte man bei der Bibliotheksautomatisierung beispielsweise Module, die die verschiedenen Kataloge und Ablagen – Gesamtkatalog, Ausleihkatalog, verfügbarer Bestand, laufende Bestellungen, Rechnungen – im Rechner abbilden und entsprechende Verarbeitungsfunktionen zur Veränderung oder Abfrage dieser Datenstrukturen bereitstellen.

Simula, eine für Simulationsanwendungen Ende der 60er Jahre entwickelte Programmiersprache, ermöglichte zum ersten Mal einen objektorientierten Programmierstil.

Wir verwenden im Kurs die objektorientierte Programmiersprache Java zur Implementierung programmtechnischer Lösungen, die auf Rechnern ausführbar sind. Wir werden im nächsten Kapitel Java dann genauer betrachten. Weitere objektorientierte Programmiersprachen sind C++, Eiffel und Smalltalk.

Selbsttestaufgabe 5.1-1:

Was bedeutet das Geheimnisprinzip und warum ist es so wichtig? ◊

5.2 Eingabe, Verarbeitung, Ausgabe

Der sequenziellen Programmierung liegt die Vorstellung zu Grunde, die Arbeitsweise eines Computers entspräche einem sequenziellen Ablauf, der von einem Ausgangszustand zu einem Ergebnis führt und dann endet. In dieser Vorstellungswelt besteht die Aufgabe des Programmierers darin, eine bestimmte Folge von Anweisungen zu finden, die bestimmte Eingabedaten in ein gewünschtes Ergebnis umformen. Dieser Programmieransatz wird gelegentlich auch das EVA-Prinzip genannt (Abb. 5.2-1). EVA steht für „Eingabe, Verarbeitung, Ausgabe“.

EVA-Prinzip

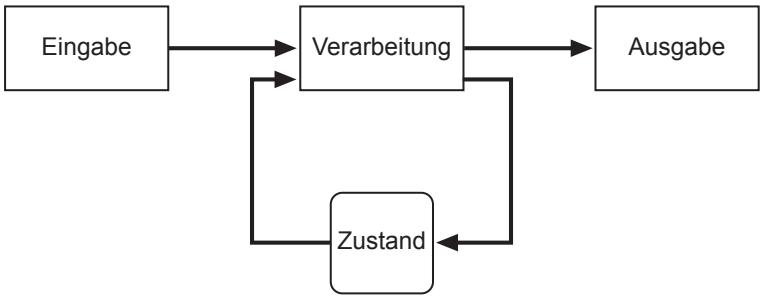

Abb. 5.2-1: EVA-Prinzip

Die Mehrzahl heutiger Rechner entspricht in ihrem Aufbau immer noch der von Neumann-Rechnerarchitektur (Abb. 5.2-2).

von Neumann-Architektur

Gemäß dieser Architektur besteht ein Rechner aus einem Speicher, der Daten und Instruktionen verwaltet, einer Zentralrecheneinheit oder CPU (engl. *central processing unit*) und einer Ein- und Ausgabeeinheit. Instruktionen werden eine nach der anderen aus dem Speicher in die CPU geladen und ausgeführt. Bei der Ausführung von Instruktionen werden auch Daten aus dem Speicher in die CPU eingelesen und durch logische oder arithmetische Operationen verändert. Die Ergebnisse dieser Verarbeitung werden in den Speicher zurück geschrieben.

Die Ausführung solcher Instruktionen bewirkt also eine Veränderung des Zustands der Maschine, der im Wesentlichen durch den Speicherinhalt, Registerwerte und den Wert des Instruktionszählers dargestellt wird.

Die im Rechner verarbeiteten Instruktionen orientieren sich begrifflich an den Möglichkeiten der Maschine und werden daher als Maschinensprache bezeichnet. Jede

Maschinensprache

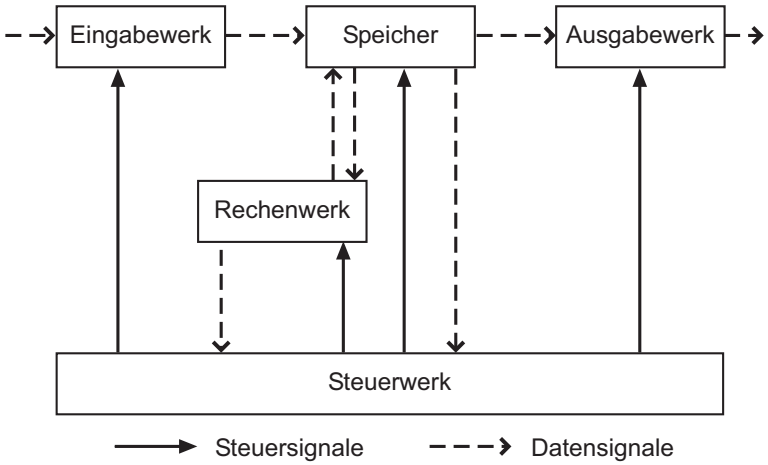

Abb. 5.2-2: von Neumann-Architektur

Maschinensprache ist jedoch für menschliche Leser schwer zu interpretieren. Deshalb wurden relativ rasch so genannte höhere Programmiersprachen entwickelt, die es auch einfacher machen, komplexe Anwendungen zu programmieren. Um Programme, die in einer höheren Programmiersprache geschrieben wurden, ausführen zu können, müssen die Programme in Maschinensprache übersetzt werden.

5.3 Interaktive Programme

Die Auffassung einer schrittweisen Abarbeitung von Anweisungen ist nicht falsch, liefert uns aber keinen brauchbaren Ansatz, um die Verhaltensweise vieler interaktiver Anwendungen wie Platzreservierungssysteme, Videospiele, Tabellenkalkulatoren, Webserver, die Airbag-Steuerung in Pkws oder Bankautomaten zu erklären.

Interaktive Anwendungsprogramme, denen wir heute vielfach begegnen, verhalten sich vielmehr wie eigenständige Agenten, die kontinuierlich auf Anforderungen oder Ereignisse in der Umgebung reagieren. Diese Anwendungen sind in ihre Umgebung eingebettet und interagieren mit dieser Umgebung, also mit Benutzer, technischen Geräten wie Sensoren und Aktoren, Kommunikationseinrichtungen oder anderen Programmen.

Will man solche Anwendungen programmieren, genügt es nicht, sich zu fragen:

Welcher Schritt ist als Nächstes zu tun?

Die Aufgabe besteht eher darin, eine gedachte Welt miteinander kooperierender Agenten oder Einheiten zu entwerfen; sich also fragen:

- Welche Einheiten werden gebraucht?
- Welche Rolle spielen sie im Verbund mit anderen?
- Welchen Dienst bieten sie an, und welche Dienstleistung benötigen sie zur Erfüllung ihrer Aufgabe?

