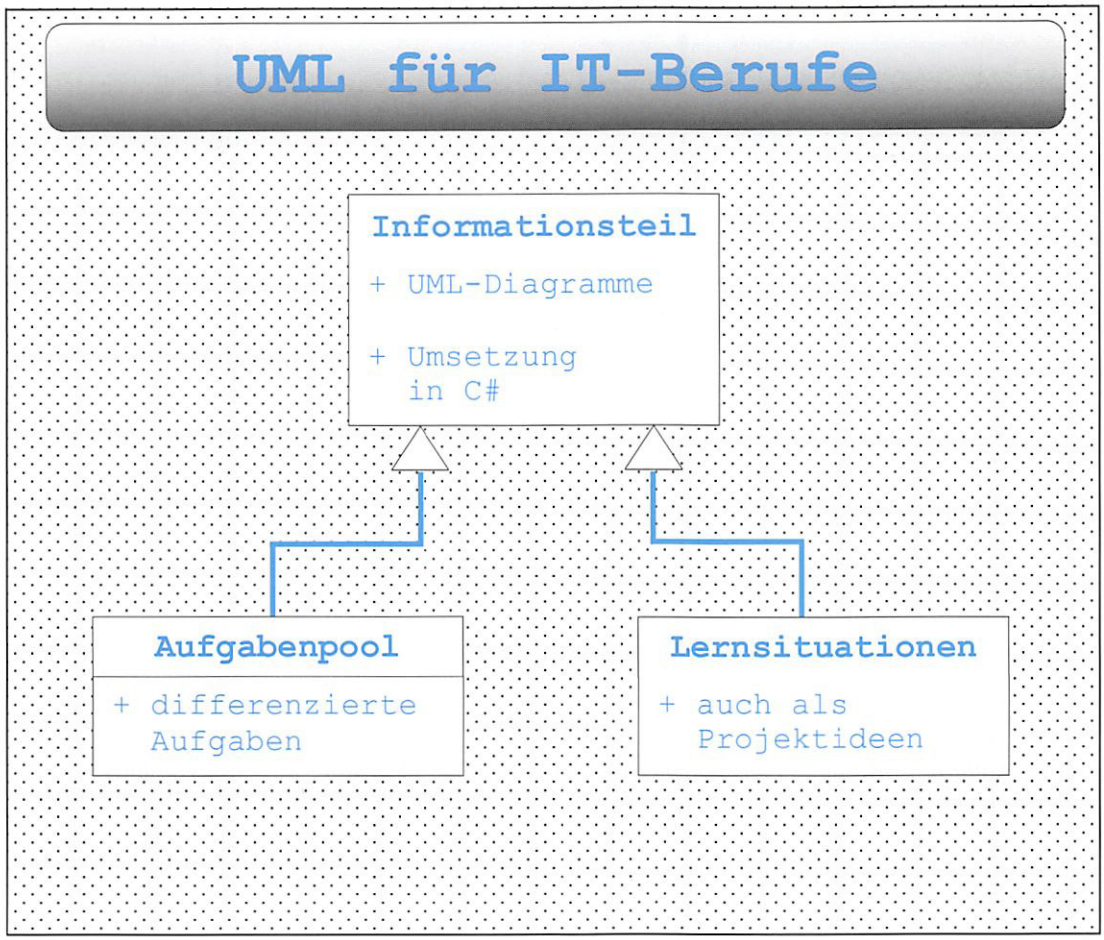
EUROPA-FACHBUCHREIHE
Programmierung für die
IT-Ausbildung



2. Auflage

VERLAG EUROPA-LEHRMITTEL · Nourney, Vollmer GmbH & Co. KG
Düsselberger Str. 23 · 42781 Haan-Gruiten

Europa-Nr.: 85580

Verfasser:
Dirk Hardy, 46049 Oberhausen

2. Auflage 2019

Druck 5 4 3 2 1

Alle Drucke derselben Auflage sind parallel einsetzbar, da sie bis auf die Behebung von Druckfehlern untereinander unverändert sind.

ISBN 978-3-8085-8588-7

Alle Rechte vorbehalten. Das Werk ist urheberrechtlich geschützt. Jede Verwertung außerhalb der gesetzlich geregelten Fälle muss vom Verlag schriftlich genehmigt werden.

©2019 by Verlag Europa-Lehrmittel, Nourney, Vollmer GmbH & Co. KG, 42781 Haan-Gruiten
http://www.europa-lehrmittel.de
Satz: Reemers Publishing Services GmbH, Krefeld
Umschlag: braunwerbeagentur, 42477 Radevormwald
Umschlagfotos: envfx-fotolia.com; Gina Sanders-fotolia.com; Gordon Bussiek-fotolia.com
Druck: Medienhaus Plump GmbH, 53619 Rheinbreitbach

Vorbemerkung

Die moderne Softwareentwicklung geht weit über das Entwickeln von Algorithmen und die Umsetzung in eine Programmiersprache hinaus. Vielmehr geht es um eine umfassende Planung aller beteiligten Komponenten sowie eine solide Dokumentation der umzusetzenden Prozesse. Dieser Entwicklungsprozess kann durch die formale Sprache **UML** (**U**nified **M**odeling **L**anguage) hervorragend unterstützt werden. Dabei können die verschiedenen Diagramme der UML helfen, das geplante Softwaresystem in allen Phasen der Entwicklung zu beschreiben und damit auch die Grundlage für die Implementierung zu legen. Zusätzlich kann der Entwicklungsprozess verbessert werden, indem sogenannte CASE-Tools (**C**omputer-**A**ided-**S**oftware-**E**ngineering-Tools) eingesetzt werden. Damit ist nicht nur die Erstellung von UML-Diagrammen, sondern auch eine automatische Codeerzeugung sowie ein Reverse-Engineering möglich. Der Entwicklungsprozess wird dadurch professioneller und effizienter. Allerdings erfordert es eine hinreichende Einarbeitung in diese CASE-Tools. Für die Ausbildung im IT-Bereich ist die Auseinandersetzung gerade mit diesen Techniken ein sehr wichtiger Aspekt.

Aufbau des Buches

Das vorliegende Buch möchte die formale Sprache UML möglichst anschaulich, praxis- und unterrichtsnah vermitteln. Damit verfolgt dieses Buch einen **praktischen Ansatz**. Es ist die Ansicht des Autors, dass gerade in der schulischen Ausbildung der Zugang zu den komplexen Themen der Softwareentwicklung verstärkt und durch anschauliche und praktische Umsetzung vorbereitet werden muss. Neben dem Erlernen von wichtigen UML-Diagrammen wird ein besonderer Schwerpunkt auf die Umsetzung bestimmter Diagramme in die Programmiersprache C# gelegt. Damit soll die Lücke, die oftmals zwischen den eher theoretischen Konstrukten der UML und den eher praktischen Umsetzungen in den Programmiersprachen herrscht, geschlossen werden.

Das Buch ist in **drei Teile** gegliedert.

Der **erste Teil** des Buches dient als **Informationsteil** und bietet eine **systematische Einführung in die formale Sprache UML sowie die Umsetzungen in die Programmiersprache C#.**

Der **zweite Teil** des Buches ist eine **Sammlung von Übungsaufgaben**. Nach der Erarbeitung der entsprechenden Kenntnisse aus dem Informationsteil können die Aufgaben aus diesem Teil zur weiteren Auseinandersetzung mit den Themen dienen und durch verschiedene Schwierigkeitsgrade auch die Differenzierung im Unterricht ermöglichen.

Der **dritte Teil** des Buches beinhaltet **Lernsituationen** basierend auf dem Lernfeld „Entwickeln und Bereitstellen von Anwendungssystemen" aus dem Rahmenlehrplan für die IT-Berufe (speziell Fachinformatiker-Anwendungsentwicklung). Lernsituationen konkretisieren sich aus den Lernfeldern und sollen im Idealfall vollständige Handlungen darstellen (Planen, Durchführen, Kontrollieren). Aus diesem Grund werden die Lernsituationen so angelegt, dass neben einer Planungsphase nicht nur die Durchführung im Blickpunkt steht, sondern auch geeignete Testverfahren zur Kontrolle des Entwicklungsprozesses in die Betrachtung einbezogen werden. Die Lernsituationen können aber auch als **Projektideen** verstanden werden.

Das Buch ist für alle berufsbezogenen Ausbildungsgänge im IT-Bereich konzipiert. Durch die differenzierten Aufgabenstellungen kann es in allen IT-Berufen (speziell Fachinformatiker), aber auch von den informationstechnischen Assistenten genutzt werden.

In diesem Buch werden einige Diagramme der UML mit dem CASE-Tool **StarUML** erstellt. Das Tool wird kostenfrei zum Download (als Evaluierungs-Edition) angeboten. Für die Umsetzung in die Programmiersprache dient die Entwicklungsumgebung von Microsoft (Community-Edition für C#). Diese Entwicklungsumgebung ist ebenfalls kostenfrei als Download im Internet verfügbar.

Für Anregungen und Kritik zu diesem Buch sind wir Ihnen dankbar (gerne auch per E-Mail).

Dirk Hardy

E-Mail: Hardy@DirkHardy.de

Im Frühjahr 2019

Verlag Europa-Lehrmittel

E-Mail: Info@Europa-Lehrmittel.de

Vorbemerkung 3

Aufbau des Buches 3

Teil 1 Einführung in UML 9

1 Grundbegriffe der UML und der objektorientierten Softwareentwicklung 11
1.1 Die Unified Modeling Language (UML) 11
1.1.1 Historische Entwicklung der UML 11
1.1.2 Diagrammtypen der UML 13
1.2 Grundbegriffe der objektorientierten Softwareentwicklung 13
1.2.1 Klassen und Objekte 13
1.2.2 Generalisierung und Spezialisierung 15
1.2.3 Polymorphie 16
1.2.4 Entwurfsmuster 18
1.2.5 Architekturmuster 18

2 Das Anwendungsfalldiagramm 20
2.1 Anwendungsfälle und Akteure 20
2.1.1 Anwendungsfall 20
2.1.2 Akteur 21
2.2 Beziehungen zwischen Anwendungsfall und Akteur 22
2.2.1 Die ungerichtete Assoziation 22
2.2.2 Die gerichtete Assoziation 22
2.2.3 Multiplizität der Assoziation 23
2.3 Beziehungen zwischen Anwendungsfällen 23
2.3.1 Generalisierung und Spezialisierung 23
2.3.2 Die include-Beziehung 24
2.3.3 Die extend-Beziehung 25
2.4 Beziehungen zwischen Akteuren 26
2.4.1 Generalisierung und Spezialisierung 26

3 Das Klassendiagramm 28
3.1 Die Darstellung der Klasse 28
3.1.1 Grundlegender Aufbau 28
3.1.2 Beschreibung der Attribute 28
3.1.3 Beschreibung der Methoden 30
3.1.4 Umsetzung eines Klassendiagramms in eine objektorientierte Programmiersprache 31
3.2 Beziehungen zwischen Klassen 33
3.3 Die Assoziation 34
3.3.1 Allgemeiner Aufbau einer Assoziation 35
3.4 Umsetzung von Assoziationen 37
3.4.1 Umsetzung einer bidirektionalen Assoziation in C# 37
3.4.2 Umsetzung einer unidirektionalen Assoziation in C# 38
3.5 Die Aggregation 41
3.5.1 Allgemeiner Aufbau einer Aggregation 42
3.5.2 Umsetzung einer *0..1:1*-Aggregation in C# 42
3.5.3 Umsetzung einer **:**-Aggregation in C# 44
3.6 Die Komposition 46
3.6.1 Allgemeiner Aufbau einer Komposition 46
3.6.2 Umsetzung einer *1:1..5*-Komposition in C# 47
3.7 Die Generalisierung und Spezialisierung 49
3.7.1 Sichtbarkeit von Attributen 50
3.7.2 Mehrfachgeneralisierung 50
3.7.3 Umsetzung einer einfachen Generalisierung in C# 51
3.7.4 Abstrakte Basis-Klassen 52

3.8 Stereotype ... 53
3.8.1 Primitive und einfache Datentypen ... 53
3.8.2 Umsetzung eines einfachen Datentyps in C# ... 54
3.8.3 Aufzählungen ... 54
3.8.4 Schnittstellen ... 55
3.8.5 Umsetzung einer Schnittstelle in C# ... 56

4 Das Objektdiagramm ... 58
4.1 Die Darstellung eines Objektes ... 58
4.1.1 Grundlegender Aufbau ... 58
4.1.2 Klassen und Objekte gemeinsam darstellen ... 58
4.2 Beziehungen zwischen Objekten ... 58
4.2.1 Der Link ... 59
4.3 Umsetzung eines Objektdiagramms ... 60
4.3.1 Umsetzungen des Beispiels in C# ... 61

5 Das Sequenzdiagramm ... 63
5.1 Allgemeine Darstellung ... 63
5.1.1 Der Interaktionsrahmen ... 63
5.1.2 Lebenslinien ... 63
5.1.3 Aktivitäten ... 64
5.1.4 Nachrichten ... 64
5.2 Fragmente ... 66
5.2.1 Alternativen ... 66
5.2.2 Parallele Ausführung ... 67
5.2.3 Schleifen ... 67
5.2.4 Weitere Fragmente ... 68
5.3 Umsetzung eines Sequenzdiagramms ... 68
5.3.1 Umsetzung des Sequenzdiagramms in C# ... 68

6 Das Aktivitätsdiagramm ... 71
6.1 Allgemeine Darstellung ... 71
6.1.1 Die Aktion ... 71
6.1.2 Steuerungsfluss ... 71
6.1.3 Verzweigungen ... 71
6.1.4 Aktivitäten ... 72
6.1.5 Start- und Endpunkte ... 73
6.1.6 Verantwortungsbereiche ... 75
6.1.7 Gabelungen und Vereinigungen ... 76
6.2 Besondere Kommunikation ... 76
6.2.1 Objekte und Objektfluss ... 76
6.2.2 Signale senden ... 78
6.2.3 Unterbrechungen ... 79
6.3 Selektion und Iteration ... 81
6.3.1 Die Selektion ... 81
6.3.2 Die Iteration ... 82
6.3.3 Expansionsbereiche ... 83
6.4 Umsetzung eines Aktivitätsdiagramms ... 83
6.4.1 Umsetzung des Aktivitätsdiagramms in C# ... 85

7 Beispiel einer Softwareentwicklung ... 88
7.1 Nutzung eines CASE-Tools ... 88
7.1.1 Aspekte von CASE-Tools ... 88
7.1.2 Ein Modell mit StarUML anlegen ... 88
7.1.3 Diagramme in StarUML anlegen ... 89
7.2 Beispiel einer objektorientierten Softwareentwicklung mit UML und C# ... 92
7.2.1 Anforderungen mit einem Anwendungsfalldiagramm beschreiben ... 92
7.2.2 Objektorientierte Analyse (OOA) ... 93

7.2.3 Objektorientiertes Design (OOD) ... 95
7.2.4 Objektorientierte Programmierung (OOP) ... 98

8 Weitere UML-Diagramme ... 107
8.1 Strukturdiagramme ... 107
8.1.1 Das Kompositionsstrukturdiagramm ... 107
8.1.2 Das Komponentendiagramm ... 109
8.1.3 Verteilungsdiagramm ... 111
8.1.4 Paketdiagramm ... 113
8.1.5 Profildiagramm ... 116
8.2 Verhaltensdiagramme ... 116
8.2.1 Zustandsdiagramm ... 116
8.2.2 Kommunikationsdiagramm ... 118
8.2.3 Zeitverlaufsdiagramm ... 120
8.2.4 Interaktionsübersichtsdiagramm ... 123

Teil 2 Aufgabenpool ... 125

Aufgabenpool ... 126
1 Aufgaben zu den Grundbegriffen UML / OOP ... 126
2 Aufgaben zum Anwendungsfalldiagramm ... 127
3 Aufgaben zum Klassendiagramm ... 129
4 Aufgaben zum Objektdiagramm ... 131
5 Aufgaben zum Sequenzdiagramm ... 134
6 Aufgaben zum Aktivitätsdiagramm ... 136
7 Aufgaben zur Softwareentwicklung ... 138
8 Aufgaben zu den weiteren Diagrammen ... 138

Teil 3 Lernsituationen ... 145
Lernsituation 1: Erstellen einer Präsentation mit Hintergrundinformationen zur Sprache UML (in Deutsch oder Englisch) ... 146
Lernsituation 2: Anfertigen einer Dokumentation für den Einsatz eines CASE-Tools (in Deutsch oder Englisch) ... 147
Lernsituation 3: Entwicklung einer Software zur Darstellung von Wetterdaten mit dem Model-View-Controller-Konzept ... 148
Lernsituation 4: Durchführung einer objektorientierten Analyse und eines objektorientierten Designs zur Entwicklung eines Softwaresystems zur Verwaltung der Schulbibliothek eines Berufskollegs ... 151
Lernsituation 5: Entwicklung einer Software zur Verwaltung eines Schulungsunternehmens ... 152

Index ... 155

Teil 1
Einführung in UML

1.1 Die Unified Modeling Language (UML) 11
1.2 Grundbegriffe der objektorientierten Softwareentwicklung 13
2.1 Anwendungsfälle und Akteure 20
2.2 Beziehungen zwischen Anwendungsfall und Akteur 22
2.3 Beziehungen zwischen Anwendungsfällen 23
2.4 Beziehungen zwischen Akteuren 26
3.1 Die Darstellung der Klasse 28
3.2 Beziehungen zwischen Klassen 33
3.3 Die Assoziation 34
3.4 Umsetzung von Assoziationen 37
3.5 Die Aggregation 41
3.6 Die Komposition 46
3.7 Die Generalisierung und Spezialisierung 49
3.8 Stereotype 53
4.1 Die Darstellung eines Objektes 58
4.2 Beziehungen zwischen Objekten 58
4.3 Umsetzung eines Objektdiagramms 60
5.1 Allgemeine Darstellung 63
5.2 Fragmente 66
5.3 Umsetzung eines Sequenzdiagramms 68
6.1 Allgemeine Darstellung 71
6.2 Besondere Kommunikation 76
6.3 Selektion und Iteration 81
6.4 Umsetzung eines Aktivitätsdiagramms 83
7.1 Nutzung eines CASE-Tools 88
7.2 Beispiel einer objektorientierten Softwareentwicklung mit UML und C# 92
8.1 Strukturdiagramme 107
8.2 Verhaltensdiagramme 116

1 Grundbegriffe der UML und der objektorientierten Softwareentwicklung

1.1 Die Unified Modeling Language (UML)

Die Entwicklung von Software bzw. von Softwaresystemen ist ein schwieriger Prozess. Von der Problemstellung über die Planung bis zur Realisierung und dem Testen gibt es viele Klippen zu umschiffen. In der **objektorientierten Softwareentwicklung** kristallisieren sich drei wichtige Phasen bei der Entwicklung heraus:

- **Objektorientierte Analyse (OOA)**
  Analyse der Objekte und ihrer Beziehungen
- **Objektorientiertes Design (OOD)**
  Konzeption der entsprechenden Klassen und der Benutzeroberflächen aus den Vorgaben der Analyse
- **Objektorientierte Programmierung (OOP)**
  Implementierung der Klassen in einer Sprache wie C#

In allen Phasen unterstützt die **Unified Modeling Language (UML)** die Entwicklung der Software. Vor allem in der Planungsphase, der objektorientierten Analyse, hilft die UML bei der Beschreibung des zu erstellenden Softwaresystems. Ein solches Softwaresystem kann beispielsweise eine Datenbankanwendung, ein Grafikprogramm oder eine Workflow-Anwendung sein.

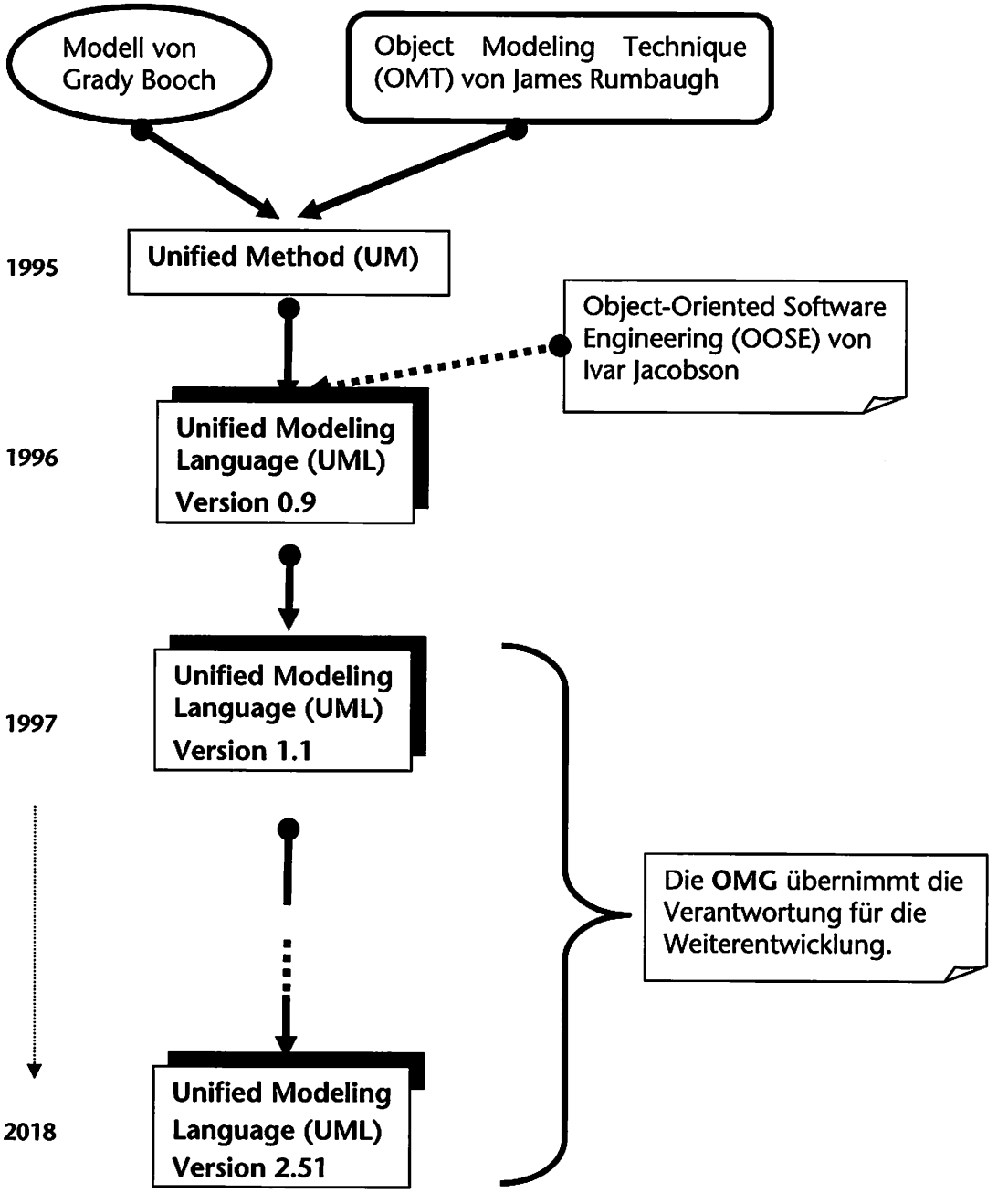
Die UML stellt dazu verschiedene Diagramme zur Verfügung, die wiederum verschiedene grafische Elemente enthalten. Innerhalb der UML gibt es allerdings für ein und denselben Sachverhalt manchmal mehrere Darstellungsarten.