- Wie muss ihre innere Arbeitsweise gestaltet werden?
- Wie arbeiten sie zusammen?

Diese Denkweise wird von objektorientierten Programmiersprachen – so auch von Java – unterstützt.

6 Einführung in die Java-Programmierung

Mit der ersten Sprache erlernt man nicht nur ein Vokabular und eine Grammatik, sondern man erschliesst sich eine Gedankenwelt. (Niklaus Wirth im Buch „Systematisches Programmieren[7]“)

Bevor wir in der nächsten Kurseinheit in die Programmierung mit Java einsteigen werden, gibt dieses Kapitel einen Überblick über die Entstehung von Java sowie die Erstellung und die Ausführung von Java-Programmen.

Dass wir für den vorliegenden Kurs über objektorientierte Programmierung die Sprache Java gewählt haben, hat mehrere Gründe: Java ist eine der am weitesten verbreiteten objektorientierten Programmiersprachen. Auf Grund seiner Ähnlichkeit zu anderen Sprachen, wie zum Beispiel C und C++, erleichtert es später die Erlernung weiterer Sprachen. Zudem gibt es für die meisten gängigen Rechnerplattformen entsprechende Übersetzer und Laufzeitumgebungen.

6.1 Entwicklung und Eigenschaften der Programmiersprache Java

Java 1.0 wurde 1996 von einer Gruppe um James Gosling veröffentlicht. Seitdem wurde die Sprache stetig weiterentwickelt, so dass im Jahr 2004 Java 5 und im Jahr 2006 Java 6 veröffentlicht wurden.

Im Rahmen dieses Kurses werden wir Version 5 verwenden. Da die Änderungen zwischen Version 5 und 6 für diesen Kurs nicht relevant sind, können Sie auch Version 6 verwenden.

Bei der Entwicklung von Java wurde entschieden, dass Java-Programme ohne Anpassungen auf möglichst vielen Plattformen laufen sollen. Aus diesem Grund wurde eine virtuelle Maschine (VM) entwickelt, die von der konkreten Architektur und dem Betriebssystem des Rechners abstrahiert. Java-Programme werden immer in der VM ausgeführt und sind somit unabhängig von der konkreten Architektur. Für jede gewünschte Plattform muss nur einmal eine VM entwickelt werden, sodann ist die Ausführung von Java-Programmen möglich. Zusätzlich zu gängigen Rechnerplattformen gibt es zum Beispiel auch virtuelle Maschinen für mobile Geräte. Details der VM finden sich in *The Java™ Virtual Machine Specification*. Die Details dieser Spezifikation sind für diesen Kurs jedoch nicht relevant.

Virtuelle Maschine

Zusätzlich zu dieser virtuellen Maschine existiert eine umfangreiche Bibliothek, die viele nützliche Klassen zur Verfügung stellt. Diese Bibliothek wird als API[8]

API

7 http://www.cs.inf.ethz.ch/~wirth/books/SystemProgD/

8 http://java.sun.com/j2se/1.5.0/docs/api/

(Application Programming Interface) bezeichnet. Ein paar der in dieser Bibliothek enthaltenen Klassen werden wir im Laufe des Kurses noch kennen lernen.

The Java™ Language Specification (kurz: JLS) definiert die Syntax und Semantik aller Sprachkonstrukte. Diese Sprachspezifikation muss man für diesen Kurs nicht lesen, aber bei Detailfragen kann es durchaus interessant und hilfreich sein, dort nachzulesen, zumal an einigen Stellen auch Beispiele geboten werden. Wir werden deshalb bei der Einführung von Sprachkonstrukten auf die entsprechenden Abschnitte der Spezifikation verweisen.

Weiterführende Literatur

[JLS05] James Gosling, Bill Joy, Guy Steele, Gilad Bracha
The Java™ Language Specification, Third Edition[9]

[JVM99] Tim Lindholm, Frank Yellin:
The Java™ Virtual Machine Specification, Second Edition[10]

6.2 Erstellen, Übersetzen und Ausführen von Java-Programmen

Zur Programmierung und Ausführung von Java-Programmen gibt es die verschiedenen Möglichkeiten und Entwicklungsumgebungen. Im Folgenden werden wir uns mit den Grundlagen und den dahinter stehenden Abläufen vertraut machen. Welche Umgebung Sie für die Programmierung nutzen wollen, können Sie selbst entscheiden. Auf der Webseite zu dieser Kurseinheit finden Sie entsprechende Erläuterungen und Hinweise zu ein paar ausgewählten Varianten.

`java`-Datei

Der Quelltext von Java-Programmen wird in Dateien mit der Endung `java` gespeichert. Die Datei muss in der Regel den gleichen Namen wie die enthaltene Klasse tragen. Der Quelltext eines ganzen Programms besteht aus mehreren `java`-Dateien.

Bytecode

`class`-Datei

Bevor ein Java-Programm ausgeführt werden kann, muss es zunächst in eine Sprache übersetzt werden, die die virtuelle Maschine versteht. Diese Sprache wird Bytecode genannt. Übersetzt man Programme mit dem Java-Übersetzer (javac), so entsteht für jede Klasse eine Datei mit der Endung `class`, die den entsprechenden Bytecode enthält. Der Name der `class`-Datei ist ebenfalls der Klassenname. Sollten im Quelltext syntaktische Fehler enthalten sein, so wird der Übersetzer dies melden und die Übersetzung danach beenden. Es entstehen dann keine `class`-Da-

9 http://java.sun.com/docs/books/jls/

10 http://java.sun.com/docs/books/jvms/

teien für die fehlerhaften Klassen. Die Fehler müssen zunächst behoben und das Programm erneut übersetzt werden (siehe Abb. 6.2-1).

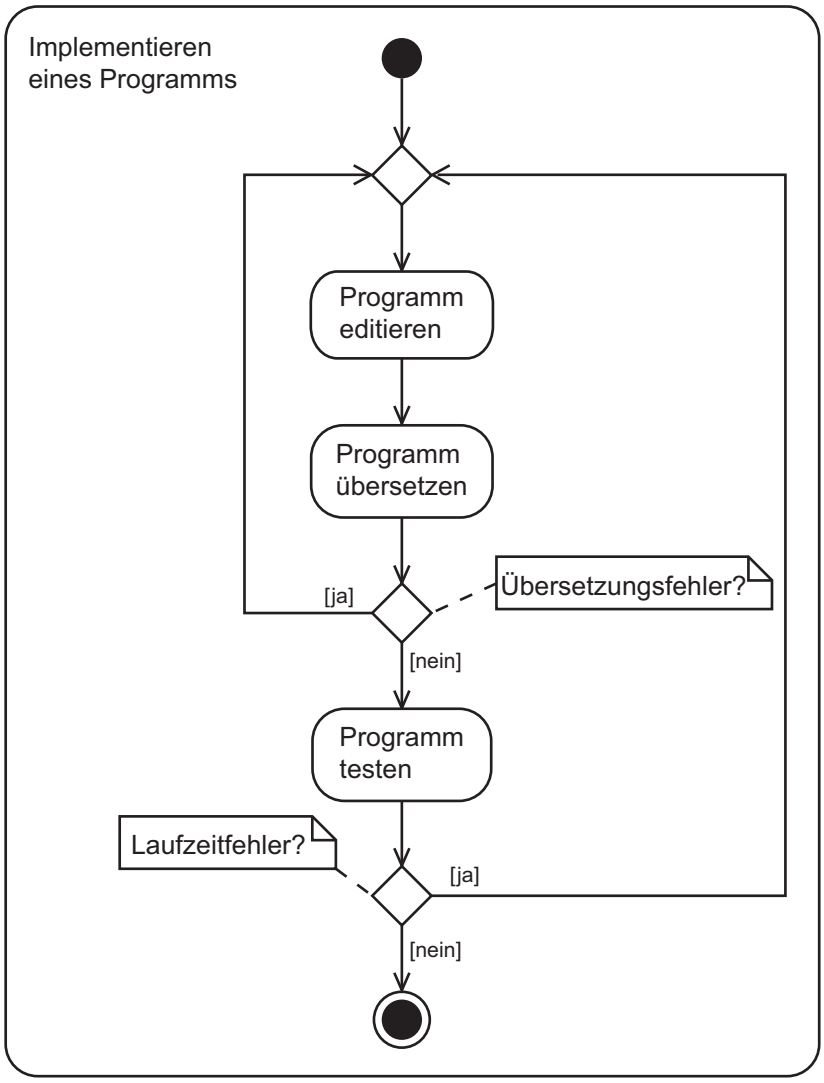

Abb. 6.2-1: Aktivitäten bei der Implementierung (nach [Horstmann08])

Beispiel 6.2-1: Übersetzung zweier Java-Klassen

Befinden sich zwei korrekte Java-Klassen `Kunde` *und* `Rechnung` *in den beiden Dateien* `Kunde.java` *und* `Rechnung.java`*, so werden diese mit Hilfe des Java-Übersetzers in die beiden* `class`*-Dateien* `Kunde.class` *und* `Rechnung.class` *übersetzt.*

⌟

Beispiel 6.2-2: Eine Fehlermeldung und ihre Bedeutung

```
MeineKlasse.java:4: ';' expected
                int x = 10
                         ^
1 error
```

In der Datei `MeineKlasse.java` *fehlt ein Semikolon in Zeile 4 hinter der Zahl* `10`.

Übersetzungsfehler

⌟

Bemerkung 6.2-1: Warnungen

Übersetzer-Warnungen

Zusätzlich zu Fehlern warnt der Übersetzer manchmal auch bei bestimmten Konstruktionen. Diese verhindern nicht die erfolgreiche Übersetzung der Klassen, sind jedoch oft ein Hinweis für eine unsaubere Stelle im Quelltext.

⌟

`main()`-Methode

Für die Programmausführung muss die virtuelle Maschine gestartet und dieser mitgeteilt werden, in welcher Klasse sich die Anfangsmethode, die `main()`-Methode, befindet. Wenn nötig können der virtuellen Maschine oder dem Programm Argumente mitgegeben werden. Die Ausführung des Java-Programms beginnt bei der `main()`-Methode der angegebenen Klasse. In der `main()`-Methode werden in der Regel die für den Programmstart nötigen Objekte erzeugt. Sodann können Methoden dieser Objekte aufgerufen werden.

Die bei der Entwicklung eines Programms entstehenden Artefakte und deren Verwendung sind in Abb. 6.2-2 veranschaulicht.

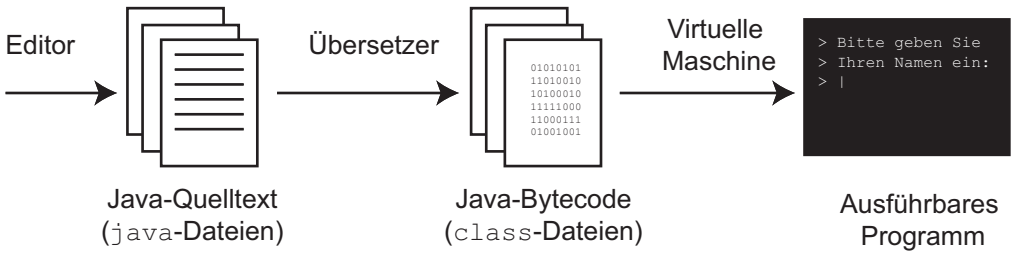

Abb. 6.2-2: Von einer `java`-Datei zum ausführbaren Programm (nach [Horstmann08])

Laufzeit-Fehler

Auch wenn ein Programm korrekt übersetzt wurde, können Fehler bei der Ausführung auftreten. Es kann zum Beispiel vorkommen, dass das Programm nicht das gewünschte oder erwartete Ergebnis erzeugt; in solchen Fällen müssen die verwendeten Algorithmen und Verfahren bzw. ihre Umsetzung im Quelltext untersucht werden (siehe Abb. 6.2-1).

Während der Programmausführung können auch Fehler auftreten, die das Programm sofort mit einer Fehlermeldung beenden. Sie werden oft durch Eingaben ausgelöst, deren Verarbeitung das Programm in einen problematischen Zustand bringt. Ein typisches Beispiel für einen solchen Fehler ist eine Division durch Null.