1.1.1 Historische Entwicklung der UML

Bei den vielen Modellen zur Planung von objektorientierter Programmierung, die Anfang der 90er-Jahre existierten, gab es zwei besonders wichtige: das Modell von **Grady Booch** und die *Object Modeling Technique (OMT)* von **James Rumbaugh**.

Im Jahr 1995 wurden die beiden Modelle zur Unified Method (UM) zusammengefasst. Ein Jahr später wurde die erste Version der UML herausgegeben. Die Unified Method wurde dabei um die Methode von **Ivar Jacobson**, die *Object-Oriented Software Engineering (OOSE)*, ergänzt. Diese drei Modelle sind die Grundlage der UML. Neben den drei wichtigen „Gründern" der UML arbeiten heute viele große Softwareunternehmen daran, die Sprache UML weiterzuentwickeln und zu standardisieren. Die Hauptaufgabe übernimmt dabei die OMG (*Object Management Group*). Die OMG ist ein Konsortium aus Firmen, die sich für die Entwicklung von Standards in der objektorientierten Programmierung einsetzen. Das Konsortium besteht aus vielen wichtigen Firmen – unter anderem IBM, Oracle oder auch Microsoft, das erst 2008 in das Konsortium eingetreten ist.

Zeitliche Entwicklung der UML



Die drei Gründer der UML wurden auch als die drei *Amigos* bezeichnet. Booch arbeitete seit den 80er-Jahren bei der Firma **Rational Software** in Kalifornien. Rational Software ist ein Unternehmen mit dem Schwerpunkt Systemanalyse und -design. Ein besonderer Schwerpunkt war die objektorientierte Analyse und das objektorientierte Design. Dafür entwickelte die Firma auch spezielle Tools wie das Produkt **Rational Rose** (ein mächtiges Softwaredesignwerkzeug, das auf der UML basiert). 1994 trat James Rumbaugh in die Firma ein und entwickelte mit Grady Booch zusammen die **Unified Method** (UM). Ein Jahr später kam wegen der Übernahme der Firma **Objectory AB** durch Rational Software auch der Besitzer dieser Firma, Ivar Jacobson, zu Rational Software.

Damit war das Trio komplett und die drei *Amigos* entwickelten die erste Version der UML.

Die Firma Rational Software wurde 2003 von **IBM** übernommen, die Produkte laufen aber weiter unter den bekannten Namen.

1.1.2 Diagrammtypen der UML

Die UML in der aktuellen Version 2.51 enthält viele Diagrammtypen für die unterschiedlichen Anforderungen der objektorientierten Softwareentwicklung. Dabei können die Diagramme in zwei große Bereiche eingeteilt werden – in die Strukturdiagramme und Verhaltensdiagramme.

Strukturdiagramme

- ✓ **Klassendiagramm**
- ✓ **Objektdiagramm**
- ✓ Kompositionsstrukturdiagramm
- ✓ Komponentendiagramm
- ✓ Verteilungsdiagramm
- ✓ Paketdiagramm
- ✓ Profildiagramm

Verhaltensdiagramme

- ✓ **Anwendungsfalldiagramm**
- ✓ **Sequenzdiagramm**
- ✓ **Aktivitätsdiagramm**
- ✓ Zustandsdiagramm
- ✓ Kommunikationsdiagramm
- ✓ Zeitverlaufsdiagramm
- ✓ Interaktionsübersichtsdiagramm

Die Strukturdiagramme beschreiben das zu entwickelnde System in statischer Hinsicht. Die Elemente des Systems werden zeitunabhängig (also nicht dynamisch) beschrieben. Hingegen modellieren die Verhaltensdiagramme das zu entwickelnde System in dynamischer Hinsicht.

Hinweis:

Die Ziele der weiteren Kapitel werden keine ausführlichen Einführungen in alle Diagrammtypen sein. Es werden Schwerpunkte auf einige der Diagramme gelegt (Fettdruck in der obigen Auflistung), die für die Entwicklung eines Softwaresystems besonders wichtig erscheinen. Die weiteren Diagrammtypen werden aber trotzdem kurz vorgestellt.

1.2 Grundbegriffe der objektorientierten Softwareentwicklung

1.2.1 Klassen und Objekte

Im Mittelpunkt der objektorientierten Softwareentwicklung steht das Objekt bzw. die Klasse. Eine Klasse kann als eine Art Bauplan betrachtet werden, mit dem Objekte gebildet werden können. Die Objekte besitzen Eigenschaften (Attribute) und Methoden. Diese wichtigen Begriffe sollen nun näher betrachtet werden.

Was ist ein Objekt?

Ein Objekt ist eine softwaretechnische Repräsentation eines realen oder gedachten, klar abgegrenzten Gegenstandes oder Begriffs. Das Objekt erfasst alle Aspekte des Gegenstandes durch Attribute (Eigenschaften) und Methoden.

Was sind Attribute und Methoden?

Attribute sind die Eigenschaften des Objektes. Sie beschreiben den Gegenstand vollständig. Attribute sind geschützt gegen Manipulation von außen (das nennt man Kapselung oder Geheimnisprinzip). Methoden beschreiben die Operationen, die mit dem Objekt (bzw. seinen Attributen) durchgeführt werden können. Von außen erfolgt der Zugriff auf Attribute nur über die Methoden.

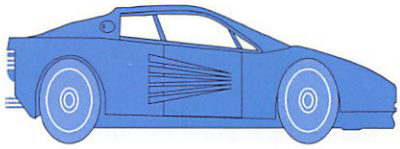
Was ist eine Klasse?

Unter einer Klasse versteht man die softwaretechnische Beschreibung eines Bauplanes für ein Objekt. Aus einer Klasse können dann Objekte abgeleitet (gebildet, instanziiert) werden.

Diese etwas abstrakten, aber wichtigen Begriffsdefinitionen sollen nun anhand eines Beispiels veranschaulicht werden.

Beispiel:

Diese Rennwagen sind konkrete Objekte. Sie haben Attribute wie Farbe, Leistung in kW und den Hubraum.

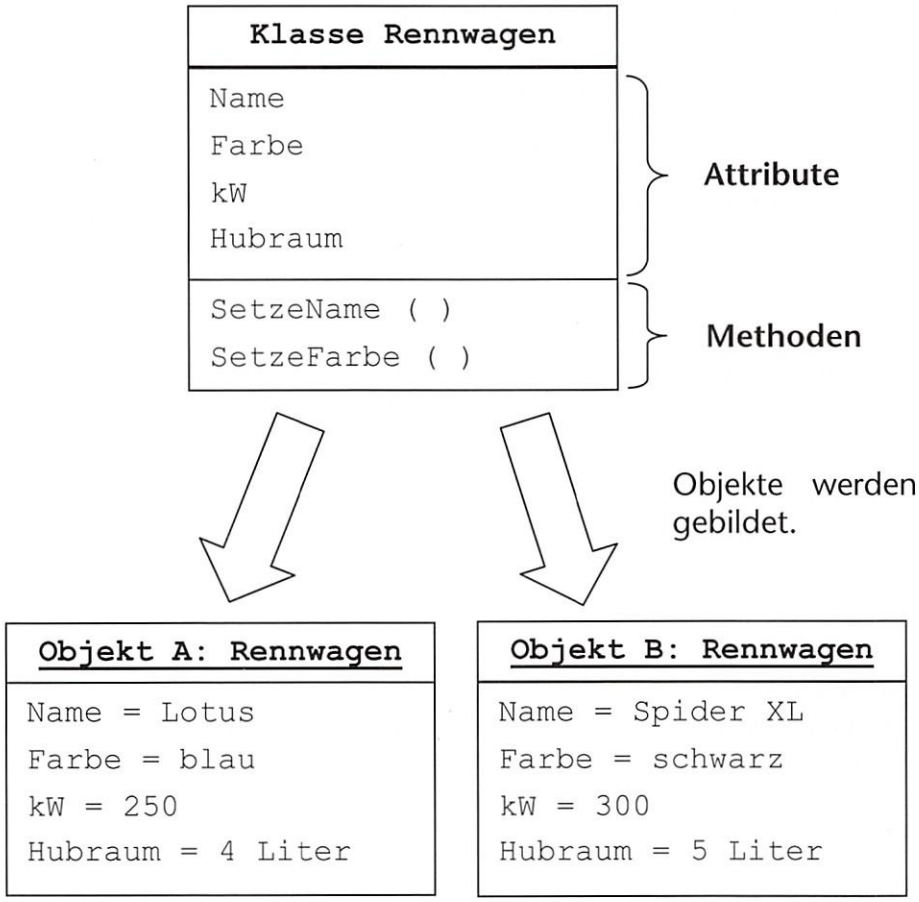


Name:	Lotus
Farbe:	blau
kW:	250
Hubraum:	4 Liter



Name:	Spider XL
Farbe:	schwarz
kW:	300
Hubraum:	5 Liter

Beide Rennwagen haben dieselben Attribute. Sie unterscheiden sich nur in den Attributwerten. Der Spider XL hat beispielsweise eine höhere Leistung als der Lotus. Man könnte sagen, dass beide Rennwagen mithilfe desselben Bauplanes hergestellt worden sind. Der zugrunde liegende Bauplan könnte als **Klasse** `Rennwagen` bezeichnet werden. Die folgende Darstellung entspricht der Form, die im UML-Klassendiagramm verwendet wird, um Klassen und Objekte darzustellen.



In einer objektorientierten Programmiersprache könnte die Klasse `Rennwagen` und die Instanzierung eines Objektes A so umgesetzt werden. Dabei wird die Umsetzung beispielhaft mithilfe von Pseudocode[1] vorgenommen.

1 Pseudocode dient dazu, einen Algorithmus (oder ein Programm) in möglichst verständlicher Sprache aufzuschreiben. Pseudocode ist deshalb auch keine Programmiersprache.

```
Definiere Klasse Rennwagen
   Geschützter Bereich:
   Attribut Name:      Typ Zeichenkette
   Attribut Farbe:     Typ Zeichenkette
   Attribut kW:        Typ Ganzzahl
   Attribut Hubraum:   Typ Gleitpunktzahl

   Öffentlicher Bereich:
   Methode SetzeName (Übergabeparameter N: Typ Zeichenkette)
      Falls N den Regeln entspricht
         Zuweisung: Name ← N
      Ende Falls
   Ende Methode

   Methode SetzeFarbe (Übergabeparameter F: Typ Zeichenkette)
      Falls F den Regeln entspricht
         Zuweisung: Farbe ← F
      Ende Falls
   Ende Methode
Ende Klasse

Bilde Objekt A vom Typ Rennwagen

A.SetzeName ("Lotus")
A.SetzeFarbe ("blau")
:
A.SetzeFarbe ("&((2j38xcxc")
```

Diese Werte sind sinnvoll und werden von der Methode akzeptiert.

Dieser Wert ist nicht sinnvoll und wird von der Methode nicht akzeptiert.

Die Methode wird mithilfe des sogenannten Punktoperators aufgerufen.

An dem Pseudocode wird deutlich, dass die Klasse `Rennwagen` als Bauplan dient, von dem dann beliebig viele Objekte gebildet werden können. Ein einzelnes Objekt bekommt über die entsprechenden Methoden die Werte für seine Attribute. Dabei prüft die Methode, ob der Wert sinnvoll ist und dem Attribut zugewiesen werden kann. Ein Objekt kann dadurch niemals unsinnige Werte erhalten. Die Sicherheit einer Software wird damit deutlich erhöht.

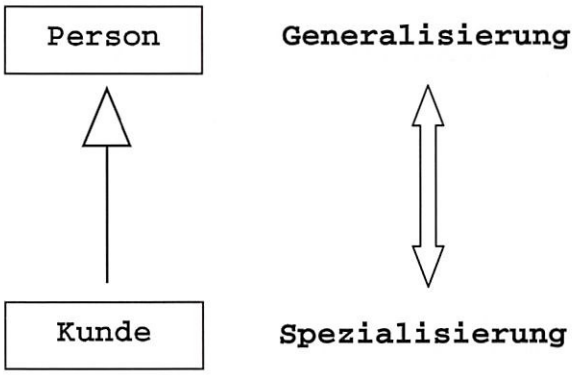
1.2.2 Generalisierung und Spezialisierung

Das Konzept der Generalisierung (bzw. Spezialisierung) ist ein zentrales Thema der objektorientierten Softwareentwicklung. Durch dieses Konzept können Situationen aus der „realen" Welt oder der gegebenen Problemstellung besser in ein objektorientiertes Modell und anschließend auch in eine objektorientierte Programmiersprache umgesetzt werden. Von Spezialisierung spricht man, wenn eine bestehende Klasse ihre Attribute und Methoden an eine neue Klasse weitergibt (vererbt). Die neu entstandene Klasse kann dadurch alle Attribute und Methoden nutzen – allerdings unter gewissen Voraussetzungen, um die Kapselung der Attribute nicht zu gefährden.

Das folgende Beispiel zeigt eine solche Generalisierung bzw. Spezialisierung. Dabei wird eine bestehende Klasse `Person` mit ihren „personentypischen" Attributen und Methoden zu einer Klasse `Kunde` spezialisiert. Der Kunde **ist** eine Person, allerdings mit zusätzlichen Eigenschaften und Methoden.

Deshalb spricht man bei dieser Beziehung zwischen den Klassen von einer **Ist-Beziehung** oder auch von einer **Vererbung**.

Beispiel:



Hinweis:

Die Klasse, die vererbt (Person), wird in der Regel **Basis-Klasse** oder **Ober-Klasse** genannt. Die Klasse, die erbt (Kunde), wird **abgeleitete Klasse** oder **Unter-Klasse** genannt.

Mithilfe von Pseudocode könnte die Person-Kunde-Spezialisierung so dargestellt werden:

```
Definiere Klasse Person
   Geschützter Bereich:
   Attribut Name:    Typ Zeichenkette
   Öffentlicher Bereich:
   Methode SetzeName (Übergabeparameter N: Typ Zeichenkette)
      :
   Ende Methode
Ende Klasse

Definiere Klasse Kunde: erbt von Klasse Person
   Geschützter Bereich:
   Attribut Kundennummer:    Typ Ganzzahl
   Öffentlicher Bereich:
   Methode SetzeNummer (Übergabeparameter N: Typ Ganzzahl)
      :
   Ende Methode
Ende Klasse

Bilde Objekt A vom Typ Kunde

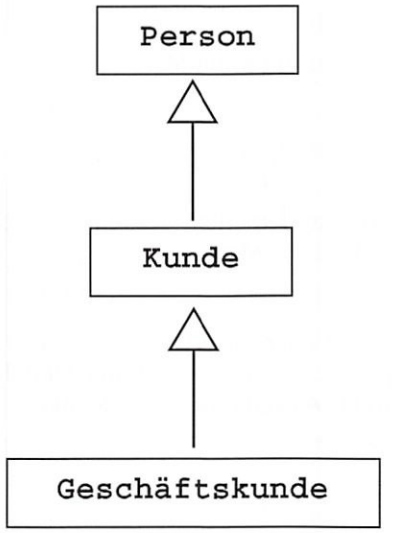
A.SetzeName ("Maier")
A.SetzeNummer (123456)
```

Diese Methode hat der Kunde „geerbt".

1.2.3 Polymorphie

Polymorphie bedeutet wörtlich *Vielgestaltigkeit*. In der objektorientierten Programmierung hat es damit zu tun, dass Methoden in mehreren Klassen den gleichen Namen haben, aber unterschiedliche Aufgaben erfüllen. Die Klassen gehören dabei zu einer Vererbungshierarchie. Das folgende Beispiel zeigt eine solche Vererbungshierarchie:

Beispiel:



Jede dieser Klassen soll nun beispielsweise über eine Methode verfügen, die die Daten (Attributwerte) des jeweiligen Objektes auf dem Bildschirm ausgibt. Eine solche Methode könnte in jeder Klasse **Ausgabe()** heißen. Weiterhin soll es möglich sein, dass beliebige Objekte der einzelnen Klassen in einem Array (Feld) gespeichert werden. Dazu wird ein Array vom Typ der Basis-Klasse `Person` angelegt und mit Objekten der anderen Klassen gefüllt. Die technische Umsetzung der Polymorphie sorgt dann dafür, dass von einem Arrayelement immer die korrekte Ausgabe-Methode aufgerufen wird. Der folgende Pseudocode soll die Problematik verdeutlichen:

```
Definiere Klasse Person
   :
   Methode Ausgabe()
      Zeige Personendaten auf Bildschirm an
   Ende Methode
Ende Klasse

Definiere Klasse Kunde: erbt von Klasse Person
   :
   Methode Ausgabe()
      Zeige Kundendaten auf Bildschirm an
   Ende Methode
Ende Klasse

Definiere Klasse Geschäftskunde: erbt von Klasse Kunde
   :
   Methode Ausgabe()
      Zeige Geschäftskundendaten auf Bildschirm an
   Ende Methode
Ende Klasse
```

```
Bilde Objekt A vom Typ Person
Bilde Objekt B vom Typ Kunde
Bilde Objekt C vom Typ Geschäftskunde

Bilde Array Z mit drei Elementen vom Typ Person

Z(1) ← A
Z(2) ← B
Z(3) ← C

Z(3).Ausgabe()
```

Den Arrayelementen werden verschiedene Objekte zugewiesen.

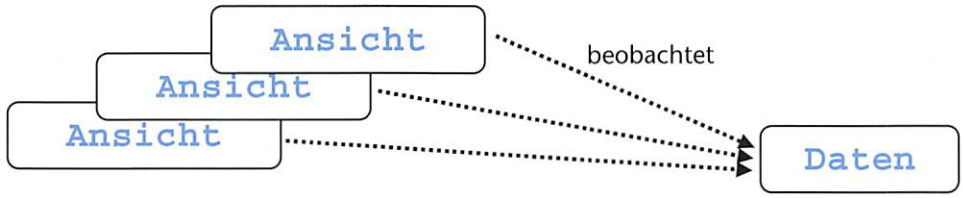
`Z(3)` ist zwar vom Typ `Person`, verweist aber auf ein Objekt vom Typ `Geschäftskunde`. Dank der Polymorphie wird trotzdem die korrekte Ausgabemethode der Klasse Geschäftskunde aufgerufen.

Hinweis:

In den objektorientierten Programmiersprachen wie Java, C++ und C# wird die Polymorphie (oder der Polymorphismus) mithilfe von Basis-Klassen-Arrays umgesetzt. Die Elemente des Arrays verweisen dabei auf beliebige Objekte aus der zugrunde liegenden Vererbungshierarchie. In C++ geschieht die Umsetzung mit Basis-Klassen-Zeigern, denen die Adressen von Objekten aus der Vererbungshierarchie zugewiesen werden. In Java und C# erfolgt die Realisierung mit Verweisen auf dynamisch erstellte Objekte der Vererbungshierarchie.

1.2.4 Entwurfsmuster

Besonders wichtige Aspekte der Softwareentwicklung sind Sicherheit und Wiederverwendbarkeit. Eine Softwaresystem ist umso sicherer, je mehr es auf einer erprobten Struktur basiert. Solche erprobten Strukturen stehen als sogenannte Entwurfsmuster zur Verfügung und helfen dem Entwickler dabei, das Softwaresystem zu implementieren. Durch die Wiederverwendbarkeit gut strukturierter Systeme kann auch die Effizienz gesteigert werden – die Software kann kostengünstiger entwickelt werden. Ein häufig verwendetes Entwurfsmuster ist das **Observer**-Muster. Bei diesem Muster gibt es eine Klasse, deren Objekte beliebige Daten repräsentieren. Beispielsweise repräsentiert ein Objekt dieser Klasse die Datensätze aller Kunden einer Firma. Diese Daten könnten nun in verschiedenen Ansichten dargestellt werden (Liste oder Tabelle). Bei einer Änderung der Daten müssen die Ansichten informiert werden, damit sie Aktualisierungen durchführen können. Dieser Informationsfluss wird mithilfe des Observer-Konzeptes realisiert. Alle Ansichten beobachten (engl. *to observe*) das Datenobjekt und reagieren dann auf Änderungen. Technisch gesehen wird dieses Beobachten dadurch realisiert, dass sich die Ansichten bei dem Datenobjekt anmelden (in eine Liste eintragen) und bei Änderungen der Daten durch einen Methodenaufruf informiert werden.



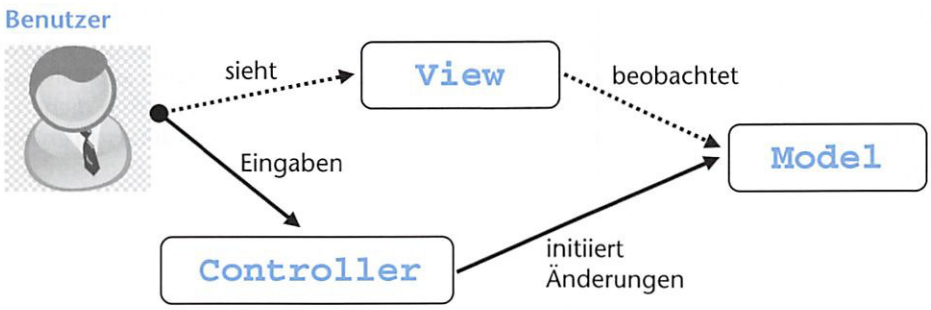
Das automatische Informieren der Ansichten ist ein großer Vorteil dieses Musters. Allerdings könnte es nachteilig sein, wenn sehr viele Ansichten das Datenobjekt beobachten. Da bei der kleinsten Änderung alle Ansichten informiert werden (auch die Ansichten, die diese Änderung möglicherweise nicht betreffen), kann es zu Performance-Problemen kommen.

1.2.5 Architekturmuster

Die oben besprochenen Entwurfsmuster dienen als Vorlage für Teilprobleme des zu entwickelnden Softwaresystems. Architekturmuster können hingegen als eine Vorlage für das ganze System betrachtet werden. Sie regeln den Aufbau der Kommunikation innerhalb des Systems und auch die Verteilung von Aufgaben an Komponenten des Softwaresystems. Beispielhaft sollen hier zwei Architekturmuster vorgestellt werden: das **Model View Controller-Konzept** (welches das Observer-Entwurfsmuster beinhaltet) und das **Schichtenmodell**.

Model View Controller (MVC)

Dieses Konzept trennt ein System in eine Modell-Klasse, eine Ansichts-Klasse (oder mehrere Ansichts-Klassen) und eine Steuerungs-Klasse. Die Modell-Klasse repräsentiert die Daten des Systems, die mithilfe der Ansichts-Klassen in verschiedenen Sichtweisen dargestellt werden können. Angenommen, es handelt sich bei den Daten um Messwerte, so könnten diese Werte beispielsweise in einer Tabellenansicht, aber auch in einer Grafik in einem Koordinatensystem angezeigt werden. Der Vorteil dieser Trennung ist die Unabhängigkeit von Datenhaltung und Sicht. Das Hinzufügen weiterer Ansichten ist unabhängig von der Modell-Klasse. Die Steuerungs-Klasse ist das Bindeglied zwischen Modell und Ansicht. Sie registriert Benutzereingaben, die über die Ansicht kommen, und kommuniziert dann mit dem Modell, um beispielsweise weitere Daten zu erhalten oder Änderungen der Daten zu initiieren.



Das Schichtenmodell

Das MVC-Konzept liefert einen Entwurf für ein Softwaresystem, welches auch ein einzelnes Programm sein kann. Das Schichtenmodell geht noch einen Schritt weiter und liefert einen Entwurf für ein Softwaresystem, das über mehrere Komponenten (Schichten, engl. *tier*) verfügt, die auch physikalisch getrennt sein können und über ein Netzwerkprotokoll miteinander kommunizieren. Häufig genutzt werden das Zwei-Schichten-Modell (engl. *two tier*) und das Drei-Schichten-Modell (engl. *three tier*). Das Zwei-Schichten-Modell ist oftmals das klassische Client-Server-Modell. In der Regel ist der Client ein Programm, das eine Benutzeroberfläche und die zugehörige Geschäftslogik anbietet. Die benötigten Daten fragt der Client dann vom Server ab. Das Drei-Schichten-Modell koppelt hingegen die Geschäftlogik von der Benutzoberfläche ab – damit wird eine weitere Schicht angelegt. In der Web-Programmierung wird ein Drei-Schichten-Modell sehr oft durch einen Client in Form eines Browsers und durch die Anzeige mit HTML, XHTML oder XML umgesetzt. Die Geschäftslogik liegt dann auf einem Webserver und wird mit einer Skriptsprache wie PHP oder Perl implementiert. Die Daten werden von einem zusätzlichen Server mithilfe geeigneter Schnittstellen wie ODBC ausgelesen. Dieses System hat den großen Vorteil, dass Änderungen an einer Schicht (Ansicht, Logik oder Datenhaltung) die anderen Schichten nicht tangieren. Wartbarkeit und Erweiterbarkeit eines solchen Systems sind damit sehr gut.

Drei-Schichten-Modell:



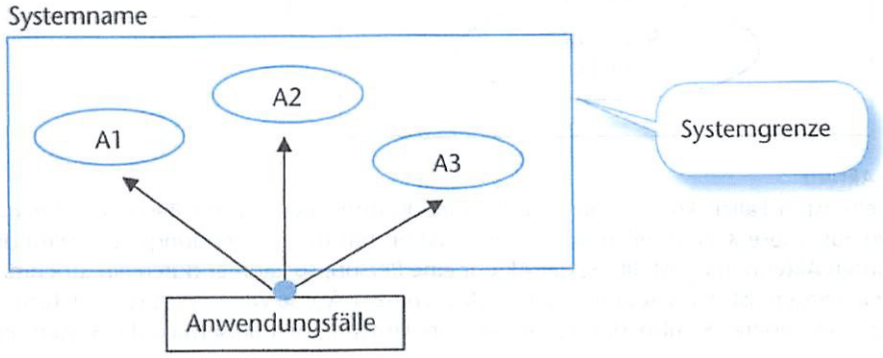
2 Das Anwendungsfall-diagramm

Anwendungsfalldiagramme (Use-Case-Diagramme) beschreiben die Funktionalitäten eines Systems. Sie zeigen die möglichen Anwendungsfälle (einzelne Funktionalitäten, Geschäftsprozesse) und die Beziehungen zwischen diesen Fällen und den beteiligten Akteuren (Personen, Maschinen). Sie werden in der Regel in einer frühen Phase der Entwicklung, bei der Formulierung der Anforderungen, eingesetzt und dann in den späteren Phasen weiter spezifiziert. Sie dienen auch als Grundlage für die Identifizierung der beteiligten Klassen an dem Softwaresystem. Weiterhin dienen Anwendungsfalldiagramme auch als Grundlage für die Erstellung von Testplänen für das zu entwickelnde Softwaresystem.

2.1 Anwendungsfälle und Akteure

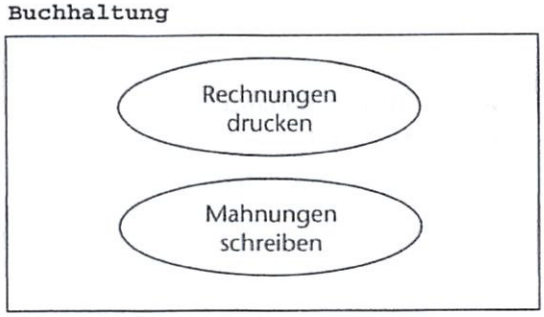
2.1.1 Anwendungsfall

Ein Anwendungsfall (engl. *use case*) beschreibt eine Funktionalität eines Systems, die durch eine bestimmte Anzahl von Aktionen durchgeführt werden kann. Es wird nur beschrieben, welche Funktionalität bereitgestellt wird. Es wird nicht beschrieben, in welcher Form das System die Funktion realisiert. Anwendungsfälle werden in Ellipsen dargestellt. Ein Anwendungsfall gehört zu einem System und wird deshalb innerhalb der Systemgrenzen dargestellt.



Beispiel:

In einer Firma werden Rechnungen für Kunden gedruckt und Mahnungen geschrieben, wenn offene Rechnungen bis zu einem bestimmten Datum nicht bezahlt sind. Das System ist die Firma bzw. die Buchhaltungsabteilung. Die Anwendungsfälle sind *Rechnungen drucken* und *Mahnungen schreiben*.



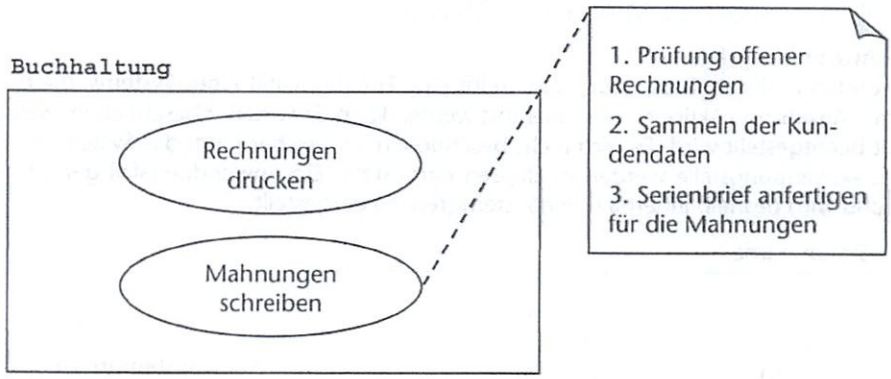
Anwendungsfälle beschreiben

Die UML sieht es nicht zwingend vor, aber es ist durchaus sinnvoll, die einzelnen Anwendungsfälle näher zu beschreiben. Das vereinfacht auch die weitere Entwicklung des Softwaresystems. Die einzelnen Anwendungsfälle können beispielsweise durch zusätzliche Anmerkungen oder durch einen angehängten UML-Kommentar beschrieben werden. Ein solcher Kommentar kann in jedem UML-Diagramm eingesetzt werden. Der oben beschriebene Anwendungsfall *Mahnungen schreiben* wird nun beispielhaft durch eine Beschreibung und alternativ durch einen Kommentar dargestellt:

Beschreibung Anwendungsfall: *Mahnungen schreiben*

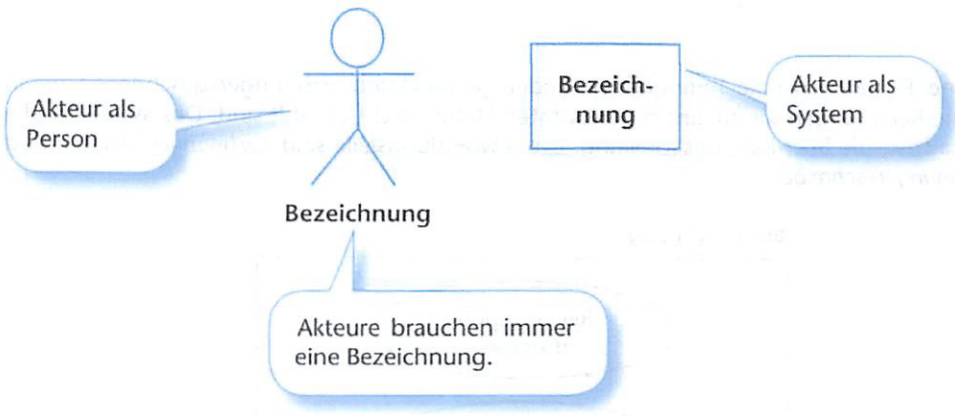
- **Ziel**: Kunden, die ihre Rechnungen nicht bezahlt haben, sollen eine Mahnung erhalten.
- **Beteiligte Akteure**: der zuständige Sachbearbeiter
- **Vorbedingungen**: Prüfung der offenen Rechnungen
- **Nachbedingungen**: Kontrolle der geschriebenen Mahnung
- **Ablauf**: 1. Suche in der Datenbank nach offenen Rechnungen, 2. Sammeln der Kundenadressen, 3. Anlegen eines Serienbriefes mit dem Mahnschreiben

Darstellung mit einem UML-Kommentar



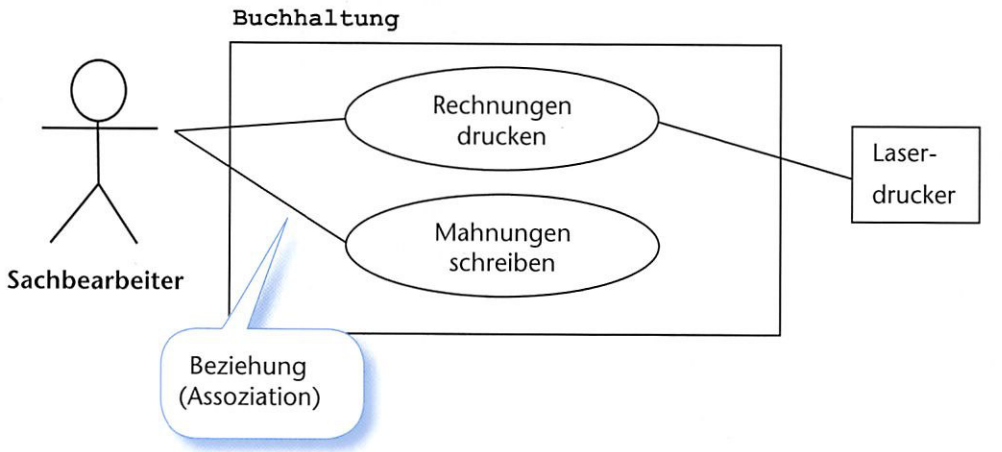
2.1.2 Akteur

In den seltensten Fällen kommt ein System ohne Kommunikation mit Benutzern oder anderen Systemen aus. Diese Kommunikation oder Interaktion mit den Anwendungsfällen wird durch die sogenannten Akteure dargestellt. Ist der Akteur eine Person, so kann er durch ein Strichmännchen dargestellt werden. Ist der Akteur ein System (Maschine o. Ä.), so wird das durch ein Rechteck oder ein geeignetes anderes Symbol dargestellt. Akteure liegen immer außerhalb der Systemgrenzen.



Beispiel:

Die Rechnungen werden in der Buchhaltungsabteilung vom Sachbearbeiter auf einem Laserdrucker ausgedruckt. Er schreibt ebenfalls die Mahnungen.



Der Akteur steht mit dem Anwendungsfall in Beziehung (Assoziation), was durch eine einfache Linie symbolisiert wird. Diese Beziehung kann einerseits bedeuten, dass der Akteur den Anwendungsfall initiiert oder zumindest daran beteiligt ist (aktiv oder passiv). Die genaue Beziehung zwischen Akteur und Anwendungsfall wird durch die Beschreibung (oder den Kommentar) näher spezifiziert. Zusätzlich bietet die UML auch die Möglichkeit, die Beziehung im Diagramm näher zu beschreiben, wie in dem folgenden Unterkapitel dargestellt wird.

2.2 Beziehungen zwischen Anwendungsfall und Akteur

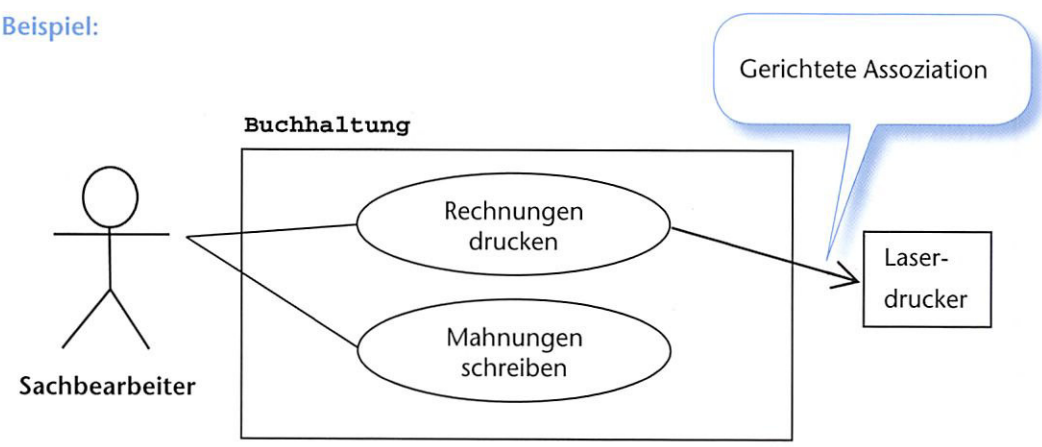
2.2.1 Die ungerichtete Assoziation

Diese Beziehung wurde bereits in dem obigen Beispiel zwischen dem Akteur *Sachbearbeiter* und dem Anwendungsfall *Mahnungen schreiben* (bzw. *Rechnungen drucken*) verwendet. Sie wird durch eine einfache Linie dargestellt. Sie ist die allgemeine Form der Beziehung und lässt Freiraum bei der Interpretation. Die Kommunikation zwischen Anwendungsfall und Akteur kann in beide Richtungen erfolgen.

2.2.2 Die gerichtete Assoziation

Diese Assoziation gibt die Richtung der Kommunikation bzw. der Ausführung vor. Sie wird durch einen Pfeil am Ende der Linie gekennzeichnet. Der Pfeil zeigt dadurch die Richtung an.

Beispiel:

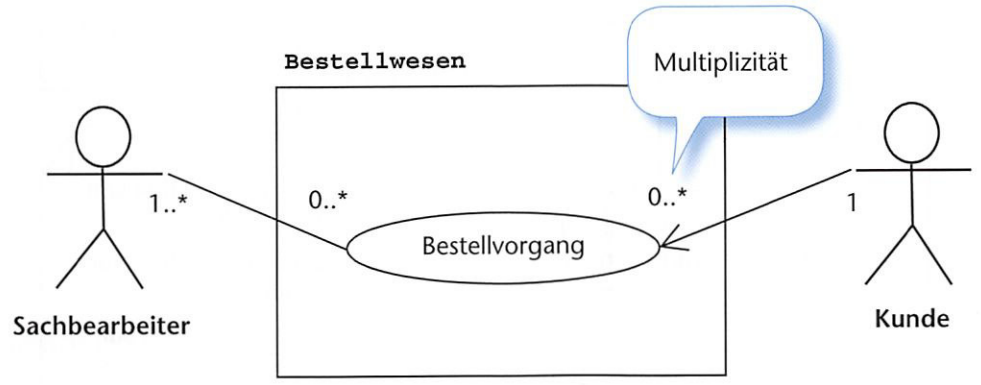


Die Kommunikation zwischen Anwendungsfall *Rechnungen drucken* und Laserdrucker geht nur in eine Richtung (der Drucker erhält die Druckdaten) – deshalb kann diese Assoziation gerichtet sein.

2.2.3 Multiplizität der Assoziation

Die einfache Assoziation sagt nichts darüber aus, ob ein Akteur mit einem oder mehreren gleichen Anwendungsfällen in Beziehung steht. Ebenso ist es unklar, ob ein Anwendungsfall mit genau einem Akteur oder verschiedenen, gleichartigen Akteuren kommuniziert. Aus diesem Grund ist es möglich, die Beziehung durch sogenannte Multiplizitäten weiter zu qualifizieren. Die Multiplizitäten geben an, wie viele Akteure mit wie vielen Anwendungsfällen in Beziehung stehen. Das folgende Beispiel zeigt die Beziehungen zwischen Sachbearbeitern, Bestellungen und Kunden.

Beispiel:



Die Multiplizitäten sind so zu lesen:

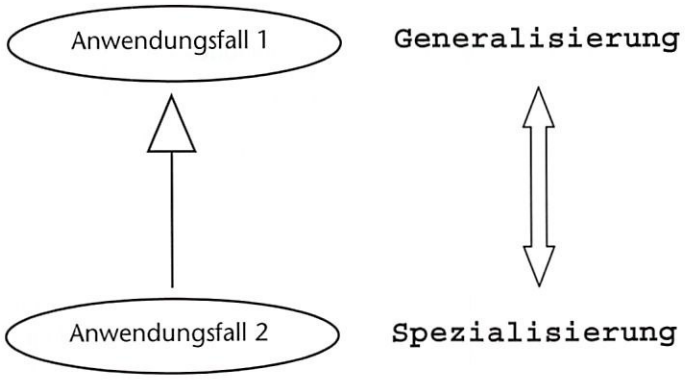
- Ein Sachbearbeiter kann keine bis beliebig viele Bestellvorgänge bearbeiten: (0..*)
- Ein Bestellvorgang wird von einem oder mehreren Sachbearbeiten bearbeitet: (1..*)
- Ein Kunde kann keine bis beliebig viele Bestellvorgänge initiieren: (0..*)
- Ein Bestellvorgang ist genau einem Kunden zugeordnet: (1)

Die Multiplizitäten werden noch einmal ausführlich bei dem Thema Klassendiagramm beleuchtet. Neben der Qualifizierung der Beziehung zwischen Akteur und Anwendungsfall dienen die Multiplizitäten auch als Hinweis für eine spätere Umsetzung der beteiligten Klassen des Softwaresystems und deren Beziehungen untereinander.

2.3 Beziehungen zwischen Anwendungsfällen

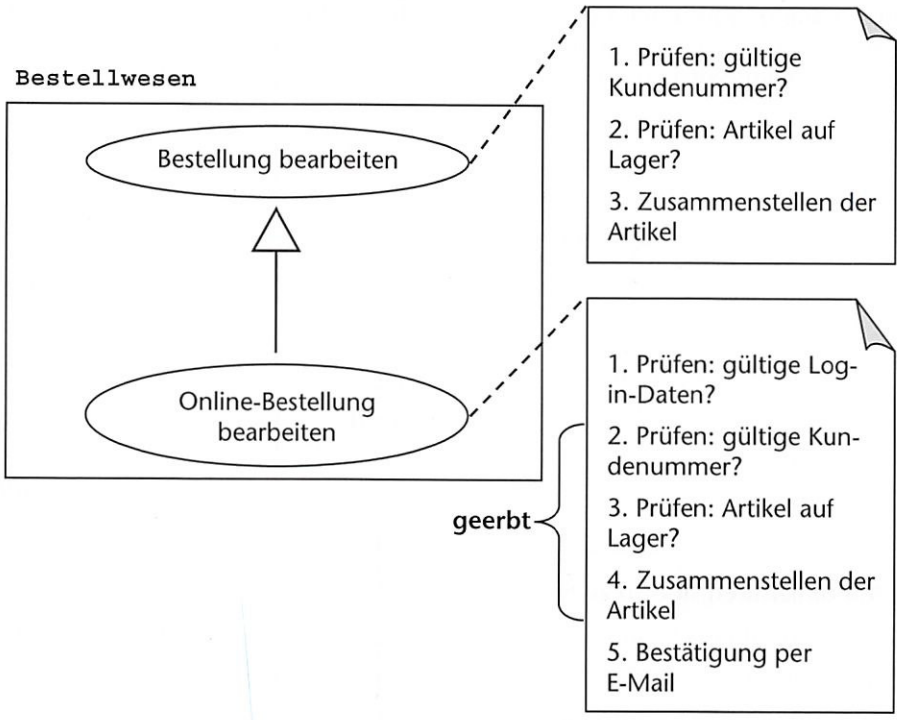
2.3.1 Generalisierung und Spezialisierung

Manchmal ist es sinnvoll, einen Anwendungsfall weiter zu verfeinern. Das bedeutet, dass der Anwendungsfall weitere Aktionen enthält und sich damit spezialisiert. Umgangssprachlich könnte man auch von einer Vererbungsbeziehung sprechen – ein Anwendungsfall vererbt einem anderen seine Funktionalität. Der erbende Anwendungsfall ergänzt die geerbte Funktionalität um weitere (spezielle) Funktionen. Diese Beziehung wird mit einem Pfeil in Richtung des generalisierten Anwendungsfalls beschrieben.



Beispiel:

Der Anwendungsfall *Bestellung bearbeiten* wird durch den Anwendungsfall *Online-Bestellung bearbeiten* spezialisiert.