Beispiel 6.2-3: Fehler bei der Ausführung

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MeineKlasse.main(MeineKlasse.java:8)
```

Die erste Zeile gibt an, was für ein Fehler aufgetreten ist. Die restlichen Zeilen geben an, in welcher Methode der Fehler aufgetreten ist.

⌟

Bei einem solchen Fehler muss die Ursache gefunden, der Quelltext korrigiert und anschließend übersetzt und ausgeführt werden (siehe Abb. 6.2-1).

Bemerkung 6.2-2: Übersetzungsfehler vs. Laufzeitfehler
Übersetzungsfehler sind Fehler im Programm, die während der Übersetzung entdeckt werden. Ein Beispiel für Übersetzungsfehler sind syntaktische Fehler. Bei der Ausführung können sowohl Fehler auftreten, die das Programm zum Abbruch zwingen, als auch Fehler im Sinne von falschen Ergebnissen oder fehlerhaftem Verhalten.

⌟

Selbsttestaufgabe 6.2-1:

Führen Sie die folgenden Aufgaben mit den verschiedenen auf der Webseite vorgestellten Varianten zur Entwicklung von Java-Programmen aus. Überlegen Sie sich anschließend, welche der Varianten Sie in Zukunft verwenden wollen.

◊

Selbsttestaufgabe 6.2-2:

Laden Sie die Datei `MeineErsteKlasse.java` *von der Kurswebseite herunter, übersetzen Sie diese und führen Sie die enthaltene* `main()`*-Methode aus. Was für eine Ausgabe erhalten Sie? Versuchen Sie, auch die zugehörige* `class`*-Datei zu finden und öffnen Sie diese. Was erkennen Sie dabei? Vergleichen Sie den Quelltext mit dem Verhalten des Programms.*

◊

Selbsttestaufgabe 6.2-3:

Laden Sie die Datei `FehlerhafteKlasse.java` *von der Kurswebseite herunter, finden und beheben Sie die Fehler. Übersetzen Sie die Klasse anschließend und führen Sie sie aus. Was passiert, wenn Sie* `3` *oder* `10` *als Argument übergeben?*

◊

7 Zusammenfassung

Mit den vorangegangenen Überlegungen haben wir versucht, Ihnen das Denken in Objekten und die Formulierung von Algorithmen mit Hilfe von Kontrollstrukturen näher zu bringen. Nachdem wir dieses Denken verinnerlicht haben, werden wir daraus in den folgenden Kurseinheiten eine Programmiertechnik entwickeln, die wir konkret am Beispiel der objektorientierten Programmiersprache Java verdeutlichen werden.

Wir fassen also noch einmal zusammen:

In der Welt der objektorientierten Programmierung werden konkrete und abstrakte Gegenstände durch **Objekte** modelliert.

Alle Objekte sind **Exemplare** einer **Klasse**. Die Klasse definiert die Eigenschaften und Fähigkeiten jedes Objekts dieser Klasse.

Die Eigenschaften werden durch eine Menge von **Attributen** in der Klassendefinition bestimmt.

Die jeweiligen **Attributwerte** der Attribute eines Objekts bestimmen seinen inneren **Zustand**.

Die Fähigkeiten oder das **Verhalten** aller Objekte der Klasse werden durch die **Methoden** bestimmt.

Die Beziehungen zwischen Klassen werden mit Hilfe von **Assoziationen** und **Multiplizitäten** modelliert.

Eine Klasse kann die **Spezialisierung** einer oder mehrerer anderen Klasse sein und somit deren Eigenschaften und Verhalten erben.

Die Programmierung von Klassen mit ihren Attributen und Methoden werden wir im Einzelnen in den folgenden Kurseinheiten erörtern. Dazu ist es notwendig, eine ganze Reihe programmiersprachlicher Begriffe zu erlernen und einzuüben.

Wir haben in dieser ersten Kurseinheit gesehen, wie die objektorientierte Programmierung fachliche und datenverarbeitungstechnische Begriffe vereinheitlicht. Die Begriffe der Anwendung dienen als Grundlage zur Modellierung einer programmtechnischen Lösung.

Das Anwendungsmodell kann ohne gedanklichen Bruch in ein objektorientiertes Programm überführt werden. Die Änderung, die Erweiterung und die Wiederverwendung aller mit der programmtechnischen Lösung zusammenhängenden Dokumente werden damit unterstützt.

Zudem haben wir uns mit der Analyse einer gegebenen Aufgabe mit Hilfe von **Anwendungsfällen** beschäftigt. Mit Hilfe von **UML-Anwendungsfalldiagrammen**

werden das System selbst, **Akteure**, **Anwendungsfälle** und die Beziehungen zwischen diesen Elementen dargestellt. Danach folgt in der Regel eine **Substantivanalyse**, die uns zu einem ersten Klassenmodell, zum Beispiel in Form eines **UML-Klassendiagramms** führt. Konkrete Objekte und deren **Verknüpfungen** lassen sich mit Hilfe von **UML-Objektdiagrammen** darstellen.

Um nicht nur die Struktur und Beziehungen der Klassen beschreiben zu können, haben wir grundlegende Kontrollstrukturen wie **Sequenz**, **Verzweigung** und **Schleife** sowie ihre Darstellung in **UML-Aktivitätsdiagrammen** kennen gelernt. Zur Darstellung von Lösungsverfahren für Problemklassen haben wir **Algorithmen** und ihre Eigenschaften kennen gelernt.

Als Einstieg für die Programmierung mit Java in den folgenden Kurseinheiten haben wir uns schon einmal mit dem Übersetzen und Ausführen von Java-Programmen auseinandergesetzt.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 1.2-1:

Überprüfen Sie Ihre Lösung mit der interaktiven Übung auf der Webseite.

Lösung zu Selbsttestaufgabe 2.2-1:

Mögliche Handlungsträger sind die Chefin, Verkäufer, Großlieferanten sowie Kunden und Premiumkundinnen. Wichtige Geschäftsabläufe sind unter anderem der Verkauf, Bestellungen beim Lieferanten und Lieferungen an Premiumkundinnen. Probleme können z. B. auftreten, wenn eine Bezahlung fehlschlägt oder sich im Sortiment des Lieferanten eine Veränderung ergibt. Wir werden in den folgenden Abschnitten einige dieser Fragen noch in Beispielen aufgreifen.

Lösung zu Selbsttestaufgabe 2.5-1:

Eine weitere Ergänzung zu den schon bisher dargestellten Anwendungsfällen ist die Bestellung eines Premiumkunden. Diese ist in Abb. ML 1 dargestellt.

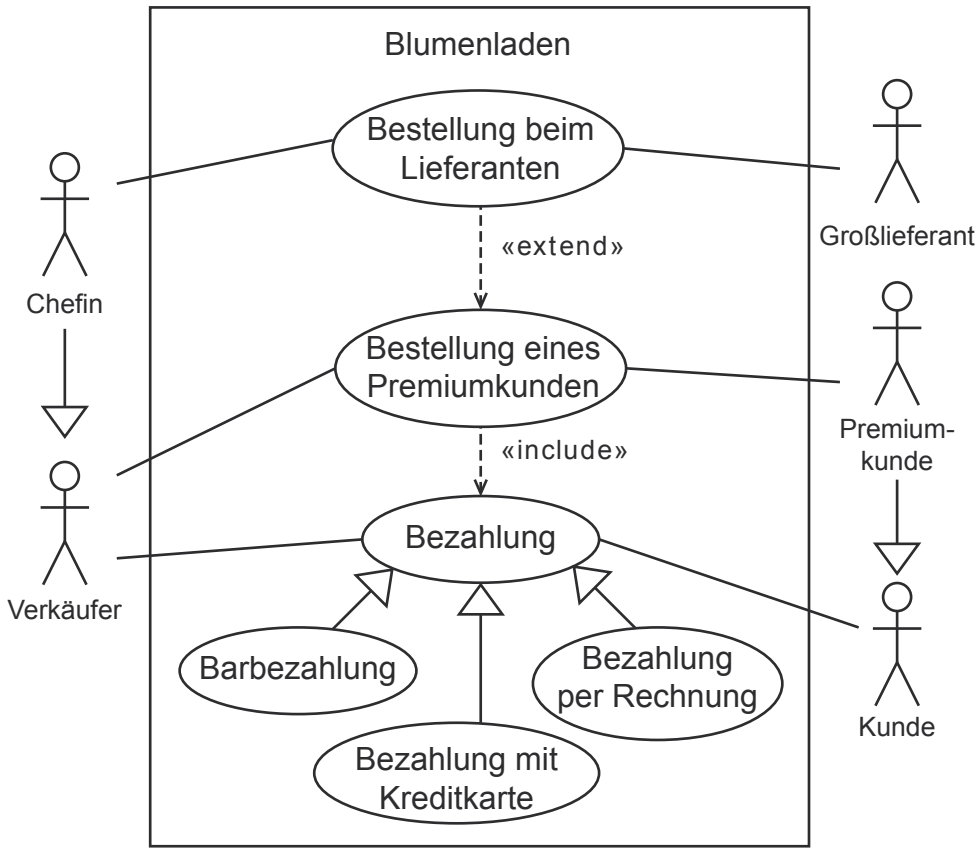

Abb. ML 1: Anwendungsfalldiagramm für die Fallstudie

Lösung zu Selbsttestaufgabe 2.6-1:

Tab. ML 1: Anwendungsfall „Artikelverkauf“

Anwendungsfall	Artikelverkauf	
Kurzbeschreibung	Eine Kundin kauft bei einem Verkäufer im Laden einen Artikel. Die Bezahlung erfolgt Bar oder mit Kreditkarte, oder bei einem Betrag von mehr als 100 Euro auch per Rechnung.	