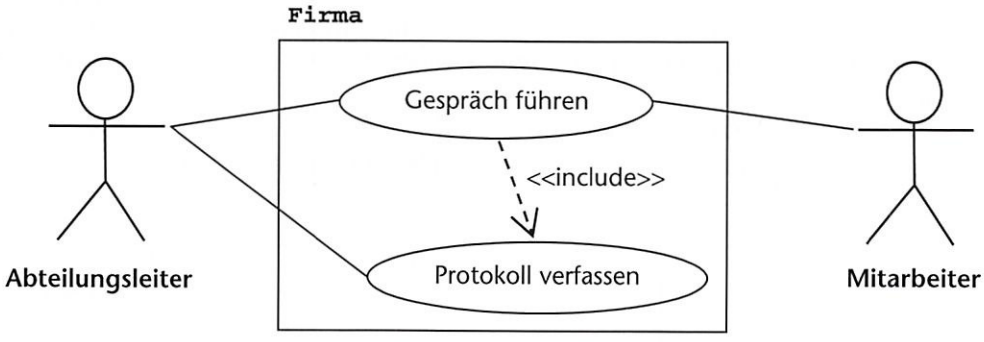


2.3.2 Die include-Beziehung

In manchen Fällen kann es sinnvoll sein, dass ein Anwendungsfall einen anderen zwingend einschließt. Tritt also ein bestimmter Anwendungsfall ein, so wird damit automatisch ein weiterer aufgerufen. Ein solches Einbinden wird durch die sogenannte **include**-Beziehung dargestellt. Diese Beziehung wird durch eine gestrichelte Linie mit einem Endpfeil auf den einzubindenden Anwendungsfall und die Bezeichung **<<include>>** symbolisiert.

Beispiel:

Der Anwendungsfall *Gespräch führen* hat zur unbedingten Folge, dass ein Protokoll verfasst wird. Der Anwendungsfall *Protokoll verfassen* ist in den Anwendungsfall *Gespräch führen* eingebunden (inkludiert).



Hinweis:

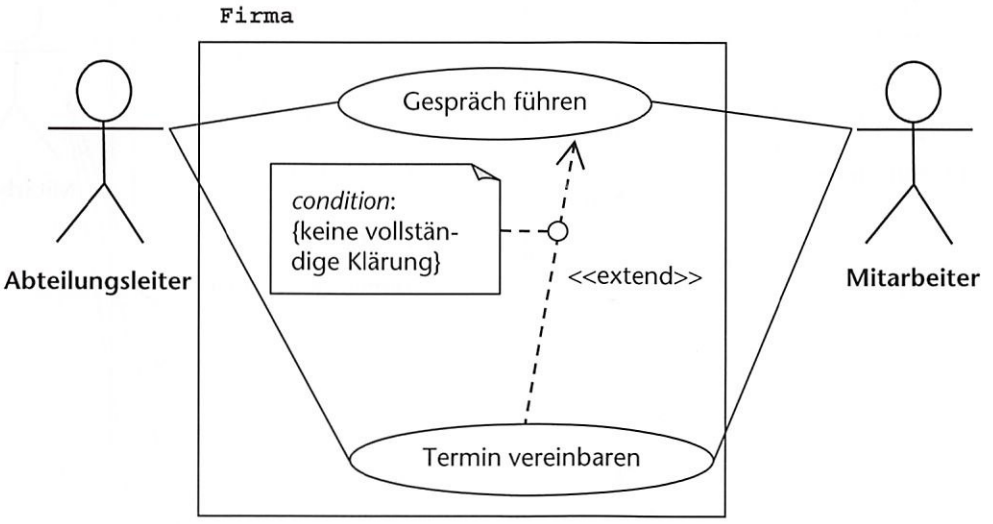
Der eingebundene Anwendungsfall ist kein vollständiger Anwendungsfall, der auch alleine aufgerufen werden kann. Er muss von einem anderen Anwendungsfall eingebunden werden (also bestimmte Vorbedingungen müssen erfüllt sein). Allerdings kann er natürlich von beliebig vielen anderen Anwendungsfällen eingebunden werden. Der Anwendungsfall *Protokoll verfassen* aus dem obigen Beispiel könnte auch bei vielen anderen Gelegenheiten genutzt werden (Meeting etc.).

2.3.3 Die extend-Beziehung

Im Gegensatz zur unbedingten (automatischen) Einbindung eines weiteren Anwendungsfalls bei einer **include**-Beziehung bindet die **extend**-Beziehung einen weiteren Anwendungsfall nur unter einer bestimmten Voraussetzung (Bedingung) ein – oder anders gesagt: Die **extend**-Beziehung erweitert den Anwendungsfall nur unter einer bestimmten Bedingung. Diese Bedingung (engl. *condition*) muss zusätzlich angegeben werden. Der erweiterte Anwendungsfall ist selbstständig und kann auch alleine aufgerufen werden (im Gegensatz zum inkludierten Anwendungsfall). Diese Beziehung wird durch eine gestrichelte Linie mit einem Endpfeil auf den zu erweiternden Anwendungsfall und die Bezeichung **<<extend>>** symbolisiert. Zusätzlich wird die Bedingung angegeben.

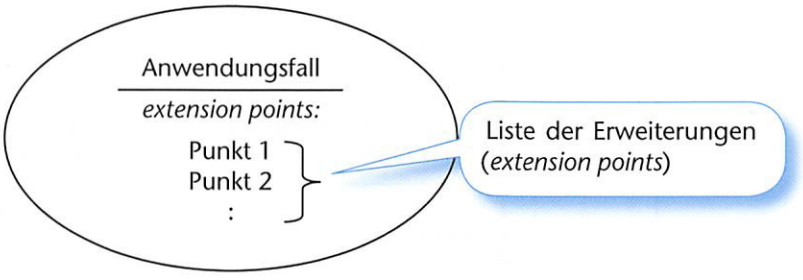
Beispiel:

Der Anwendungsfall *Gespräch führen* kann durch einen Anwendungsfall *Termin vereinbaren* erweitert werden, wenn die Bedingung erfüllt ist und keine vollständige Klärung der Angelegenheit stattgefunden hat.



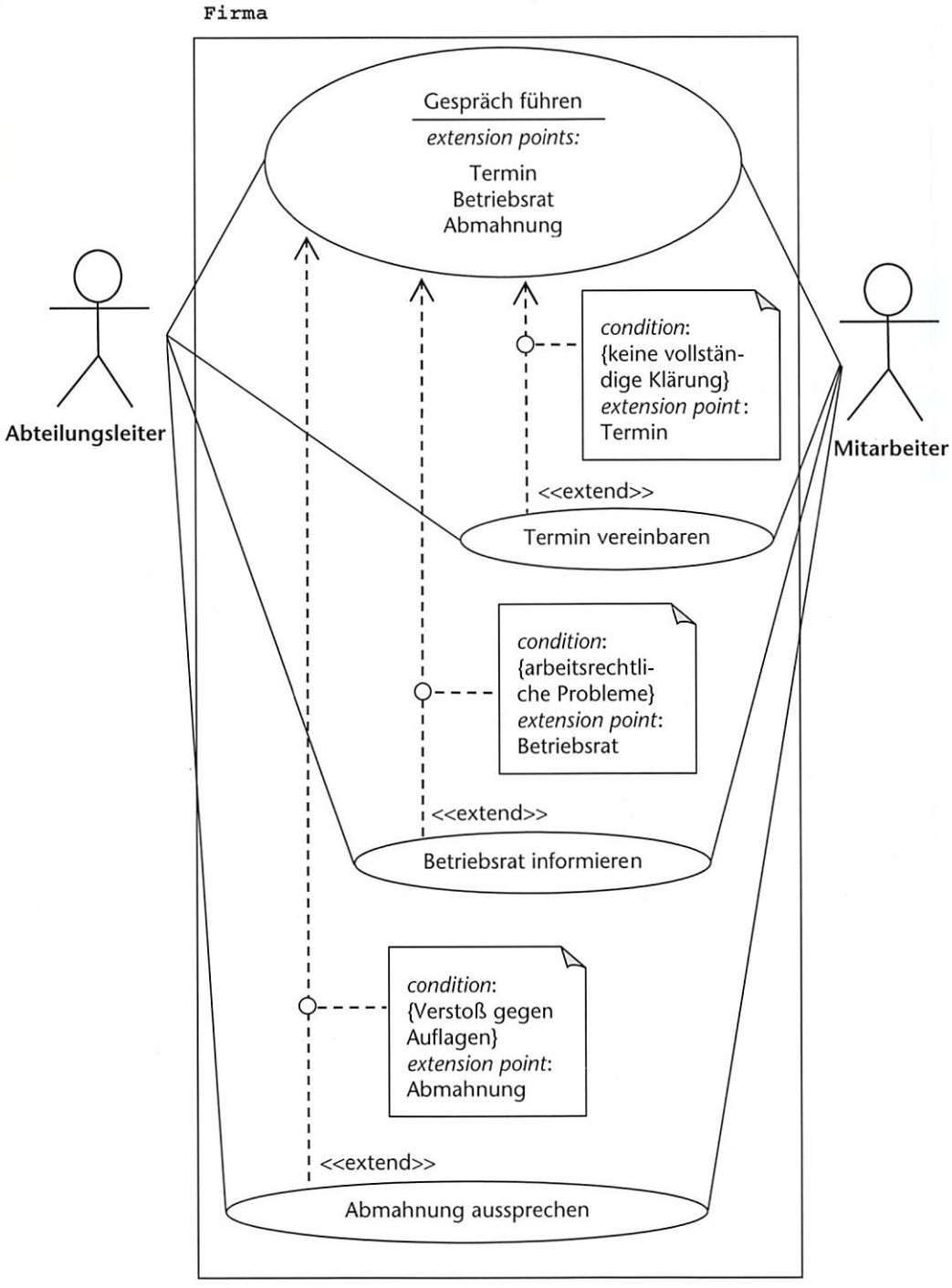
Extension points

In dem obigen Beispiel wurde der Anwendungsfall *Gespräch führen* mit einem Anwendungsfall erweitert. Es ist durchaus möglich, dass zusätzliche Erweiterungen sinnvoll sind. In solchen Fällen erhält der Anwendungsfall eine Liste der zusätzlichen Erweiterungen (engl. *extension points*), die dann auch unterhalb der einzelnen Bedingungen angegeben werden.



Beispiel:

Der Anwendungsfall *Gespräch führen* kann nicht nur durch den Anwendungsfall *Termin vereinbaren*, sondern auch durch die Anwendungsfälle *Betriebsrat informieren* und *Abmahnung aussprechen* erweitert werden.



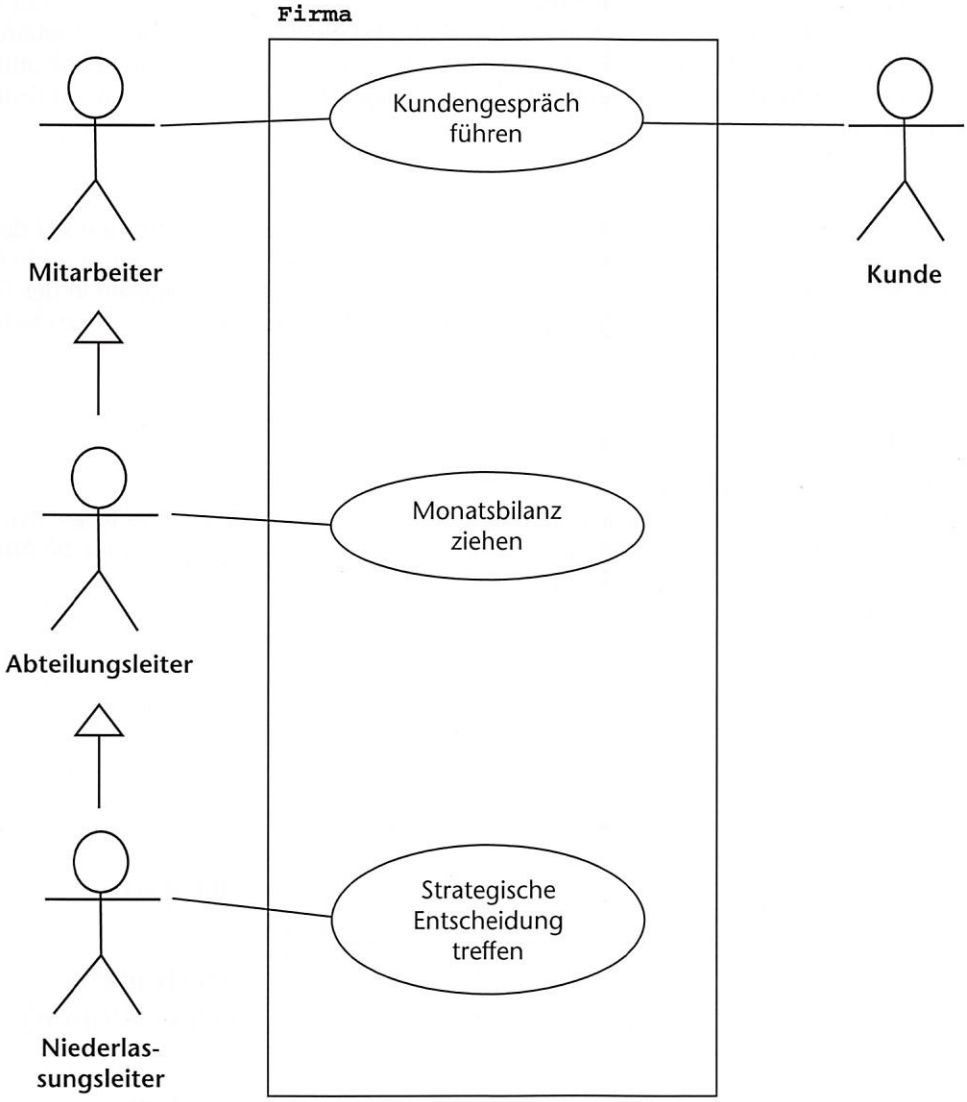
2.4 Beziehungen zwischen Akteuren

2.4.1 Generalisierung und Spezialisierung

Die Generalisierung bzw. Spezialisierung bei Akteuren erfolgt analog zu den Anwendungsfällen. Ein Akteur spezialisiert sich zu einem weiteren Akteur. Dabei ist ganz wichtig, dass der spezialisierte Akteur automatisch an allen Anwendungsfällen des generalisierten Akteurs beteiligt ist. Das bedeutet mit anderen Worten, dass der spezialisierte Akteur die Beteiligungen an den Anwendungsfällen einfach *erbt*. Die Beziehung wird wie gewohnt durch einen Pfeil in Richtung des generalisierten Akteurs beschrieben.

Beispiel:

Der Mitarbeiter einer Firma wird zu einem Abteilungsleiter spezialisiert, der Abteilungsleiter wiederum zu einem Niederlassungsleiter. Der Mitarbeiter führt Kundengespräche. Die beiden spezialisierten Akteure (Abteilungsleiter und Niederlassungsleiter) führen selbstverständlich auch Kundengespräche – sie *erben* diese Beziehung von dem Mitarbeiter. Zusätzlich sind sie an eigenen Anwendungsfällen beteiligt, an denen der Mitarbeiter nicht beteiligt ist.



3 Das Klassendiagramm

Das Klassendiagramm ist das wichtigste Diagramm für die Analyse und das Design von Softwaresystemen. Im Klassendiagramm werden die Klassen des zu entwickelnden Systems und deren Beziehungen untereinander dargestellt. Das Klassendiagramm ist statisch, es zeigt keinen zeitabhängigen Ablauf von Operationen. Wer bereits Erfahrungen mit der Entwicklung einer Datenbank und dem **Entity-Relationship-Modell** (ERM) gesammelt hat, wird einige Analogien zwischen Klassendiagramm und ERM erkennen.

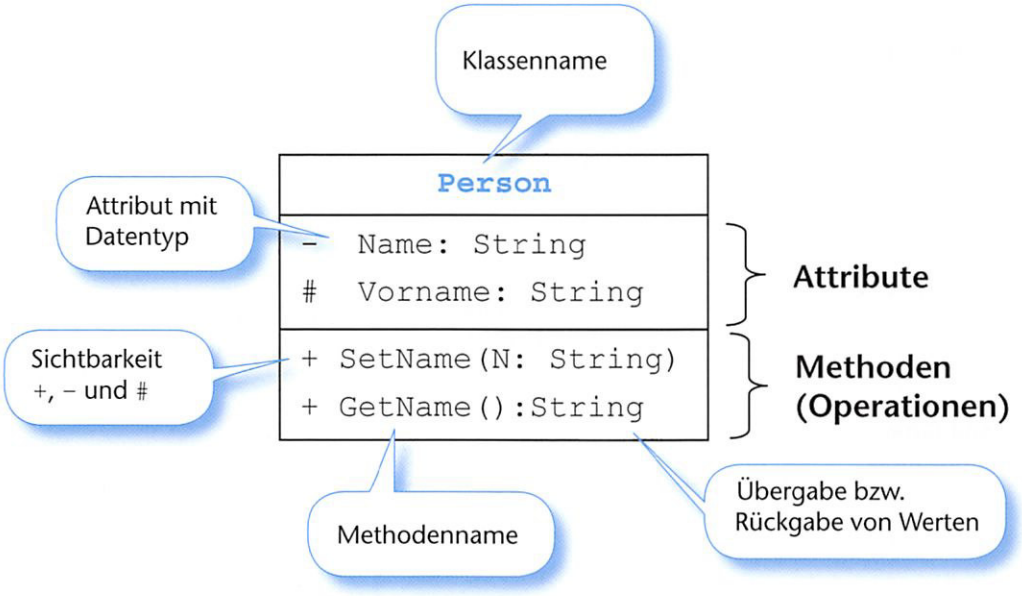
Hinweis:

Das Entity-Relationship-Modell (ERM) ist ein Hilfsmittel zur Datenmodellierung bei der Entwicklung von relationalen Datenbanken. Ausgehend von dem ERM kann dann sehr einfach das Datenmodell in der Datenbank implementiert werden – also die Tabellen in der Datenbank angelegt und die Beziehungen zwischen den Tabellen durch Primär- und Fremdschlüssel modelliert werden.

3.1 Die Darstellung der Klasse

3.1.1 Grundlegender Aufbau

Die Darstellung einer Klasse im Klassendiagramm besteht im Wesentlichen aus drei Teilen: dem Klassennamen, den Attributen und den Methoden. Die weiteren Details der Symbolik für die Attribute und Methoden werden in den nachfolgenden Unterkapiteln behandelt.



Hinweis:

In der UML-Fachsprache spricht man bei **Methoden** auch von **Operationen**.

3.1.2 Beschreibung der Attribute

Die Attribute einer Klasse können sehr unterschiedlich sein. Sie können privat oder öffentlich sein. Sie können mit einem Wert initialisiert werden oder nicht. Sie können ein Instanz- oder Klassenattribut sein. Das Klassendiagramm bietet für alle diese Fälle die entsprechende Symbolik bzw. Bezeichnung.

Sichtbarkeit:

Symbol	Beschreibung
-	privates (geschütztes) Attribut (engl. *private*)
+	öffentliches Attribut (engl. *public*)
#	geschütztes Attribut – spielt bei der Vererbung eine Rolle (engl. *protected*)

Beispiele:

- `- Name`
- `+ Gehalt`
- `# Kontonummer`

Ein privates Attribut ist gegen den Zugriff von außen geschützt (Kapselung). Es kann nur über entsprechende Methoden angesprochen werden. Ein öffentliches Attribut kann direkt angesprochen werden – es widerspricht deshalb auch dem Grundprinzip der objektorientierten Programmierung und wird deshalb nur in ganz seltenen Fällen angewendet. Das geschützte Attribut vom Typ `protected` (`#`) verhält sich nach außen wie ein privates Attribut. Innerhalb einer Vererbungshierarchie verhält es sich allerdings wie ein öffentliches Attribut. Das bedeutet, dass eine Klasse, die von einer anderen erbt, auf das `protected`-Attribut genauso zugreifen kann, als wäre es ein eigenes Attribut.

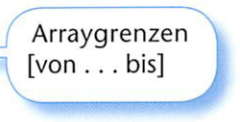
Datentyp und Multiplizität:

Möchte man das Attribut spezifizieren, so ist der Datentyp nach einem Doppelpunkt anzugeben. Es gibt keine Vorgaben für die Verwendung von Datentypnamen. Am sinnvollsten ist es jedoch, allgemein gültige Bezeichnungen für Datentypen zu verwenden oder sich an die Bezeichnungen der Implementierungssprache zu halten.

Die Multiplizität gibt an, wie viele Exemplare von diesem Attribut angelegt werden sollen. Das entspricht nichts anderem als einem Array in einer Programmiersprache wie C++, C# oder Java.

Beispiele:

- `- Werte:` `Integer [1..10]`
- `- Mittelwert:` `Double`



Initialisierung eines Attributes:

Soll ein Attribut initialisiert werden, so kann das mit der Zuweisung eines Wertes kenntlich gemacht werden.

Beispiele:

- `- Provision:` `Double = 0.5`
- `- Qualität:` `String = "Standard"`

Besondere Eigenschaften:

Sollen Attribute über weitere besondere Eigenschaften verfügen, so wird die Eigenschaft in geschweiften Klammern dem Attribut nachgestellt.

Eigenschaft	Beschreibung
{readonly}	Das Attribut darf nach der Initialisierung nur noch gelesen werden.
{ordered}	Die Werte eines Attributes müssen geordnet sein. Ein Wert darf nur einmal vorkommen.
{bag}	Die Werte eines Attributes müssen nicht geordnet sein. Ein Wert darf mehrmals vorkommen.
{sequence}	Die Werte eines Attributes müssen geordnet sein. Ein Wert darf aber mehrmals vorkommen.

Beispiele:

- `- Gehalt: Double = 1850.50 {readonly}`
- `- Werte: Integer [1..3] = { 1 , 2 , 3 } {ordered}`

Klassen- oder Instanzattribut:
Die Unterscheidung eines Klassen- oder Instanzattributes erfolgt über die Unterstreichung. Ein unterstrichenes Attribut ist ein Klassenattribut.

Beispiel:

- `- Gehalt: Double = 1850.50 {readonly}`
- `- Objektzähler: Integer = 0`

Ein Klassenattribut wird im Gegensatz zu einem Instanzattribut nur einmal im Speicher angelegt. Jedes Objekt (Instanz) der Klasse kann auf das Klassenattribut zugreifen. Ein Instanzattribut wird hingegen für jedes Objekt neu angelegt. Die Objekte können auch nur auf ihre eigenen Attribute zugreifen. Ein klassisches Beispiel für ein Klassenattribut ist ein Objektzähler (Instanzzähler). Bei jeder Erstellung eines Objektes wird dieser Zähler automatisch inkrementiert – damit kann jederzeit die Anzahl der instanzierten Objekte abgefragt werden.

3.1.3 Beschreibung der Methoden

Bei Methoden ist es vor allem wichtig, ob sie von außen aufrufbar sind und welche Werte sie übernehmen bzw. zurückgeben können.

Sichtbarkeit:

Symbol	Beschreibung
-	private Methode (von außen nicht aufrufbar)
+	öffentliche Methode (von außen aufrufbar, Schnittstelle nach außen)
#	geschützte Methode – spielt bei der Vererbung eine Rolle

Beispiele:

- `- Berechnung ()`
- `+ Initialisierung()`
- `# InterneMethode()`

Eine private Methode kann nur von den Methoden der eigenen Klasse verwendet werden. Private Methoden dienen deshalb oft als Hilfsmethoden, die zwar innerhalb der Klasse sinnvoll genutzt werden, aber von außen nicht sinnvoll zu verwenden sind. Ein Beispiel dazu wäre eine Methode, die Attributwerte vertauscht und damit als Hilfsmethode für eine öffentliche Sortiermethode dient. Ähnlich verhält es sich auch mit einer `protected`-Methode. Innerhalb einer Vererbungshierarchie kann eine `protected`-Methode so benutzt werden, als wäre sie in der eigenen Klasse als privat deklariert worden. Eine private Methode wäre innerhalb einer Vererbungshierarchie nicht aufrufbar.

Die öffentlichen Methoden können hingegen als die Schnittstellen der Klasse nach außen beschrieben werden. Mithilfe dieser Methoden kann mit einem Objekt der Klasse kommuniziert werden.

Übergabeparameter:
Methoden können Werte übernehmen und Werte zurückgeben. Bei der Übergabe kann in einen sogenannten Wertaufruf (*call by value*) und einen Referenzaufruf *(call by reference)* unterschieden werden. Bei dem Wertaufruf wird nur ein Wert in den Übergabeparameter kopiert. Beim Referenzaufruf wird eine Referenz auf eine Variable übergeben – damit sind Änderungen der Variablen innerhalb der Methode nachhaltig, denn sie beziehen sich auf die (per Referenz) übergebene Variable. Um die Übergaben zu spezifizieren, werden die Schlüsselworte `in`, `out` und `inout` benutzt:

Modus	Beschreibung
in	Der Parameter darf nur gelesen werden (*call by value*).
out	Der Parameter darf nur geschrieben werden (spezieller *call by reference* bzw. mit einem Rückgabewert vergleichbar).
inout	Der Parameter darf gelesen und geschrieben werden (*call by reference*).

Beispiele:

- `+ Methode_CBV(in X : int)`
  Diese Methode übernimmt einen Wert mit einem *call by value*. In dem Parameter `X` wird der per Kopie übergebene Wert gespeichert.
- `+ Methode_CBR_1(out X : int)`
  Diese Methode übernimmt einen Wert mit einem speziellen *call by reference*. In der Methode wird dem Parameter `X` ein Wert zugewiesen, der dann auch für die per Referenz übergebene Variable gültig ist.
- `+ Methode_CBR_2(inout X : int)`
  Diese Methode übernimmt einen Wert mit einem *call by reference*. In der Methode kann der Parameter `X` gelesen und geschrieben werden. Das Schreiben hat dann selbstverständlich eine Änderung der per Referenz übergebenen Variable zur Folge.

Hinweis:

Wird kein Modus angegeben, so wird der Modus **`in`** vorausgesetzt.

Rückgabewert:

Der Rückgabewert einer Methode wird durch die Angabe des Rückgabedatentyps nach einem Doppelpunkt kenntlich gemacht.

Beispiele:

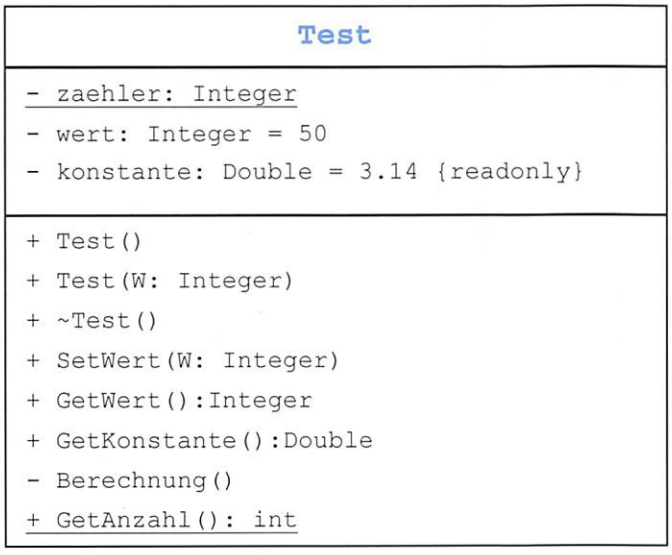
- `- Berechnung ():  Double`
- `+ GetName():  String`

Diese Berechnungsmethode gibt einen Wert vom Datentyp `Double` zurück.

3.1.4 Umsetzung eines Klassendiagramms in eine objektorientierte Programmiersprache

Nach den eher theoretischen Beschreibungen der Notation einer Klasse in UML soll nun ein konkretes Beispiel in die Programmiersprache C# umgesetzt werden.

Beispiel einer Klasse im UML-Klassendiagramm:



Die Klasse **`Test`** verfügt über drei Attribute, von denen eines ein Klassenattribut ist. Ein weiteres Attribut darf nur gelesen werden.

Neben den Standard-Get- und Set-Methoden gibt es einen Standard- sowie einen Parameterkonstruktor, der einen Wert übernimmt. Zusätzlich führt eine private Methode eine Berechnung durch.

Umsetzung der Klasse in die Sprache C#

```
using System;

namespace UML_IT_BERUFE
{

   class CTest
   {
      private static int zaehler = 0;
      int wert = 50;
      const double konstante = 3.14;

      public CTest()
      {
        zaehler++;
      }

      public CTest(int W)
      {
        SetWert(W);
        zaehler++;
      }

      public void SetWert(int W)
      {
        if ( W > 0 && W < 9999 )
        {
          wert = W;
          Berechnung();
        }
      }

      public int GetWert()
      {
        return wert;
      }

      public double GetKonstante()
      {
        return konstante;
      }

      public static int GetAnzahl()
      {
        return zaehler;
      }
```

Alternativ könnten die Get- und Set-Methoden auch durch die Eigenschaften in C# ersetzt werden.

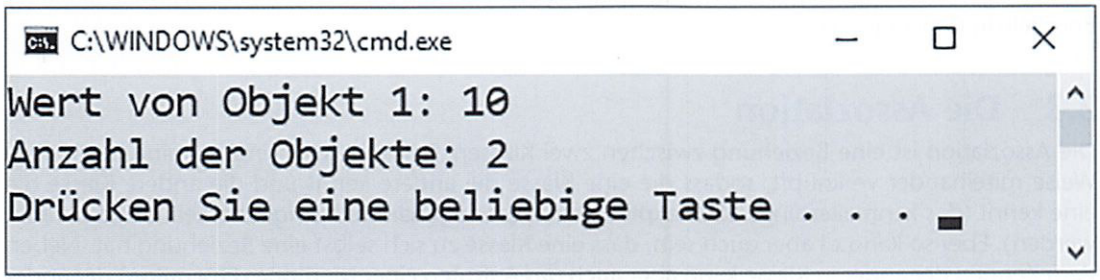
Beispiel:

```
public int WERT
{
 set
 {
   if (value > 0
            &&
       value < 9999)
   {
     wert = value;
     Berechnung();
   }
  }
  get
  {
   return wert;
  }
}
```

```
    private void Berechnung()
    {
      wert = 2 * wert;
    }
}

  class Program
  {
        static void Main(string[] args)
        {
              CTest obj_1  = new CTest(5);
              CTest obj_2  = new CTest(20);
              Console.WriteLine("Wert von Objekt 1:"
                + obj_1.GetWert());
              Console.WriteLine("Anzahl der Objekte:"
                + CTest.GetAnzahl());
        }
    }
}
```

Nach dem Starten sieht die Bildschirmausgabe dann so aus:



3.2 Beziehungen zwischen Klassen

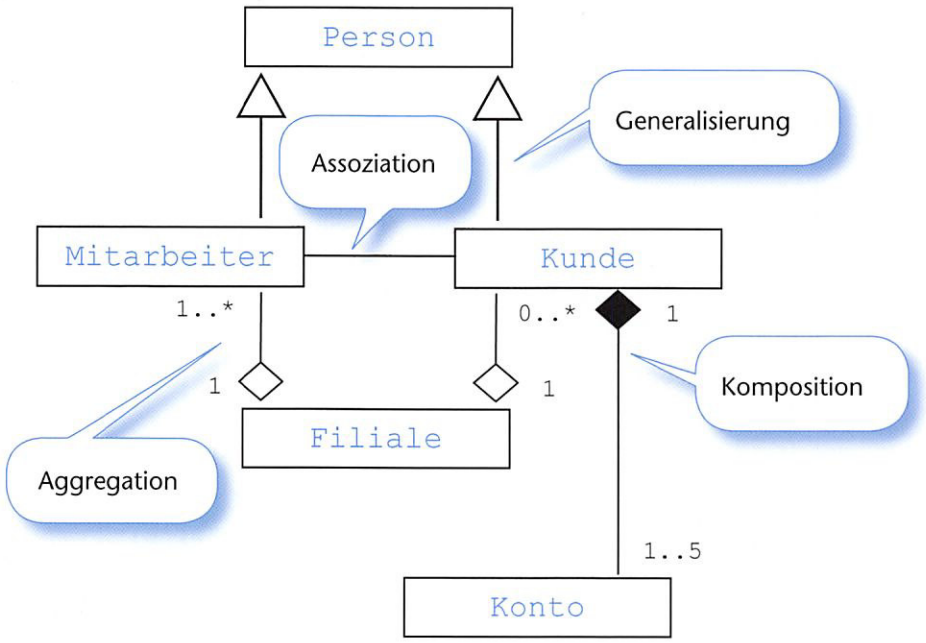
Das Klassendiagramm für ein Softwaresystem besteht natürlich nicht nur aus einer Klasse. Bei komplexen Systemen können es 20 oder mehr Klassen sein, die auch miteinander in Beziehung stehen können. Das wichtige Ziel der objektorientierten Softwareentwicklung ist es ja gerade, die Realität bzw. die Problemstellung in ein adäquates softwaretechnisches Modell abzubilden. Zwischen den Klassen bestehen deshalb auch unterschiedliche Beziehungen, die aus diesem Abbildungsprozess entstanden sind.

Das folgende Beispiel zeigt die Umsetzung einer Problemstellung (Realität) in ein Klassendiagramm. Auf Attribute und Methoden wird zuerst verzichtet. Es geht erst einmal um die Klassen und deren Beziehungen.

Beispiel einer Problemstellung:

Die Filiale einer großen deutschen Bank möchte ihre Kunden und Mitarbeiter EDV-technisch erfassen. Die Kunden können bis zu fünf Konten bei der Bank haben. Weiterhin haben die Kunden einen Mitarbeiter der Filiale als Betreuer.

Umsetzung in ein Klassendiagramm



Das obige Beispiel zeigt die Umsetzung der Problemstellung. Die nötigen Klassen wurden identifiziert und die Beziehungen der Klassen untereinander wurden angelegt. Es handelt sich um die Beziehungen Generalisierung (Vererbung), Assoziation, Aggregation und Komposition. Diese verschiedenen Beziehungen werden in den nächsten Unterkapiteln beleuchtet. Dabei werden auch jeweils konkrete Beispiele in C# umgesetzt.

3.3 Die Assoziation

Die Assoziation ist eine Beziehung zwischen zwei Klassen. Beide Klassen sind auf eine bestimmte Weise miteinander verknüpft, sodass die eine Klasse die andere kennt und die andere Klasse die eine kennt (das kann allerdings auch explizit durch die sogenannte Navigierbarkeit eingeschränkt werden). Ebenso kann es aber auch sein, dass eine Klasse zu sich selbst eine Beziehung hat. Neben dem Kennen der anderen Klasse kann aber auch das Aufrufen einer Methode (Operation) der anderen Klasse die Beziehung ausmachen. Es folgen jetzt zwei Beispiele für Assoziationen, die die obigen etwas theoretischen Ausführungen veranschaulichen sollen.

Beispiel 1:

Ein Angestellter einer Firma betreut die Lieferanten der Firma. Jeder Lieferant hat genau einen Angestellten als Ansprechpartner.

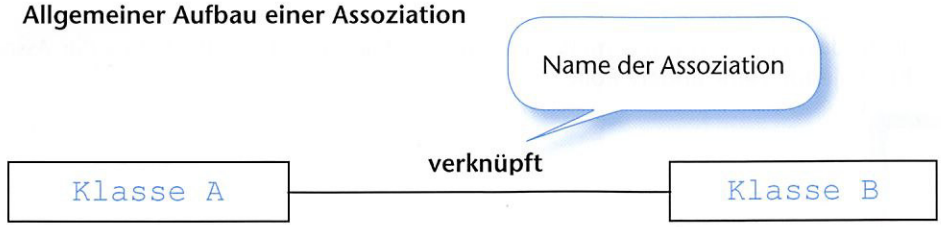


Beispiel 2:

Eine Schulklasse hat einen Klassensprecher. Der Klassensprecher ist nur für diese eine Klasse zuständig.

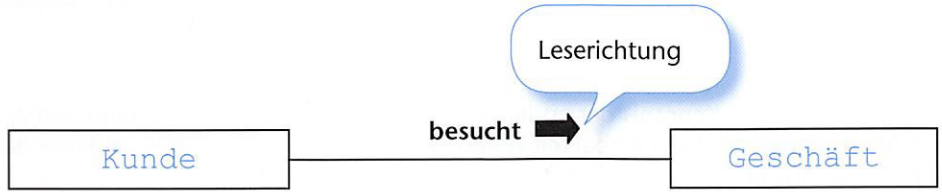


3.3.1 Allgemeiner Aufbau einer Assoziation



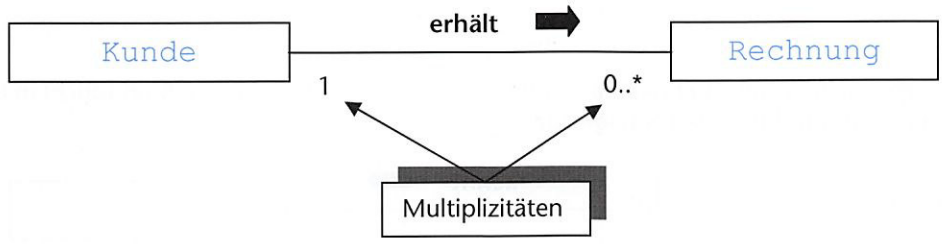
Leserichtung einer Assoziation

Um eine Assoziation näher zu spezifizieren, kann eine Leserichtung (durch einen Richtungspfeil) hinzugefügt werden.



Multiplizitäten einer Assoziation

Die Multiplizitäten einer Assoziation geben an, wie viele Objekte der einen Klasse mit wie vielen der anderen Klasse in Verbindung stehen.



Hinweis:

Die Multiplizitäten sind so zu lesen:

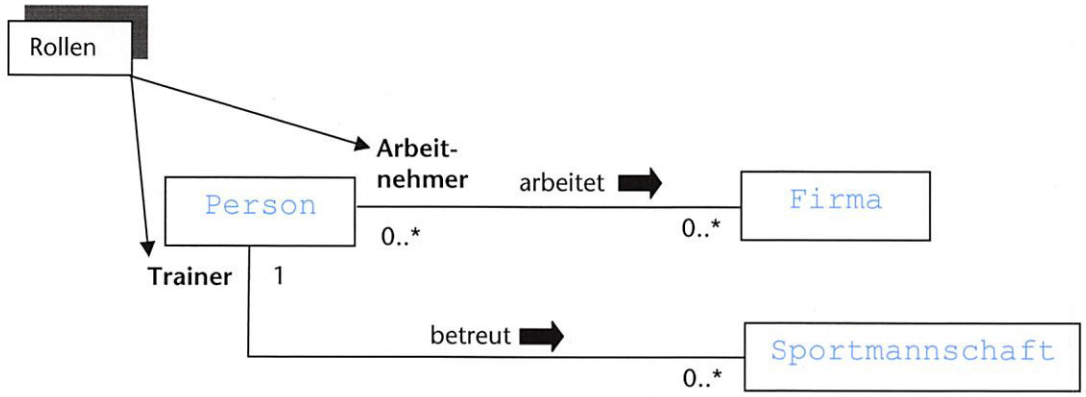
- Ein Kunde erhält keine oder beliebig viele Rechnungen.
- Eine Rechnung gehört genau zu einem Kunden.

Mögliche Multiplizitäten:

Multiplizität	Beschreibung
`0`	keins
`1`	genau eins
`*`	beliebig viele
`0..*`	keins oder beliebig viele (wie *)
`1..*`	eins oder beliebig viele
`1..3`	eins, zwei oder drei
`4..20`	4 bis 20
`1,5,7`	eins, fünf oder sieben

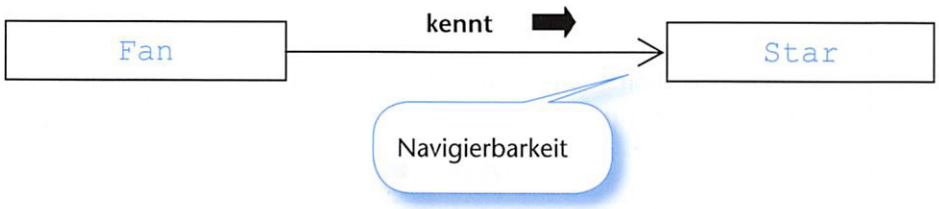
Rollen einer Assoziation

Es ist möglich, dass eine Klasse mit mehreren anderen Klassen verknüpft ist. In jeder Assoziation spielt die Klasse jedoch eine andere Rolle.



Navigierbarkeit

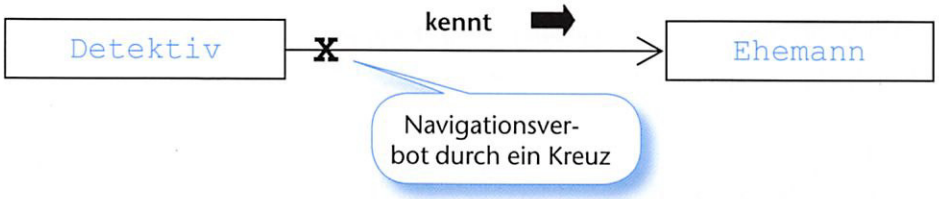
Mit Navigierbarkeit ist die Kenntnis der anderen Klasse gemeint. Man spricht dann von einer **gerichteten Assoziation**. Die Navigierbarkeit (Kenntnis) wird durch einen Pfeil am Ende der Assoziationslinie ausgedrückt.



Der Fan kennt seinen Star, aber der Star kennt seinen Fan in der Regel nicht. Man spricht in diesem Fall von einer **unidirektionalen Navigierbarkeit**.



Ein Kind kennt seinen Vater, der natürlich auch sein Kind kennt. Man spricht in diesem Fall von einer **bidirektionalen Navigierbarkeit**.



Der Detektiv kennt den Ehemann, den er im Auftrag seiner Frau beschatten soll. Der Ehemann darf aber den Detektiv nicht kennen (**Navigationsverbot**).

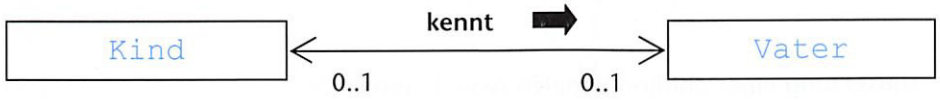
Hinweis:

Wird weder eine Navigierbarkeit noch ein Verbot angegeben, so wird von einer **bidirektionalen** Navigierbarkeit ausgegangen. Beide Klassen kennen sich dann.

3.4 Umsetzung von Assoziationen

Das Umsetzen von Assoziationen in eine objektorientierte Programmiersprache würde im Rahmen der Softwareentwicklung erst nach der Phase des objektorientierten Designs erfolgen. Die eher theoretischen Ausführungen zu den Assoziationen werden aber durch konkrete Umsetzungen besser greifbar. Deshalb werden nun verschiedene Assoziationen in die Programmiersprache C# umgesetzt.

3.4.1 Umsetzung einer bidirektionalen Assoziation in C#



Die Umsetzung erfolgt mithilfe von **Beziehungsattributen**. Das Attribut wird als Verweis auf die andere Klasse deklariert.

```
using System;
namespace UML_IT_BERUFE
{
    class CVater
    {
        private CKind vKind = null;

        public void SetvKind(CKind vK)
        {
            vKind = vK;
        }
    }

    class CKind
    {
        private CVater vVater = null;

        public void SetvVater(CVater vV)
        {
            vVater = vV;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            CVater herrMaier = new CVater();
            CKind caroline = new CKind();
```

Umsetzung des Kennens durch einen Verweis, der mit `null` initialisiert wird.

Der Vater *kennt* jetzt sein Kind.

Noch keine Kenntnis

Das Kind *kennt* jetzt seinen Vater.

Vater und Kind instanzieren.

```
            herrMaier.SetvKind(caroline);
            caroline.SetvVater(herrMaier);
        }
    }
}
```

Umsetzung des *Kennens*

3.4.2 Umsetzung einer unidirektionalen Assoziation in C#

Der Kunde einer Bücherei darf beliebig viele Bücher ausleihen. Ein Buch ist entweder von einem Kunden ausgeliehen oder befindet sich in der Bibliothek.



Die Umsetzung wird so realisiert, dass das Buch nicht seinen Ausleiher speichert sondern nur die Information, ob es ausgeliehen wurde. Der Kunde speichert die ausgeliehenen Bücher in einer eigenen Liste.

```
using System;
using System.Collections;
namespace UML_IT_BERUFE
{
    class CBuch
    {
        private string titel;
        private bool ausgeliehen = false;

        public CBuch(){}
        public void Ausleihen()
        {
            ausgeliehen = true;
        }
        public void Zurueck()
        {
            ausgeliehen = false;
        }
        public bool Status()
        {
            return ausgeliehen;
        }
        public void SetTitel(string t)
        {
            titel = t;
        }
```

Das Attribut gibt den Leihzustand an.

```
    public string GetTitel()
    {
            return titel;
    }
}

class CKunde
{
    private string name;
    private ArrayList liste = new ArrayList();

    public CKunde (){}
    public void SetName(string n)
    {
            name = n;
    }
      public string GetName()
      {
            return name;
      }
      public void BuchLeihen(CBuch buch)
      {
            if (buch.Status() == false)
            {
               liste.Add(buch);
                buch.Ausleihen();
            }
            else Console.WriteLine("Das Buch: " + buch.GetTitel()
                + " ist bereits verliehen!");
      }
      public void BuchZurueckgeben(CBuch buch)
      {
            bool zurueck = false;
            foreach (CBuch b in liste)
            {

              if (b==buch)
              {
                 Console.WriteLine("Das Buch: " +
                 buch.GetTitel() + " wurde zurueckgegeben!");
                 buch.Zurueck();
                 liste.Remove(buch);
                 zurueck = true;
                 break;
              }
```

Eine dynamische Liste für die ausgeliehenen Bücher

Das Buch wird nur in die Liste eingetragen, wenn es ausleihbar ist.