Beteiligte Akteure	Vk Kd	Verkäufer Kundin
Vorbedingung	Der gewünschte Artikel muss auf Lager sein.	
Nachbedingung	Der Artikel wurde an die Kundin ausgehändigt und die Bezahlung ist erfolgt.	
Auslöser	Besuch einer Kundin im Laden	
Standardszenario		
1.	Kd	Die Kundin nennt die gewünschten Artikel.
2.	Vk	Der Verkäufer stellt fest, dass die Artikel in ausreichender Menge auf Lager sind.
3.	Vk	Der Verkäufer berechnet den Preis.
4.	Kd	Die Kundin bezahlt bar (→ Anwendungsfall Barbezahlung).
5.	Vk	Der Verkäufer händigt die Ware der Kundin aus.
Alternative Szenarien		
zu 4.	Kd	Die Kundin bezahlt mit Kreditkarte (→ Anwendungsfall Bezahlung mit Kreditkarte).
zu 4.	Vk	Der Preis beträgt mehr als 100 Euro und der Verkäufer händigt eine Rechnung aus (→ Anwendungsfall Bezahlung per Rechnung).
zu 6.	Kd	Die Kundin begleicht die Rechnung innerhalb von 14 Tage per Überweisung.
Erweiterungen		
zu 6.	Kd	Die Kundin begleicht die Rechnung nicht innerhalb von 14 Tage und erhält eine Mahnung (→ Anwendungsfall Mahnung).
Fehlersituationen		

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite		
zu 2.	Vk	Artikel sind nicht in ausreichender Menge vorhanden; die Chefin muss eine Bestellung veranlassen (→ Anwendungsfall Bestellung beim Großlieferanten); Verkäufer erfragt voraussichtliches Lieferdatum und teilt es der Kundin mit (→ Anwendungsfall Lieferdatum erfragen); Anwendungsfall wird abgebrochen
zu 4.	Kd	Bezahlung schlägt fehl; Anwendungsfall wird abgebrochen

Lösung zu Selbsttestaufgabe 3.2-1:

Abb. ML 2 zeigt eine mögliche Lösung für die Artikelhierarchie. Die Hierarchie ist als UML-Klassendiagramm (siehe Abschnitt 3.5) dargestellt.

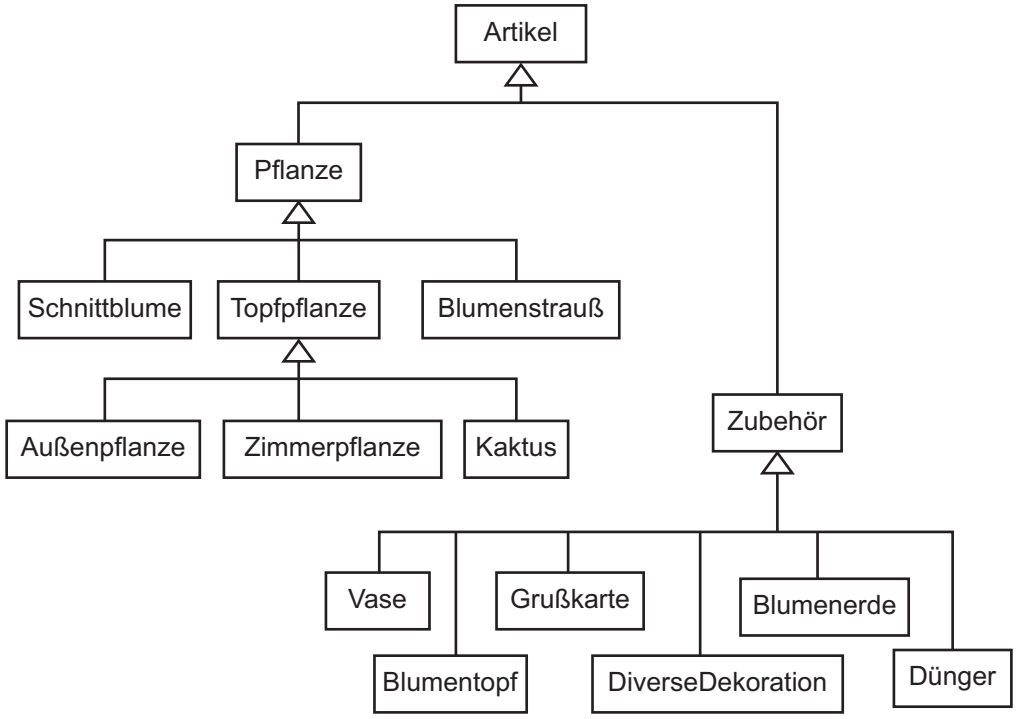

Abb. ML 2: Artikelhierarchie

Lösung zu Selbsttestaufgabe 3.2-2:

Bei Rose, Artikel, Rechnung, Kunde und Schnittblume handelt es sich um Klassen. passeBeschreibungAn, überprüfeLagerBestand und ändereName sind Methoden. Name, Farbe, Rechnungsnummer, Adresse und Preis sind Attribute. Hanna

Meier, „Zur Ecke 17, 12345 Irgendwo“, Rot und 31221 sind konkrete Attributwerte. (Hanna Meier könnte man auch als die Bezeichnung eines konkreten Objekts interpretieren).

Lösung zu Selbsttestaufgabe 3.4-1:

Weitere Klassenkandidaten sind zum Beispiel Verkäufer, Kunde, Premiumkunde. Außer der Kundenummer werden nicht viele Attribute erwähnt, so dass gerade bei den Artikeln und Rechnungen noch einmal Details erfragt werden müssen. Das Erfragen eines Lieferdatums ist ein Beispiel für eine Methode. Die Beziehung zwischen Kunde und Rechnung wird nur implizit erwähnt. Daran können Sie erkennen, dass eine Substantivanalyse sicherlich keine erschöpfende Möglichkeit zur Identifikation von Klassen, Attributen, Methoden und Assoziationen ist.

Lösung zu Selbsttestaufgabe 3.5-1:

Abb. ML 3 zeigt eine UML-Klassendiagramm mit den genannten Figuren. Aus Gründen der Übersichtlichkeit haben wir die Attribute und Methoden im Diagramm nicht dargestellt. Mögliche Methoden für die Klasse `GeometrischeFigur` sind `berechneFläcHeninhalt()` und `berechneUmfang()`. Ein Attribut der Klasse `Kreis` ist zum Beispiel `radius`.