Die Rückgabe eines Buches wird *simuliert*.

```
            }
            if (!zurueck)
            Console.WriteLine("Das Buch war nicht ausgeliehen!");
        }
        public void ListeAusgeben()
        {
            foreach (CBuch b in liste)
            {
                Console.WriteLine(b.GetTitel());
            }
        }
    }

class Program
{

    static void Main(string[] args)
    {
        CKunde [] kunden = new CKunde [3];
        CBuch [] buecher = new CBuch [5];
        for (int i = 0; i < 3; i++) kunden[i] = new CKunde();
        for (int i = 0; i < 5; i++) buecher[i] = new CBuch();

        kunden[0].SetName("Hansen");
        kunden[1].SetName("Maier");
        kunden[2].SetName("Mueller");

        buecher[0].SetTitel("Einfuehrung in C++");
        buecher[1].SetTitel("Einfuehrung in C#");
        buecher[2].SetTitel("Datenbanken");
        buecher[3].SetTitel("Einfuehrung in UML");
        buecher[4].SetTitel("OOA und OOP");
        kunden[0].BuchLeihen(buecher[0]);
        kunden[0].BuchLeihen(buecher[1]);
        kunden[1].BuchLeihen(buecher[2]);
        kunden[1].BuchLeihen(buecher[3]);
        kunden[2].BuchLeihen(buecher[4]);

        Console.WriteLine("Buecherliste: \n");
        for (int i=0; i < 3 ; i++)
        {
            Console.WriteLine("Kunde " + kunden[i].GetName() + "
                    hat folgende Buecher:");
            kunden[i].ListeAusgeben();
        }
        kunden[2].BuchLeihen(buecher[2]);
```

Kunden benennen

Bücher benennen

Realisierung der Assoziation

Ein Leih-versuch!

```
            Console.WriteLine();
            kunden[1].BuchZurueckgeben(buecher[2]);
            Console.WriteLine("Kunde " + kunden[1].GetName() + " hat
                        nun folgende Buecher:");
            kunden[1].ListeAusgeben();
            Console.WriteLine();
            kunden[2].BuchLeihen(buecher[2]);
            Console.WriteLine("Kunde " + kunden[2].GetName() + " hat
                        nun folgende Buecher:");
            kunden[2].ListeAusgeben();
        }
    }
}
```

Zuerst die Rückgabe

Neuer Leihversuch!

Nach dem Starten sieht die Bildschirmausgabe so aus:

```
C:\WINDOWS\system32\cmd.exe
Buecherliste:

Kunde Hansen hat folgende Buecher:
Einfuehrung in C++
Einfuehrung in C#

Kunde Maier hat folgende Buecher:
Datenbanken
Einfuehrung in UML

Kunde Mueller hat folgende Buecher:
OOA und OOP


Das Buch: Datenbanken ist bereits verliehen!


Das Buch: Datenbanken wurde zurueckgegeben!
Kunde Maier hat nun folgende Buecher:
Einfuehrung in UML

Kunde Mueller hat nun folgende Buecher:
OOA und OOP
Datenbanken

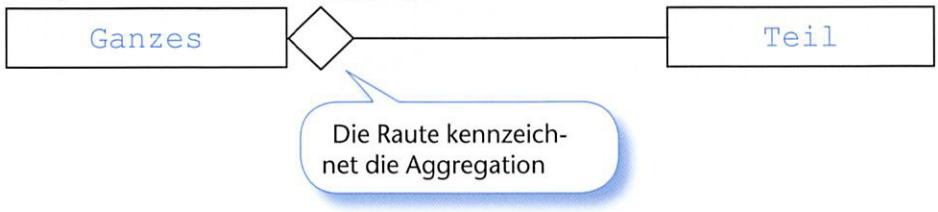
Drücken Sie eine beliebige Taste . . .
```

3.5 Die Aggregation

Die Aggregation ist eine spezielle Assoziation. Die miteinander verknüpften Klassen beschreiben dabei eine **Ganzes-Teile-Beziehung**. Das bedeutet, dass eine Klasse das Ganze verkörpert und die andere Klasse einen Teil davon. Das Ganze hat den Teil als Komponente. Man spricht auch von einer **Hat-Beziehung**. Im Gegensatz dazu wird bei der Generalisierung bzw. Spezialisierung (oder auch Vererbung) eine **Ist-Beziehung** modelliert.

Die Aggregation wird dadurch gekennzeichnet, dass sie eine Raute an dem Ganzen besitzt.

3.5.1 Allgemeiner Aufbau einer Aggregation

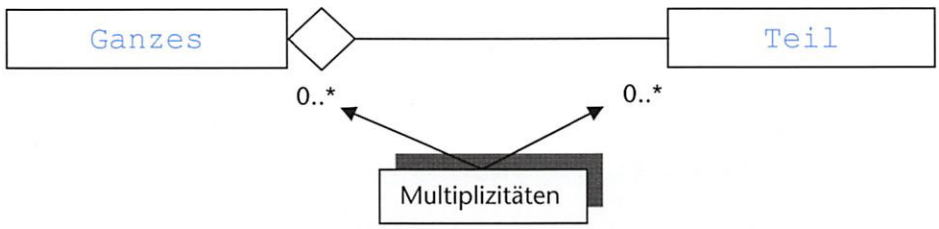


Für eine Aggregation gelten folgende Eigenschaften:

- Sowohl das Ganze als auch die Teile können unabhängig voneinander existieren. Auch wenn Teile entfernt werden, kann das Ganze existieren.
- Teile können in verschiedenen Ganzen verwendet werden.

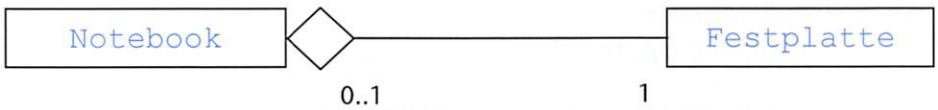
Multiplizitäten einer Aggregation

Ebenso wie bei einer Assoziation kann die Aggregation mit Multiplizitäten versehen werden.



Beispiele

- Ein Notebook hat eine Festplatte. Eine Festplatte ist entweder nicht oder in genau einem Notebook eingebaut.



- In einer Firma arbeiten beliebig viele Mitarbeiter. Die Mitarbeiter können aber auch bei beliebig vielen anderen Firmen arbeiten (zumindest theoretisch).



3.5.2 Umsetzung einer *0..1:1*-Aggregation in C#

Das Beispiel mit dem Notebook und der Festplatte soll umgesetzt werden.



Bei der Umsetzung ist zu beachten, dass sowohl das Ganze als auch die Teile für sich existent sein müssen. Allerdings muss ein Notebook über eine Festplatte verfügen (Eins-Multiplizität). Das muss bei der Konstruktion eines Notebooks beachtet werden. Ein Ausbau der Festplatte kann nur unter gleichzeitigem Einbau einer anderen Festplatte geschehen. Die Aufnahme einer Festplatte erfolgt mittels eines Verweises. Die Umsetzung der Notebook-Festplatte-Aggregation ähnelt der Umsetzung der Assoziation mit einem Beziehungsattribut – eine Aggregation ist ja auch eine spezielle Assoziation.

```
using System;
using System.Collections;
namespace UML_IT_BERUFE
{
    class CFestplatte
    {
        private bool eingebaut = false;

        public CFestplatte() { }
        public void Einbau() { eingebaut = true; }
        public void Ausbau() { eingebaut = false; }
        public bool GetStatus() { return eingebaut; }
    }
    class CNotebook
    {
        private CFestplatte vFestplatte = null;

        public CNotebook(CFestplatte vF)
        {
            if (vF.GetStatus() == false)
            {
                vFestplatte = vF;
                vFestplatte.Einbau();
            }
            else
            Console.WriteLine("Keine korrekte Erstellung moeglich
                        - Festplatte fehlt!!!");
        }

        public void WechselFestplatte(CFestplatte vF)
        {
            if (vF.GetStatus() == false)
            {
                vFestplatte.Ausbau();
                vFestplatte = vF;
                vFestplatte.Einbau();
                Console.WriteLine("Festplatte gewechselt");
            }
            else Console.WriteLine("Kein Wechsel moeglich!");
        }

    }

    class Program
    {
```

Dieses Attribut zeigt, ob die Festplatte eingebaut ist. Zu Beginn natürlich nicht.

Ein Verweis auf die Festplatte

Prüfen, ob die Festplatte bereits eingebaut ist.

Festplatte einbauen

Botschaft senden

Festplatte wechseln

```
        static void Main(string[] args)
        {
            CFestplatte f1 = new CFestplatte();
            CFestplatte f2 = new CFestplatte();

            CNotebook n1 = new CNotebook(f1);
            CNotebook n2 = new CNotebook(f1);

            Console.WriteLine();
            n1.WechselFestplatte(f2);
            Console.WriteLine();
        }
    }
}
```

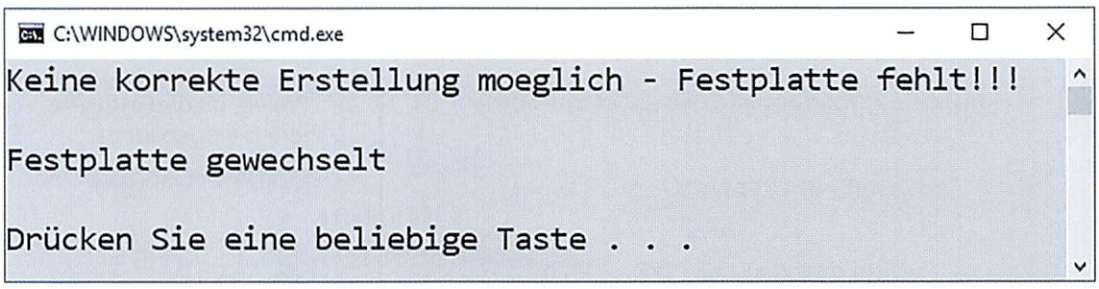
Zwei Festplatten anlegen

Ein Notebook mit Festplatte 1 instanzieren

Der Versuch, ein zweites Notebook mit Festplatte 1 zu instanzieren, scheitert.

Notebook 1 wechselt die Festplatte.

Nach dem Starten sieht die Bildschirmausgabe dann so aus:



3.5.3 Umsetzung einer *:*-Aggregation in C#

Die Umsetzung in C# arbeitet mit der `ArrayList`-Klasse. Damit sind sowohl dynamische Reservierung von Speicher als auch Freigabe automatisiert.

```
using System;
using System.Collections;
namespace UML_IT_BERUFE
{
    class CMitarbeiter
    {
        private string name = "NN";
        public CMitarbeiter() { }
        public void SetName(string n) { name = n; }
        public string GetName() { return name; }
    }

    class CFirma
    {
        private ArrayList liste = new ArrayList();
```

Liste für die Mitarbeiter

```
    public CFirma() { }
    public void Einfuegen(CMitarbeiter vM)
    {
        liste.Add(vM);
    }

    public void Loeschen(int nummer)
    {
        liste.RemoveAt(nummer - 1);
    }
    public void Ausgabe()
    {
        int i = 1;
        foreach (CMitarbeiter vM in liste)
        {
            Console.WriteLine("Mitarbeiter " + i + ": "
                    + vM.GetName());
            i++;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        CMitarbeiter[] mitarbeiter = new CMitarbeiter[3];
        mitarbeiter[0] = new CMitarbeiter();
        mitarbeiter[1] = new CMitarbeiter();
        mitarbeiter[2] = new CMitarbeiter();
        mitarbeiter[0].SetName("Maier");
        mitarbeiter[1].SetName("Hansen");
        mitarbeiter[2].SetName("Mueller");
        CFirma firma1 = new CFirma();
        CFirma firma2 = new CFirma();
        firma1.Einfuegen(mitarbeiter[0]);
        firma1.Einfuegen(mitarbeiter[1]);
        firma1.Einfuegen(mitarbeiter[2]);
        firma2.Einfuegen(mitarbeiter[0]);
        firma2.Einfuegen(mitarbeiter[2]);
        Console.WriteLine("Firma 1:");
        firma1.Ausgabe();
        Console.WriteLine();
        Console.WriteLine("Firma 2:");
```

Einfügen eines Mitarbeiters

Entlassen eines Mitarbeiters

Drei Mitarbeiter anlegen

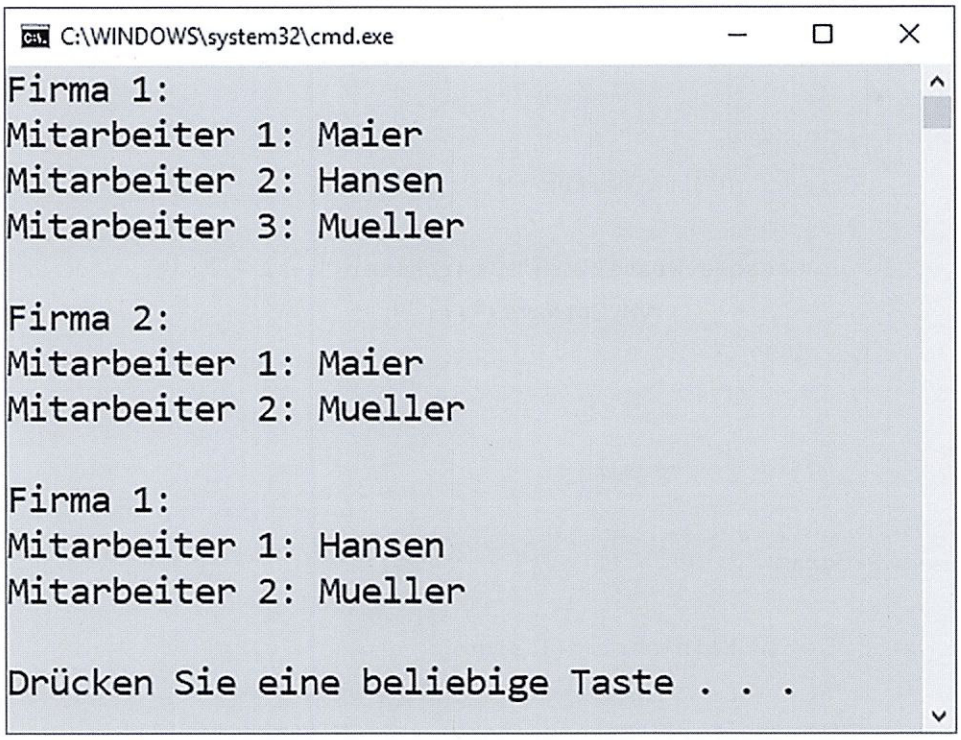
Zwei Firmen anlegen

Mitarbeiter zuweisen

```
            firma2.Ausgabe();
            Console.WriteLine();
            Console.WriteLine("Firma 1:");
            firma1.Loeschen(1);
            firma1.Ausgabe();
        }
    }
}
```

Einen Mitarbeiter entlassen

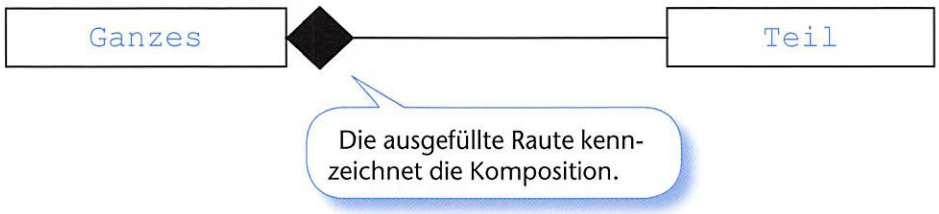
Nach dem Starten sieht die Bildschirmausgabe dann so aus:



3.6 Die Komposition

Die Komposition ist eine Aggregation, die eine zusätzliche besondere Eigenschaft hat. Es gibt eine starke Abhängigkeit zwischen dem Ganzen und den Teilen, sodass die Teile nicht ohne das Ganze existieren können. Ebenso wie bei der Aggregation spricht man von einer **Hat-Beziehung**. Die Komposition wird dadurch gekennzeichnet, dass sie eine ausgefüllte Raute an dem Ganzen besitzt.

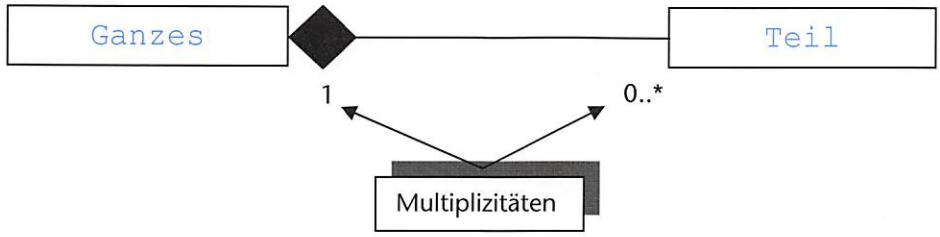
3.6.1 Allgemeiner Aufbau einer Komposition



Die ausgefüllte Raute kennzeichnet die Komposition.

Multiplizitäten einer Komposition

Ebenso wie bei der Assoziation und der Aggregation kann die Komposition mit Multiplizitäten versehen werden.



Hinweise:

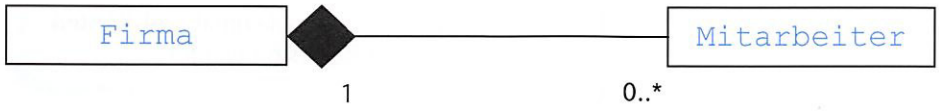
Die Multiplizität auf der Seite des Ganzen **darf immer nur 1** sein, sonst würde das der starken Bindung des Teils an das Ganze widersprechen. Wenn das Ganze gelöscht wird, so werden auch automatisch alle Teile gelöscht. Teile können auch gelöscht werden, bevor das Ganze seine Gültigkeit verliert. Ein Teil kann aber vor dem Löschen des Ganzen noch von einem anderen Ganzen aufgenommen werden, sodass das Teil in einem anderen Ganzen weiter existieren kann.

Beispiele

- Ein Kunde hat bis zu fünf Konten. Ein Konto ist aber immer nur mit einem Kunden verbunden. Wird der Kunde gelöscht, so werden auch seine Konten gelöscht.



- In einer Firma arbeiten beliebig viele Mitarbeiter. Die Mitarbeiter sind aber (vertraglich festgelegt) nur in genau einer Firma tätig. Würde die Firma nicht mehr existieren, so könnten die Mitarbeiter vorher zu einer anderen Firma wechseln.



3.6.2 Umsetzung einer *1:1..5*-Komposition in C#

Die Umsetzung in C# arbeitet wieder mit der Klasse `ArrayList`. Damit werden die Konten dynamisch angelegt.

```
using System;
using System.Collections;
namespace UML_IT_BERUFE
{
    class CKonto
    {
        private int nummer;

        public CKonto() { nummer = -1; }
        public bool SetNummer(int n)
        {
           if (n > 0 && n < 9999)
           {
              nummer = n;
              return true;
           }
```

```
            return false;
        }
        public int GetNummer()
        {
            return nummer;
        }
}
class CKunde
{
    private string name;
    private ArrayList konten = new ArrayList();

    public CKunde()
    {
        NeuesKonto();
    }
    public void SetName(string n)
    {
        name = n;
    }
    public string GetName()
    {
        return name;
    }
    public void NeuesKonto()
    {
        if (konten.Count==5) return;
        CKonto k = new CKonto();
        bool ok;
        int nr;
        do
        {
          Console.Write("Bitte die Nummer fuer Konto " +
                  (konten.Count + 1) + " angeben: ");
          nr = Convert.ToInt32(Console.ReadLine());
          ok = k.SetNummer(nr);
          if (!ok)
            Console.WriteLine("Fehlerhafte Kontonummer!");
        }
        while (!ok);
        konten.Add(k);
    }
    public void Ausgabe()
    {
        Console.WriteLine("Kunde " + name
                          + " hat folgende Konten:");
```

Die `ArrayList konten` verwaltet die fünf Konten.

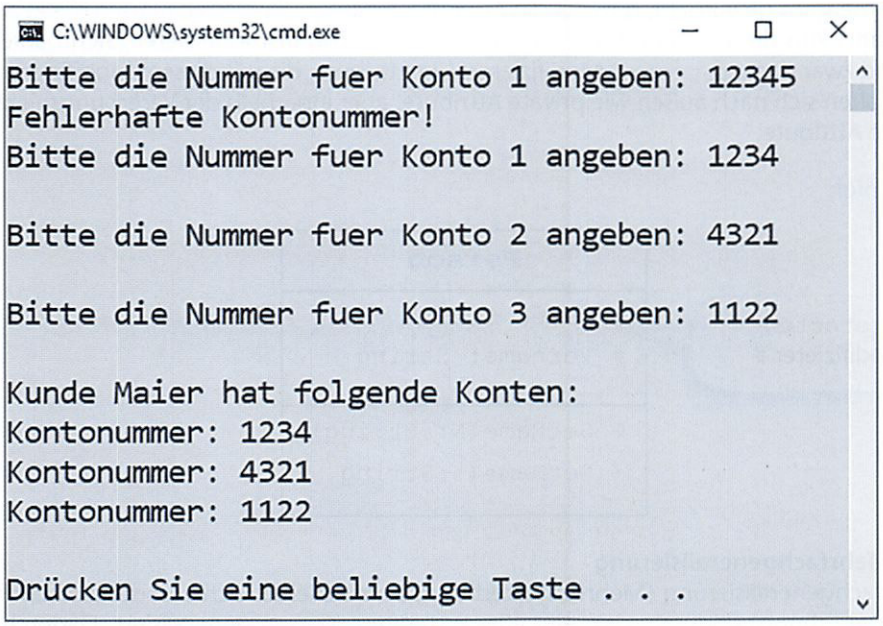
Umsetzung der *1..5*-Beziehung. Ein Konto wird immer angelegt.

Maximal fünf Konten sind erlaubt.

Hinzufügen eines neuen Kontos

```
            foreach (CKonto k in konten)
            {
                Console.WriteLine("Kontonummer: " + k.GetNummer());
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CKunde kunde = new CKunde ();
            kunde.SetName("Maier");
            Console.WriteLine();
            kunde.NeuesKonto();
            Console.WriteLine();
            kunde.NeuesKonto();
            Console.WriteLine();
            kunde.Ausgabe();
        }
    }
}
```

Nach dem Starten könnte die Bildschirmausgabe so aussehen:

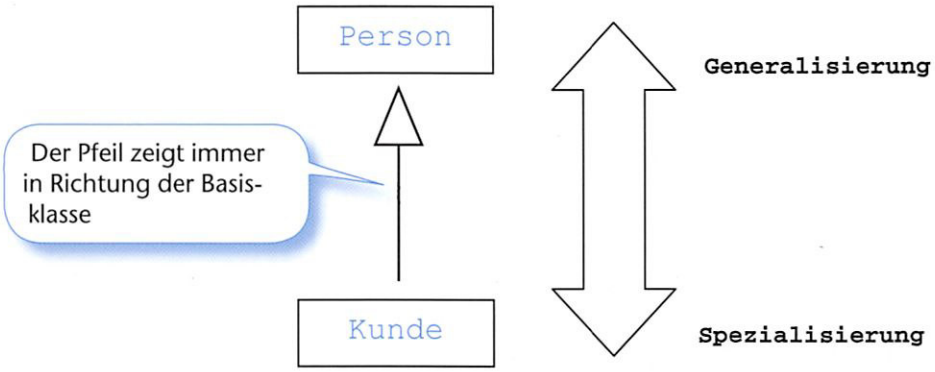


3.7 Die Generalisierung und Spezialisierung

Das Konzept der Generalisierung (bzw. Spezialisierung oder auch Vererbung) ist ein zentrales Thema in der objektorientierten Softwareentwicklung. Durch dieses Konzept können Situationen aus der realen Welt besser in einem Modell abgebildet werden. Zusätzlich hat dieses Konzept auch sehr praktische Vorteile, denn bereits existierender Programmcode (in Form von Klassen) kann durch dieses Konzept sehr gut wiederverwendet werden. Die Softwareentwicklung wird dadurch auch effizienter und sicherer. Bei der Generalisierung spricht man von einer sogenannten **Ist-Beziehung,** im Gegensatz zur bereits bekannten **Hat-Beziehung** bei der Aggregation und Komposition.

Beispiel:

Die Klasse `Kunde` ist eine spezielle Klasse `Person`. Die Klasse `Person` ist eine Verallgemeinerung der Klasse `Kunde`. Aus diesem Grund spricht man auch von Generalisierung und Spezialisierung.



Die Basis-Klasse `Person` vererbt an die Klasse `Kunde`. Der Kunde ist eine Person (Ist-Beziehung). Der Kunden-Klasse stehen nun alle Elemente der Basis-Klasse zur Verfügung (mit gewissen Einschränkungen, siehe später). Wenn beispielsweise die Personen-Klasse ein Attribut `name` hat, so *erbt* der Kunde dieses Attribut.

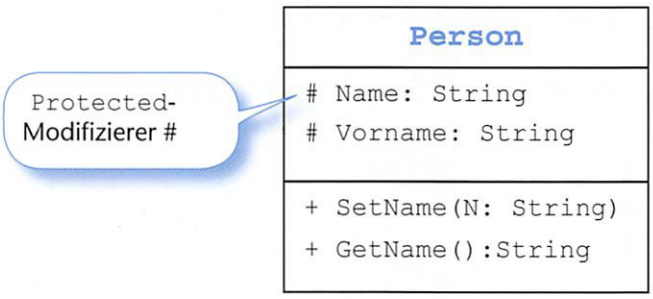
Hinweis:

Die Klasse, die vererbt (`Person`), wird in der Regel Basis-Klasse oder Ober-Klasse genannt. Die Klasse, die erbt (`Kunde`), wird abgeleitete Klasse oder Unter-Klasse genannt.

3.7.1 Sichtbarkeit von Attributen

In einer spezialisierten (vererbten) Klasse können alle öffentlichen Elemente der Basis-Klasse so genutzt werden, als wären sie in der Klasse selbst angelegt. Der direkte Zugriff auf private Elemente ist hingegen gesperrt und nur über Methoden möglich. Innerhalb einer Vererbungshierachie ist es allerdings meistens gewünscht, dass auf die Attribute in den spezialisierten Klassen direkt zugegriffen werden kann. Aus diesem Grund gibt es einen weiteren Zugriffsmodifizierer (Sichtbarkeitsmodifizierer), und zwar den `protected`-Modifizierer. Alle Attribute, die mit diesem Modifizierer versehen sind, verhalten sich nach außen wie private Attribute, aber innerhalb der Vererbungshierarchie wie öffentliche Attribute.

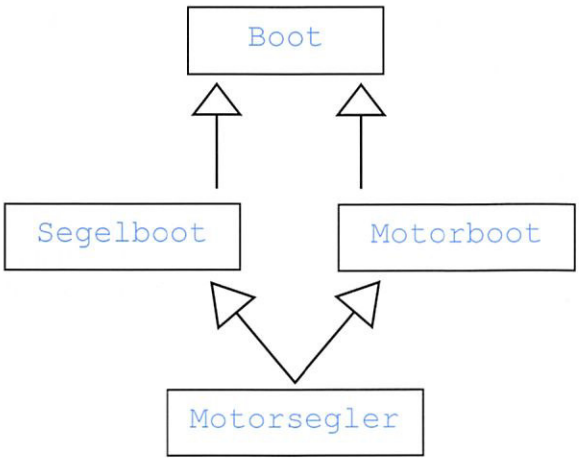
Beispiel:



3.7.2 Mehrfachgeneralisierung

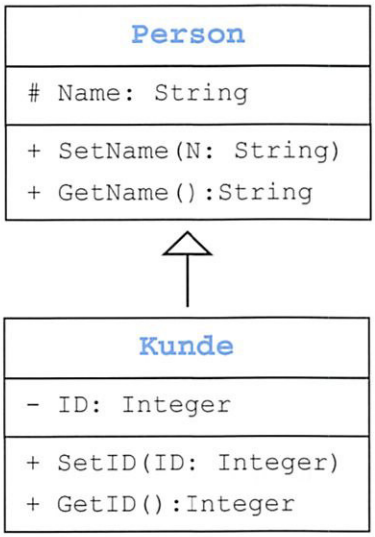
Die Mehrfachgeneralisierung (Mehrfachvererbung) unterscheidet sich von der einfachen Generalisierung (Vererbung) dadurch, dass eine Klasse mehrere Klassen generalisiert. Dadurch kann ein Softwaresystem noch realitätsnäher umgesetzt werden. Die Mehrfachgeneralisierung hat aber den Nachteil, dass sie deutlich komplexer als die einfache Generalisierung ist. Bei der Umsetzung müssen deshalb mehr Aspekte beachtet werden. Eine Programmiersprache wie C# (oder auch Java) wurde aus diesem Grund ohne die Technik der Mehrfachgeneralisierung entworfen. Das schränkt zwar den Entwickler etwas ein, führt aber zu mehr Sicherheit bei der Programmentwicklung.

Beispiel:



3.7.3 Umsetzung einer einfachen Generalisierung in C#

Das Beispiel mit der Person und der Spezialisierung als Kunde soll in C# umgesetzt werden. Das Attribut `Name` ist mit dem `protected`-Modifizierer versehen. Damit ist der Zugriff in der Kunden-Klasse leichter.



In C# gibt es nur eine Art der Vererbung – und zwar implizit die `public`-Vererbung. Da C# keine Mehrfachvererbung unterstützt (und damit nicht mehr als eine Basis-Klasse möglich ist), wird die Basis-Klasse einfach mit dem Schlüsselwort `base` angesprochen.

```
using System;
namespace UML_IT_BERUFE
{
   class CPerson
   {
         protected string name;

         public CPerson() { name = "NN"; }
         public void SetName(string n) { name = n; }
         public string GetName(){ return name; }
   }
   class CKunde: CPerson
   {
```

Attribut `name` wird `protected`.

Implizite `public`-Vererbung

```
        private int id;

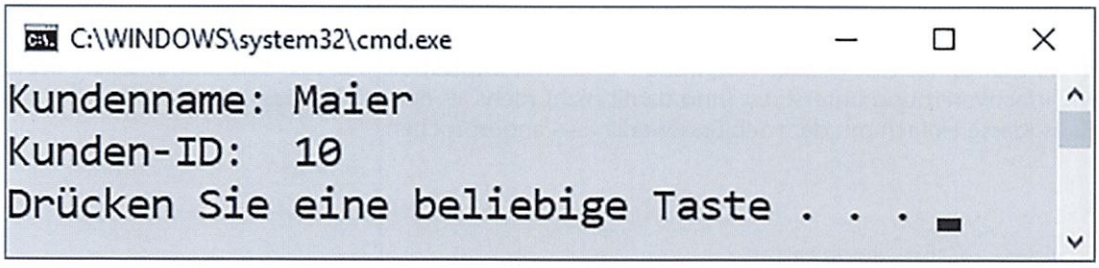
        public CKunde() : base() { id = 0; }
        public void SetID(int i)
        {
            if (i > 0 && i < 9999) id = i;
        }
        public int GetID() { return id; }
        public void Ausgabe()
        {
           Console.WriteLine("Kundenname: " + name);
           Console.WriteLine("Kunden-ID:  " + id);
        }
    }
    class Program
    {
       static void Main(string[] args)
       {
           CKunde kunde = new CKunde();

           kunde.SetName("Maier");
           kunde.SetID(10);
           kunde.Ausgabe();
       }
    }
}
```

Aufruf des Basis-klassen-Konstruktors

Direkter Zugriff auf das Attribut `name` möglich – dank `protected`.

Nach dem Starten sieht die Bildschirmausgabe so aus:

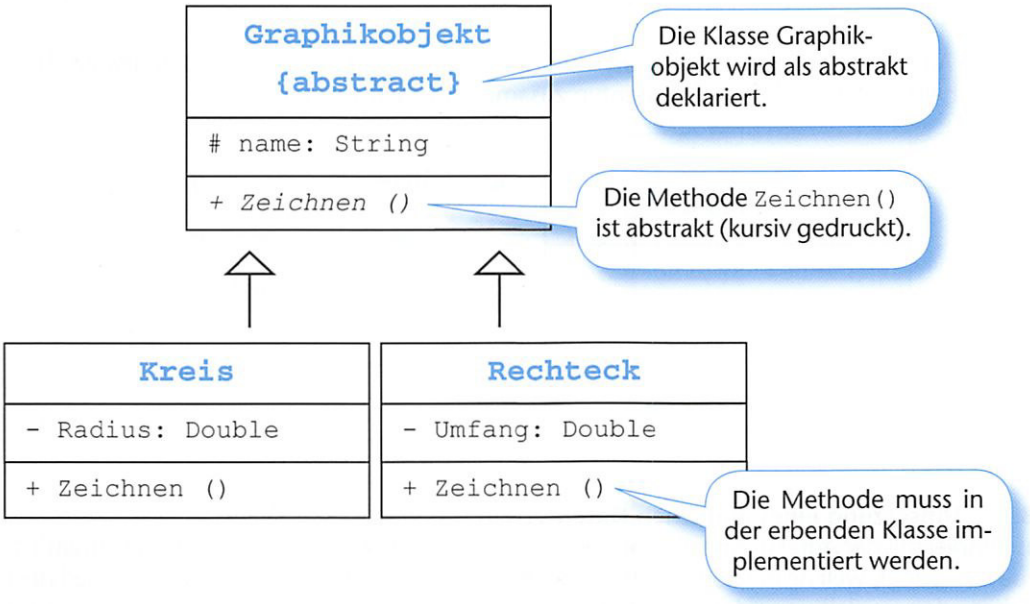


3.7.4 Abstrakte Basis-Klassen

Eine abstrakte Basis-Klasse ist eine Klasse, von der keine Objekte instanziert werden können. Sie dient als Grundlage für die weiteren Klassen einer Vererbungshierarchie. In einer abstrakten Basis-Klasse können Methoden als abstrakt deklariert werden. Diese Methoden sind nicht implementiert, müssen aber in der erbenden Klasse implementiert werden.

Beispiel:

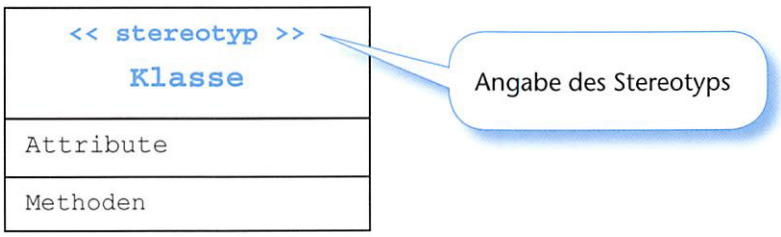
Ein Anwendungsbeispiel für abstrakte Basis-Klassen ist eine Klassenhierarchie zur Speicherung von grafischen Objekten (Kreise, Dreiecke, Rechtecke usw.). Jedes Objekt soll auf dem Bildschirm gezeichnet werden können. Deshalb ist es sinnvoll, eine abstrakte Basis-Klasse `Graphikobjekt` zu entwerfen, die über eine abstrakte Methode `Zeichnen()` verfügt. Alle Klassen, die von dieser Klasse abgeleitet werden, müssen die Methode `Zeichnen()` implementieren.



3.8 Stereotype

Mithilfe von Stereotypen kann die Bedeutung einer Klasse weiter spezifiziert werden. Dadurch soll noch deutlicher werden, in welchem Zusammenhang eine Klasse genutzt werden soll. Zusätzlich sind damit auch Implementierungsregeln verbunden. Das Stereotyp gibt dann vor, wie eine Klasse umzusetzen ist. Neben einigen vordefinierten Stereotypen in der UML können auch eigene Stereotypen entwickelt werden, um neue Bedeutungen für das zu entwickelnde Softwaresystem zu erschaffen. An dieser Stelle sollen allerdings nur drei wichtige vordefinierte Stereotypen vorgestellt werden. Die Definition eigener Stereotypen empfiehlt sich eher für fortgeschrittene Entwickler. Das Stereotyp wird mithilfe der *französischen Anführungsstriche* << >> über dem Klassennamen angegeben.

Beispiel:



3.8.1 Primitive und einfache Datentypen

Die bisher verwendeten Attribute in den Klassendiagrammen haben alle einen bestimmten Datentyp (beispielsweise `Integer` oder `String`). Die UML stellt dafür die sogenannten ***primitiven Datentypen*** bereit. Diese elementaren Datentypen werden in (fast) allen Programmiersprachen unterstützt. Die meisten Programmiersprachen besitzen natürlich noch weitere Typen.

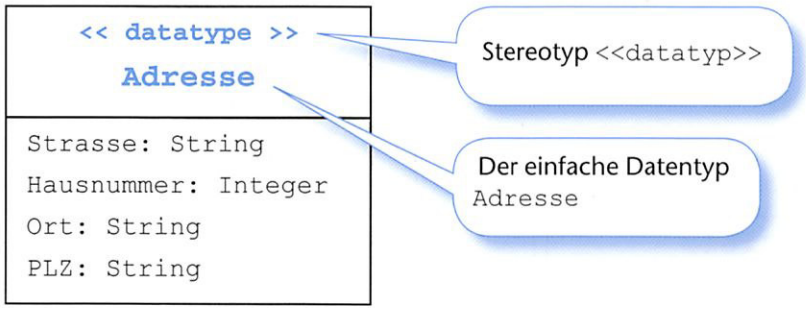
Die vier primitiven Datentypen der UML lauten:

- `Integer` (ganzzahlige Werte)
- `String` (Zeichenketten)
- `Boolean` (boolesche Werte `true` oder `false`)
- `UnlimitedNatural` (natürliche Zahlen)

Die ***einfachen Datentypen*** in der UML sind hingegen eine Erweiterung der primitiven Datentypen und mit dem Konzept der Struktur (Verbund, Record) in einer Programmiersprache vergleichbar. Ein einfacher Datentyp besitzt beliebig viele Attribute von primitiven Datentypen oder auch von anderen einfachen Datentypen. Ein einfacher Datentyp kann auch Methoden enthalten. **Ein einfacher Datentyp ist aber trotzdem keine Klasse.**

Beispiel:

Die Adresse einer Person besteht aus mehreren Komponenten primitiver Datentypen. Sie kann deshalb zu einem einfachen Datentyp zusammengefasst werden.



3.8.2 Umsetzung eines einfachen Datentyps in C#

Das Beispiel mit der Adresse einer Person soll in C# umgesetzt werden. Dazu wird eine Struktur `TAdresse` angelegt, welche die Komponenten der Adresse enthält. Das vorangestellte „`T`" soll deutlich machen, dass es sich um einen neuen Datentyp handelt – im Gegensatz zu dem vorangestellten „`C`" bei der Umsetzung einer Klasse.

```
struct TAdresse
{
   public string strasse;
   public int hausnummer;
   public string ort;
   public string plz;
}
class Program
{
   static void Main(string[] args)
   {
      TAdresse eineAdresse = new TAdresse();
      eineAdresse.strasse = "Meisenweg";
      eineAdresse.hausnummer = 15;
      eineAdresse.ort = "Koeln";
      eineAdresse.plz = "50000";
   }
}
```

Datentyp `TAdresse` wird angelegt.

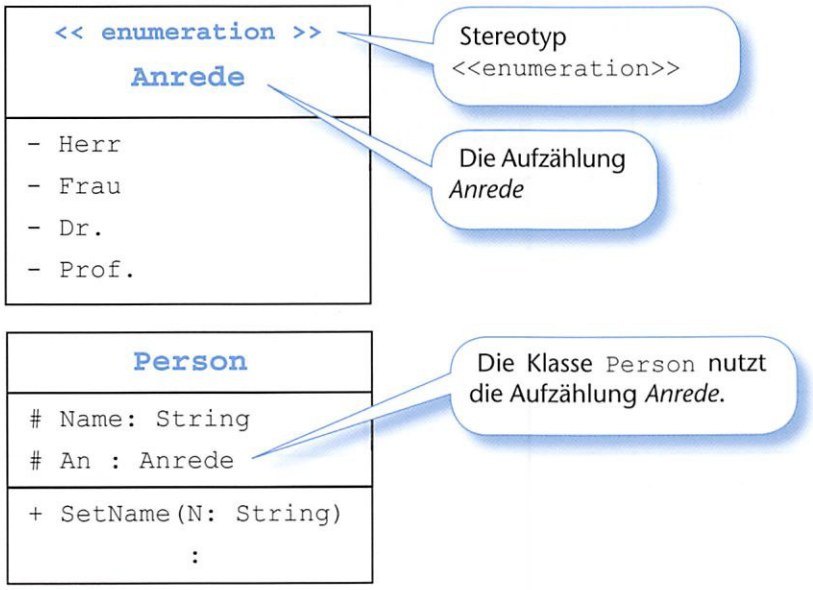
Eine Variable des neuen Typs wird mit Werten gefüllt.

3.8.3 Aufzählungen

In manchen Fällen hat ein Attribut nur ganz bestimmte (und endlich viele) Werte. Beispielsweise kann das Attribut Anrede (bei einer Person) die Werte „Herr", „Frau", „Dr." oder auch „Prof." annehmen. Dann ist es sinnvoll, dafür einen eigenen Datentyp anzulegen – eine sogenannte Aufzählung (engl. *enumeration*). In dieser Aufzählung werden dann die möglichen Werte aufgezeigt. Eine Aufzählung wird durch das Stereotyp `<<enumeration>>` gekennzeichnet.

Beispiel:

Die Aufzählung `Anrede` wird definiert und anschließend in einer Klasse `Person` genutzt.



Hinweis:

Die Aufzählungen werden in C# so umgesetzt:

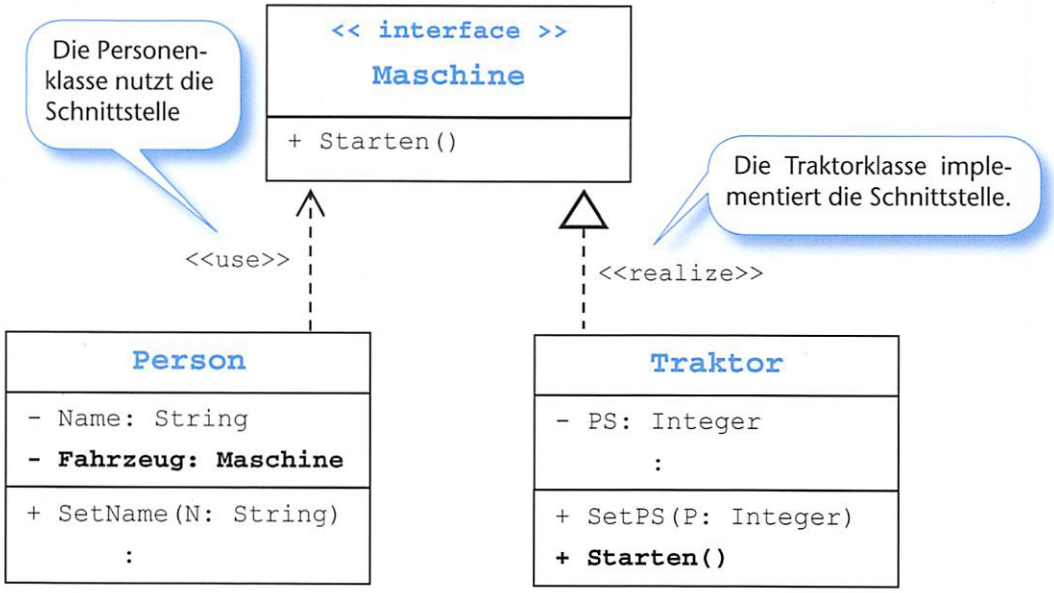
```
public enum Anrede { HERR, FRAU, DR, PROF }
Anrede an = Anrede.HERR;
```

3.8.4 Schnittstellen

Das Prinzip der Schnittstellen (engl. *interfaces*) hat viel Ähnlichkeit mit dem Konzept der abstrakten Basis-Klassen. Eine Schnittstelle sieht aus wie eine Klasse, aber ohne jegliche Implementierungen. Die Schnittstelle legt nur fest, welche Attribute und Methoden genutzt werden sollen. Eine Klasse, die eine Schnittstelle implementiert, muss diese Attribute und Methoden dann auch definieren. Eine Klasse kann beliebig viele Schnittstellen implementieren. Schnittstellen müssen aber nicht immer implementiert, sondern können auch einfach nur genutzt werden. Damit sind nutzende und implementierende Klasse in der Lage zu kommunizieren – die Schnittstelle gibt dabei vor, wie die Kommunikation aussieht. Eine Schnittstelle wird durch das Stereotyp `<<interface>>` gekennzeichnet.