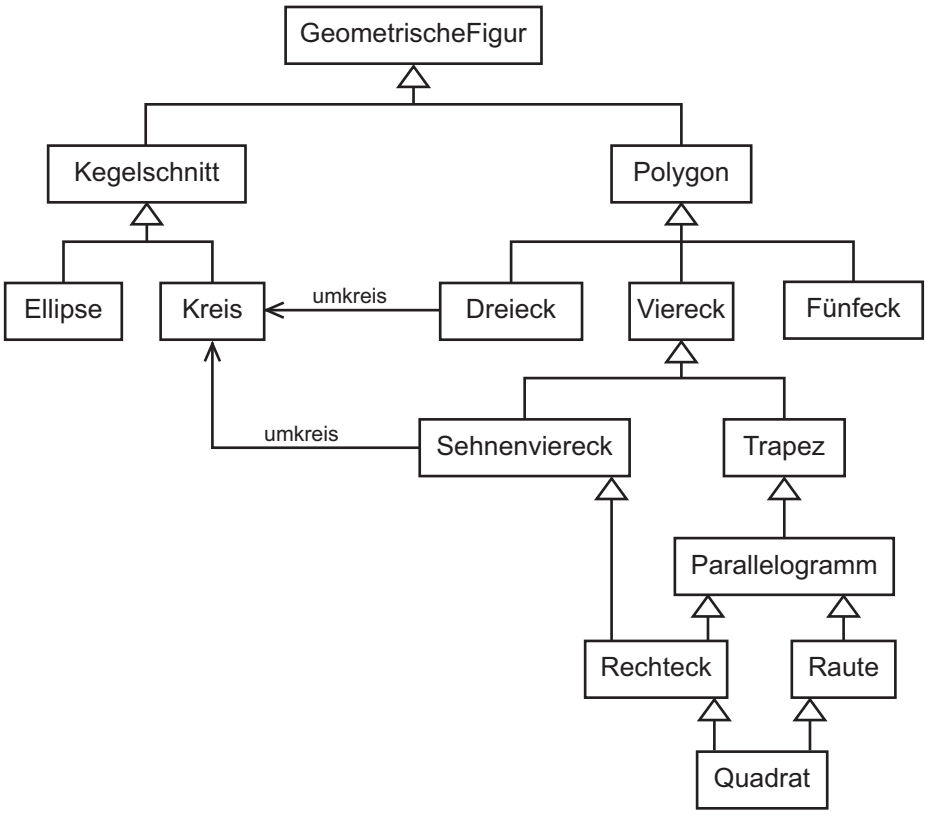

Abb. ML 3: Eine Klassendiagramm für geometrische Figuren

Lösung zu Selbsttestaufgabe 3.6-1:

Abb. ML 4 zeigt das mögliche Objektdiagramm.

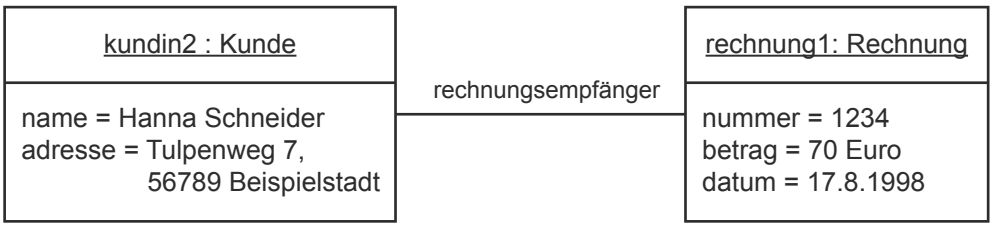

Abb. ML 4: Objektdiagramm mit Kundin und Rechnung

Lösung zu Selbsttestaufgabe 4.1-1:

Wir sprechen von einem *deterministischen* Ablauf, wenn die Schrittfolge eines Algorithmus eindeutig vorgegeben ist. Mit den Kontrollstrukturen, die wir in der vorangehenden Kurseinheit einführten, können wir nur deterministische Abläufe beschreiben.

Ein Algorithmus liefert ein *determiniertes* Ergebnis, wenn bei vorgegebener Eingabe ein eindeutiges Ergebnis geliefert wird und bei gleicher Eingabe bei mehrfacher Anwendung des Algorithmus das Ergebnis gleich lautet. Der Euklidische Algorithmus liefert ein determiniertes Ergebnis. Ein Algorithmus, der Pseudozufallszahlen erzeugt, liefert naturgemäß ein nicht-determiniertes Ergebnis.

Lösung zu Selbsttestaufgabe 4.1-2:

a) a = 21 und b = 6

1. Vergleich wird ausgeführt, aber keine Vertauschung vorgenommen → 2.
2. c = 3 → 3.
3. c ist ungleich 0 → 4.
4. a = 6 und b = 3 → 5.
5. → 2.
2. c = 0 → 3.
3. c ist 0 → Ende: ggT = b = 3

b) a = 78 und b = 273

1. Vergleich wird ausgeführt und Vertauschung vorgenommen a = 273 und b = 78 → 2.
2. c = 39 → 3.
3. c ist ungleich 0 → 4.
4. a = 78 und b = 39 → 5.
5. → 2.

2. c = 0 → 3.
3. c ist 0 → Ende: ggT = b = 39

c) a = 13 und b = 31

1. Vergleich wird ausgeführt und Vertauschung vorgenommen a = 31 und b = 13 → 2.
2. c = 5 → 3.
3. c ist ungleich 0 → 4.
4. a = 13 und b = 5 → 5.
5. → 2.
2. c = 3 → 3.
3. c ist ungleich 0 → 4.
4. a = 5 und b = 3 → 5.
5. → 2.
2. c = 2 → 3.
3. c ist ungleich 0 → 4.
4. a = 3 und b = 2 → 5.
5. → 2.
2. c = 1 → 3.
3. c ist ungleich 0 → 4.
4. a = 2 und b = 1 → 5.
2. c = 0 → 3.
3. c ist 0 → Ende: ggT = b = 1

d) a = 64 und b = 34

1. Vergleich wird ausgeführt, aber keine Vertauschung vorgenommen → 2.
2. c = 30 → 3.
3. c ist ungleich 0 → 4.
4. a = 34 und b = 30 → 5.
5. → 2.
2. c = 4 → 3.
3. c ist ungleich 0 → 4.
4. a = 30 und b = 4 → 5.
5. → 2.
2. c = 2 → 3.
3. c ist ungleich 0 → 4.
4. a = 4 und b = 2 → 5.
5. → 2.
2. c = 0 → 3.
3. c ist 0 → Ende: ggT = b = 2

Lösung zu Selbsttestaufgabe 4.1-3:

Man kann leicht zeigen, dass der klassische Euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen a und b terminiert:

Wir müssen zeigen, dass c irgendwann den Wert 0 annimmt, da genau dann der Algorithmus terminiert.

Wir zeigen zunächst, dass sich die Werte von a und b immer weiter verringern. Spätestens nach dem ersten Schritt gilt, dass a > b ist.

Der Rest c der Division a / b ist immer kleiner als a und als b. Es gilt also a > b, a > c, b > c und $c \geq 0$.

Da in Schritt 4 a den Wert von b zugewiesen bekommt und b den Wert von c, werden die beiden Zahlen im Laufe der Zeit immer kleiner.

Da der Rest einer Division niemals negativ ist nähert sich c im Laufe der Zeit dem Wert 0 an, spätestens wenn b den Wert 1 hat. Dies hat zur Folge, dass der Algorithmus dann terminiert.

Unsere Argumentation ist jedoch nur korrekt unter der Annahme, dass keine der beiden Zahlen den Wert 0 hat. In diesem Fall würde Schritt 2 fehlschlagen.

Lösung zu Selbsttestaufgabe 4.1-4:

Eine mögliche Lösung ist:

1. wenn a = 1 ist, Ende: a ist keine Primzahl
2. wenn a = 2 ist, Ende: a ist eine Primzahl
3. n sei die auf die nächste Ganzzahl abgerundete Wurzel von a
4. wenn n = 1 ist, Ende: a ist eine Primzahl
5. wenn n ein Teiler von a ist, Ende: a ist keine Primzahl
6. Verringere n um 1.
7. Fahre fort mit Schritt 4.

Lösung zu Selbsttestaufgabe 4.2-1:

Die Lösung ist in Abb. ML 5 als Aktivitätsdiagramm dargestellt.

Lösung zu Selbsttestaufgabe 4.2-2:

Der Algorithmus von Euklid ist in Abb. ML 6 als Aktivitätsdiagramm dargestellt. Er besitzt genau eine Schleife, alle zugehörigen Knoten sind grau markiert.

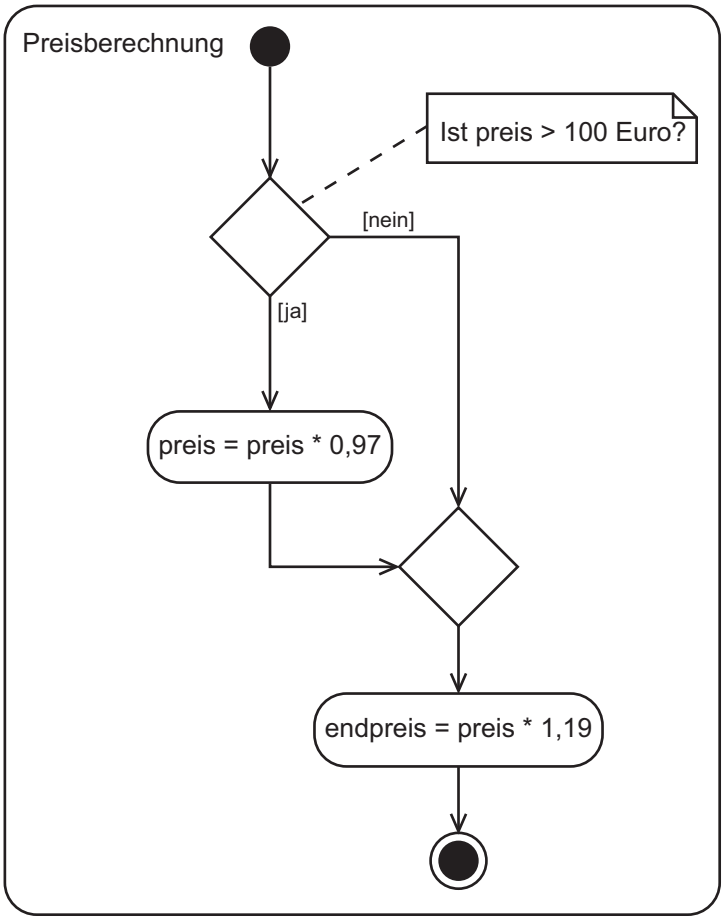

Abb. ML 5: Aktivitätsdiagramm Preisberechnung

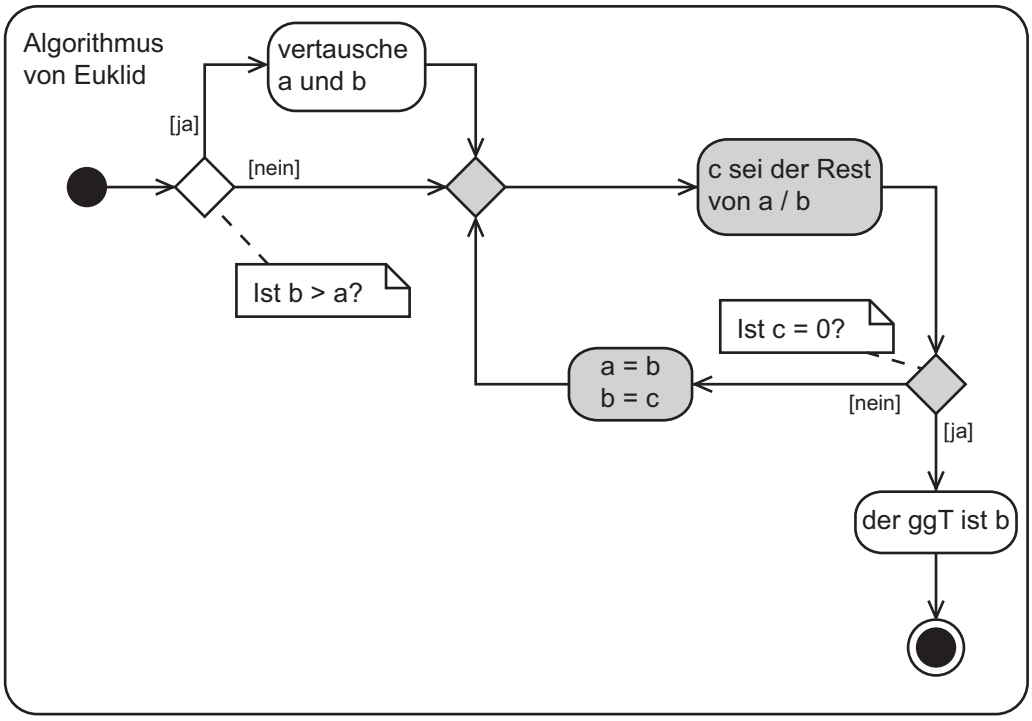

Abb. ML 6: Algorithmus von Euklid als Aktivitätsdiagramm

Lösung zu Selbsttestaufgabe 4.2-3:

Der Algorithmus aus der Lösung zu Selbsttestaufgabe 4.1-4 ist in Abb. ML 7 als Aktivitätsdiagramm dargestellt.

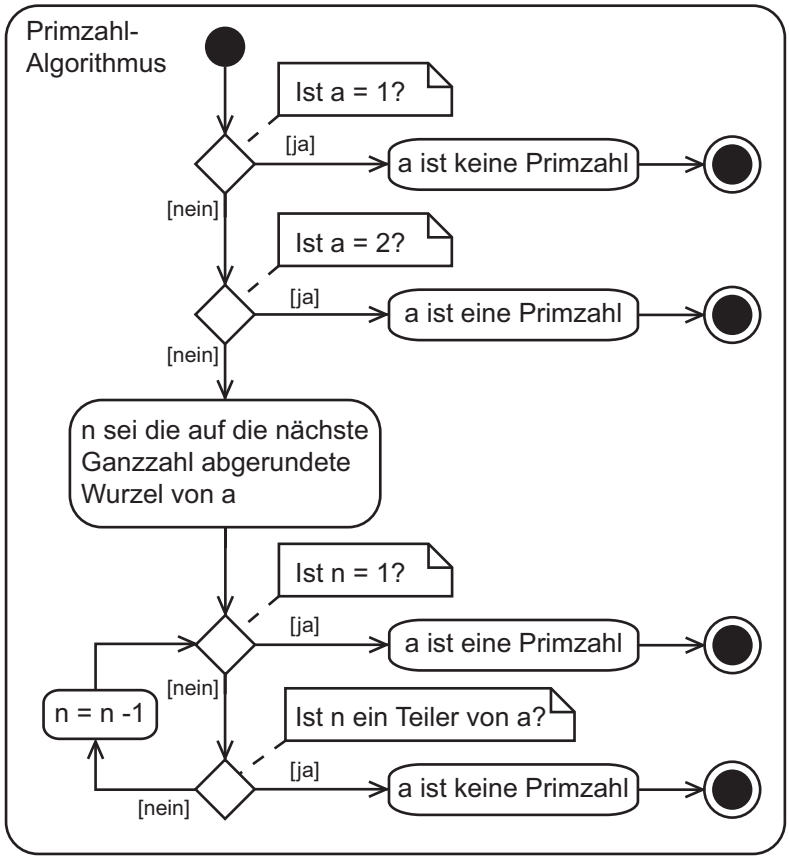

Abb. ML 7: Primzahl-Algorithmus als Aktivitätsdiagramm

Lösung zu Selbsttestaufgabe 5.1-1:

Das Geheimnisprinzip besagt, dass nur die Funktionalität eines Programmmoduls oder einer Klasse, d. h. die von diesem Modul oder der Klasse angebotenen Operationen bekannt gemacht wird. Die interne Realisierung der Operationen und auch der Aufbau der Datenstrukturen, auf denen die Operationen arbeiten, bleiben nach außen verborgen. Das Geheimnisprinzip stellt sicher, dass die Implementierungsdetails eines Moduls oder einer Klasse beliebig oft geändert werden, ohne dass andere Programmteile betroffen sind, solange die Schnittstellen des geänderten Moduls nicht betroffen sind.

Lösung zu Selbsttestaufgabe 6.2-1:

Es kann durchaus hilfreich sein, im Laufe des Kurses noch einmal andere Varianten auszuprobieren, da sich Ihre Ansprüche im Laufe der Zeit vielleicht ändern werden. Alle Aufgaben sind mit allen Varianten lösbar. Das konkrete Wissen über die verschiedenen Varianten ist nicht klausurrelevant.

Lösung zu Selbsttestaufgabe 6.2-2:

Sie erhalten die Ausgabe:

```
Hallo bei der ersten Java-Klasse!
a = 8
b = 9
c = a - b = -1
d = a * b = 72
```

In der `class`-Datei können Sie unter anderem den Namen der Klasse erkennen.

Lösung zu Selbsttestaufgabe 6.2-3:

Am Ende von Zeile 9 fehlt ein Semikolon. Zudem hat sich in Zeile 12 bei der Variablen `ergebnis` ein Tippfehler eingeschlichen.

Wenn Sie `3` als Argument übergeben, erhalten Sie `0` als Ausgabe. Wenn Sie `10` als Argument übergeben, kommt es zu einem Laufzeitfehler, da eine Division durch 0 in Zeile 12 ausgeführt wird.