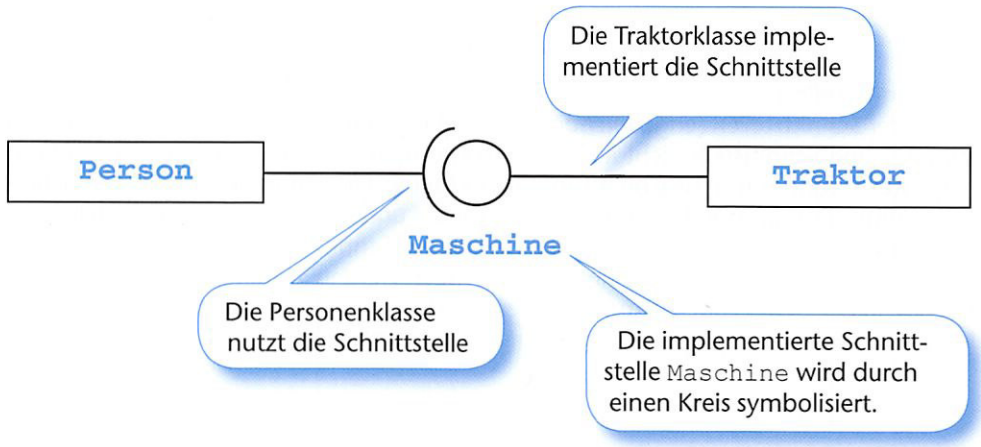
Beispiel:

Eine Person möchte einen Traktor nutzen. Dabei soll diese Nutzung über eine Schnittstelle `Maschine` realisiert werden. Die Traktor-Klasse implementiert dabei die Schnittstelle und die Personen-Klasse nutzt die Schnittstelle.



Hinweis:

Die Darstellung der Schnittstelle kann alternativ über das sogenannte `Ball and Socket`-Symbol geschehen:



3.8.5 Umsetzung einer Schnittstelle in C#

Das Person-Maschine-Traktor-Beispiel soll nun in C# umgesetzt werden.

```
using System;
namespace UML_IT_BERUFE
{
    interface IMaschine
    {
        void Starten();
    }
    class CTraktor: IMaschine
    {
        protected int ps;
```

Das Interface (Schnittstelle) wird angelegt. Es verfügt (der Einfachheit halber) nur über die Methode `Starten()`.

So wie bei der Vererbung wird das zu implementierende Interface hinter dem Klassennamen angegeben.

```
        public CTraktor() { ps = 0; }
        public void SetPS(int p) { ps = p; }
        public int GetPS() { return ps; }
        public void Starten()
        {
            Console.WriteLine("Der Traktor laeuft....");
        }
    }

    class CPerson
    {
        protected string name;
        protected IMaschine traktor;
        public CPerson() { name = "NN"; }
        public void SetName(string n) { name = n; }
        public string GetName() { return name; }
        public void NutzeTraktor()
        {
            traktor = new CTraktor();
            traktor.Starten();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CPerson einePerson = new CPerson();
            einePerson.NutzeTraktor();
        }
    }
}
```

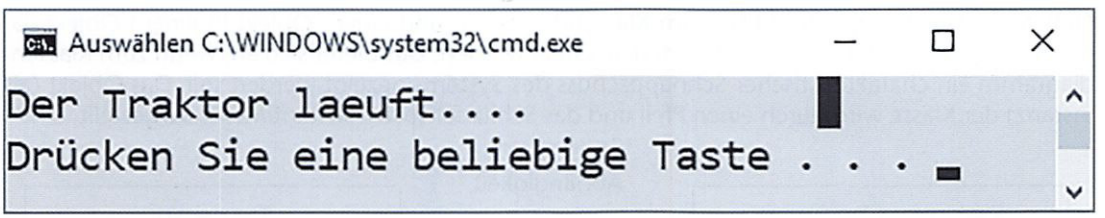
Implementierung der Methode `Starten()`

Die Personen-Klasse nutzt das Interface, indem es einen Verweis `traktor` anlegt.

Dem Interface-Verweis wird nun eine Traktor-Instanz zugewiesen.

Die Person nutzt den Traktor – dank Interface.

Nach dem Starten sieht die Bildschirmausgabe so aus:



4 Das Objektdiagramm

Das Klassendiagramm bietet eine statische Sicht auf das zu entwickelnde Softwaresystem. Die entsprechenden Klassen und ihre Beziehungen werden dargestellt. Einzelne Objekte und ihre Zustände (Attributwerte) können vom Klassendiagramm allerdings nicht angezeigt werden. Hier kommt das Objektdiagramm ins Spiel. Das **Objektdiagramm** ist eine Art **Schnappschuss** des Softwaresystems und zeigt damit konkrete Objekte und ihren aktuellen Zustand. Das Objektdiagramm ist deshalb eine sinnvolle Ergänzung des Klassendiagramms, um einen zeitlich begrenzten Zustand des Systems abzubilden.

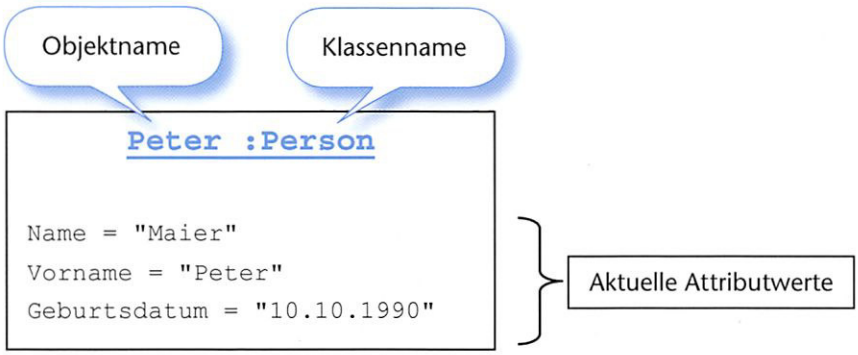
4.1 Die Darstellung eines Objektes

4.1.1 Grundlegender Aufbau

Die Darstellung eines Objektes sieht der Darstellung einer Klasse im Klassendiagramm ähnlich. Nur die Methoden werden nicht mehr angezeigt, denn es geht um einen Schnappschuss des Systems und damit um die aktuellen Attributwerte eines Objektes. Das Objekt wird konkret benannt und zusätzlich wird nach einem Doppelpunkt der zugehörige Klassenname angegeben. Der Klassenname kann entfallen, wenn es beispielsweise nur Objekte einer Klasse gibt.

Beispiel:

Das Objekt `Peter` der Klasse `Person` wird dargestellt.



Hinweis:

Bei den Attributen muss der Datentyp nicht mit angegeben werden, denn diese Informationen liegen bereits detailliert im Klassendiagramm vor. Es kann aber sein, dass ein Objektdiagramm vor einem Klassendiagramm erstellt wird, und dann könnte es sinnvoll sein, die Datentypen anzugeben (beispielsweise `Name: String = "Maier"`).

4.1.2 Klassen und Objekte gemeinsam darstellen

Neben der Darstellung einer Klasse im Klassendiagramm und einem Objekt in einem Objektdiagramm können auch beide gemeinsam dargestellt werden. Das bietet sich an, wenn zum Klassendiagramm ein charakteristischer Schnappschuss des Systems gezeigt werden soll. Das Objekt (als Instanz) der Klasse wird durch einen Pfeil und das Schlüsselwort <<instantiate>> dargestellt.



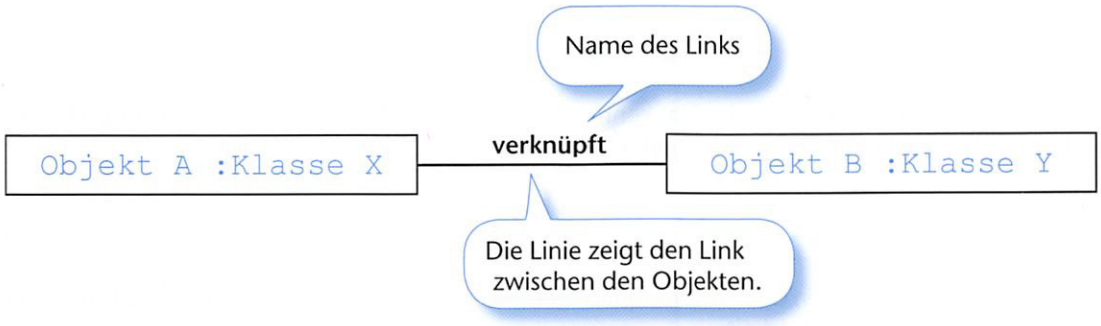
4.2 Beziehungen zwischen Objekten

Alle Beziehungen, die zwischen Klassen modelliert werden, gelten natürlich auch für die entsprechenden Objekte. Der große Unterschied zu den Klassen besteht aber darin, dass zur Zeit des

Schnappschusses immer genau ein Objekt der Klasse instanziert ist. Damit sind die Beziehungen zwischen den Objekten immer 1:1-Beziehungen – ein konkretes Objekt steht immer nur mit einem anderen konkreten Objekt in Beziehung.

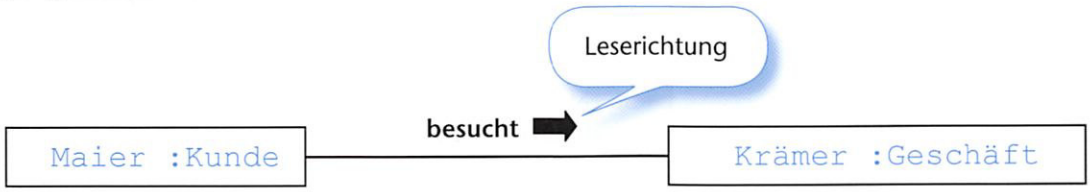
4.2.1 Der Link

Klassen können durch Assoziationen, Aggregationen und Kompositionen in Beziehung stehen. Bei den konkreten Objekten ist es einfacher – sie stehen durch einen **Link** in Beziehung. Dieser Link entspricht im Prinzip der Assoziation bei Klassen mit einer Einschränkung der Multiplizitäten – es gibt immer nur eine implizite 1:1-Multiplizität.



Leserichtung eines Links

Um einen Link näher zu spezifizieren, kann eine Leserichtung (durch einen Richtungspfeil) hinzugefügt werden.

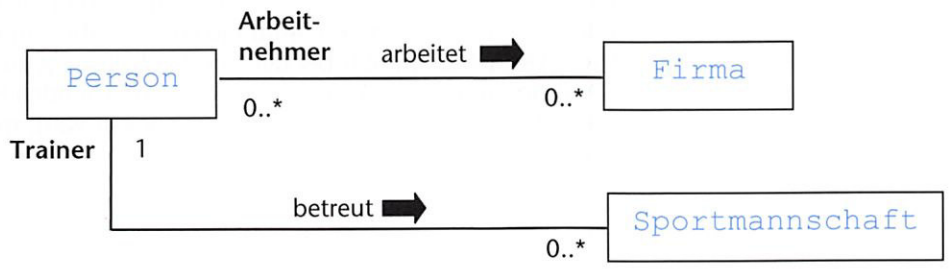


Rollen eines Links

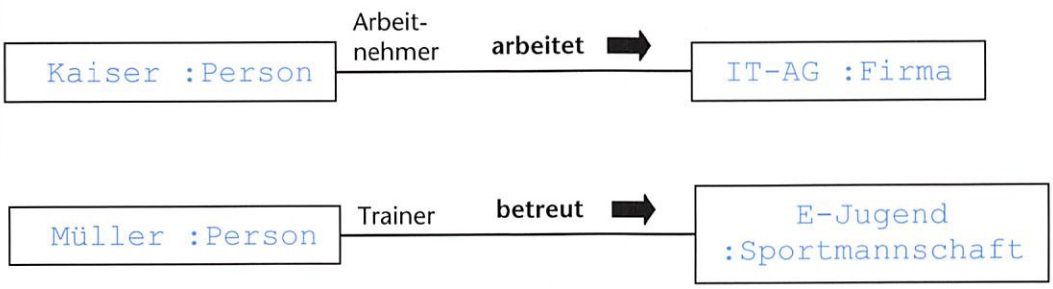
Die Assoziationen bei Klassen können durch sogenannte Rollennamen spezifiziert werden. Eine Klasse kann je nach Assoziation eine andere Rolle spielen. Das Objekt kann natürlich nur eine Rolle spielen. Das folgende Beispiel zeigt ein Klassendiagramm, in dem die Personen-Klasse verschiedene Rollen spielt. Das Objektdiagramm zeigt dann unterschiedliche Objekte mit ihren Rollen.

Beispiel:

Ein Klassendiagramm mit verschiedenen Rollen

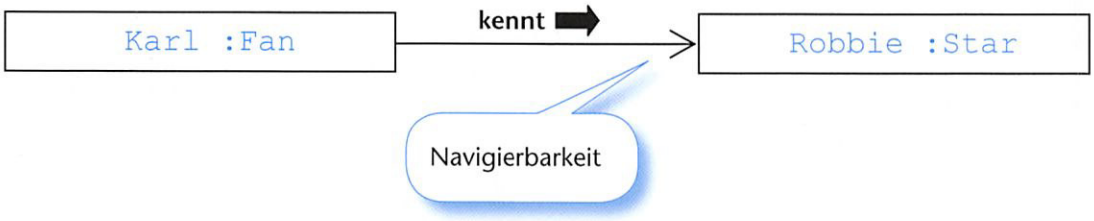


Das Objektdiagramm mit den entsprechenden Rollen:



Navigierbarkeit

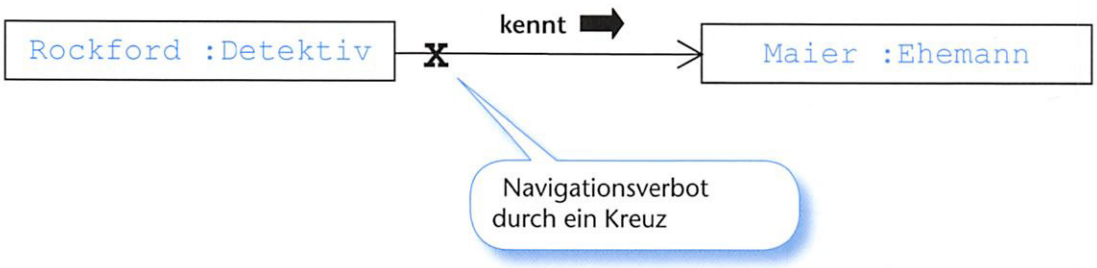
Mit Navigierbarkeit ist die Kenntnis des anderen Objektes gemeint. Man spricht dann von einem **gerichteten Link**. Die Navigierbarkeit (Kenntnis) wird durch einen Pfeil am Ende der Linklinie ausgedrückt.



Der Fan kennt seinen Star, aber der Star kennt seinen Fan in der Regel nicht. Man spricht in diesem Fall von einer **unidirektionalen Navigierbarkeit**.



Ein Kind kennt seinen Vater, der natürlich auch sein Kind kennt. Man spricht in diesem Fall von einer **bidirektionalen Navigierbarkeit**.



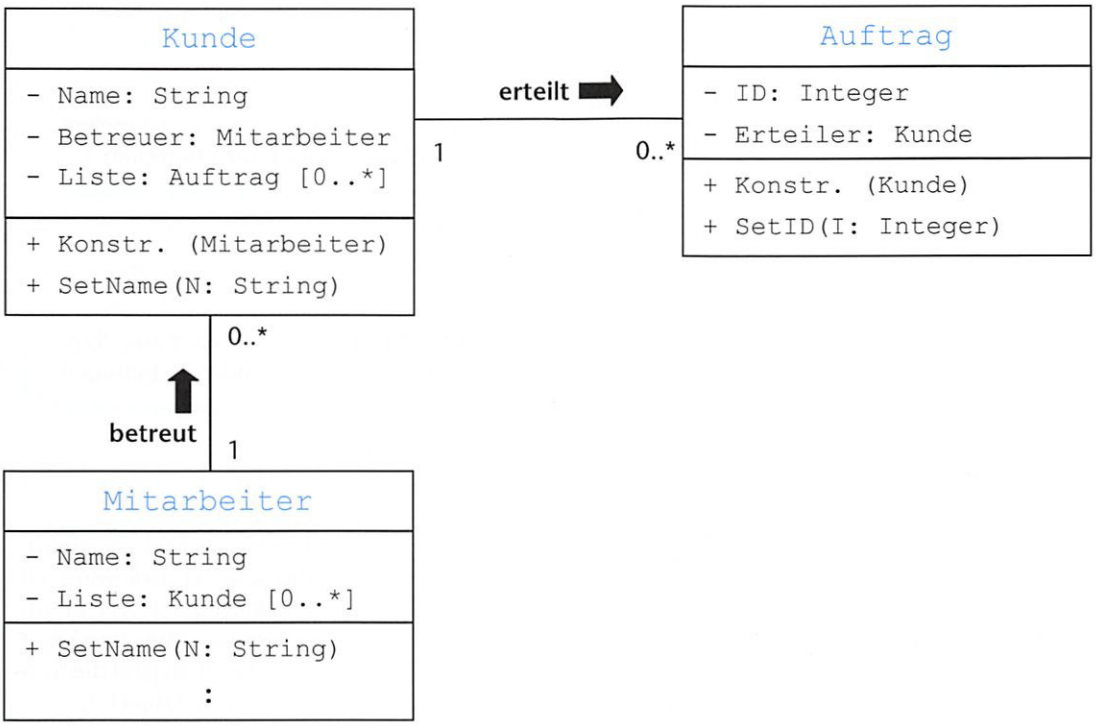
Der Detektiv kennt den Ehemann, den er im Auftrag seiner Frau beschatten soll. Der Ehemann darf aber den Detektiv nicht kennen (**Navigationsverbot**).

Hinweis:

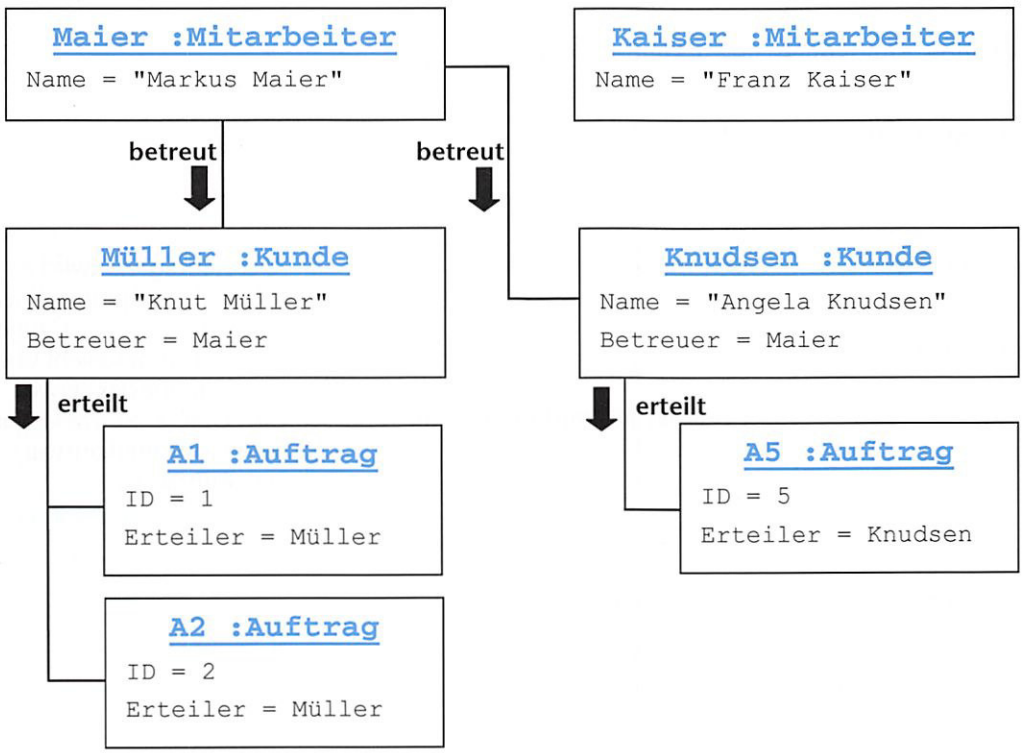
Wird weder eine Navigierbarkeit noch ein Verbot angegeben, so wird von einer **bidirektionalen** Navigierbarkeit ausgegangen. Beide Objekte kennen sich dann.

4.3 Umsetzung eines Objektdiagramms

Es soll nun beispielhaft ein Objektdiagramm in C# umgesetzt werden. Als Ausgangsbasis dient dazu das folgende Klassendiagramm: Kunden können beliebig viele Aufträge erteilen, aber ein Auftrag gehört immer zu genau einem Kunden. Die 1-Multiplizität bei der Kunden-Seite sagt zusätzlich aus, dass bei der Instanzierung eines Auftrags immer ein Verweis auf den entsprechenden Kunden stattfinden muss. Ähnlich ist es bei den Mitarbeitern. Sobald ein Kunde angelegt wird, muss ein Verweis auf einen betreuenden Mitarbeiter stattfinden.



Ein mögliches Objektdiagramm könnte dazu so aussehen:



4.3.1 Umsetzungen des Beispiels in C#

Die folgenden Umsetzungen konzentrieren sich darauf, das Objektdiagramm als einen Schnappschuss des Systems darzustellen. Die benötigten Klassen werden deshalb als implementiert vorausgesetzt (entsprechende Implementierungsbeispiele dazu sind in Kapitel 3 zu finden). Es findet nur die Darstellung einer Hauptmethode statt.

```
class Program
{

static void Main(string[] args)
{
   CMitarbeiter Kaiser = new CMitarbeiter();
   Kaiser.SetName("Franz Kaiser");

   CMitarbeiter Maier = new CMitarbeiter();
   Maier.SetName("Markus Maier");

   CKunde Mueller = new CKunde( Maier );
   Mueller.SetName("Knut Müller");
   Maier.Add( Mueller );

   CKunde Knudsen = new CKunde( Maier );
   Knudsen.SetName("Angela Knudsen");
   Maier.Add( Knudsen );

   CAuftrag A1 = new CAuftrag( Mueller );
   A1.SetID(1);
   Mueller.Add( A1 );

   CAuftrag A2 = new CAuftrag( Mueller );
   A2.SetID(2);
   Mueller.Add( A1 );

   CAuftrag A5 = new CAuftrag( Knudsen );
   A5.SetID(5);
   Knudsen.Add( A5 );
}

}
```

Der Mitarbeiter `Kaiser` wird angelegt. Er hat keinen Kunden zu betreuen.

Der Mitarbeiter `Maier` wird angelegt. Er hat noch keinen Kunden zu betreuen.

Der Kunde `Mueller` wird angelegt. Er bekommt einen Betreuer (`Maier`) zugewiesen. Das geschieht über die Übergabe des Verweises von Objekt `Maier` an den Parameterkonstruktor von Kunde `Mueller`.

Der Kunde `Müller` wird in die Liste von Betreuer `Maier` *eingetragen*.

Analog zu `Mueller`

Der Auftrag `A1` wird angelegt. Er bekommt einen Kunden (`Mueller`) zugewiesen. Das geschieht über die Übergabe des Verweises von Objekt `Mueller` an den Parameterkonstruktor von Auftrag `A1`.

Der Auftrag `A1` wird in die Liste von Kunde `Mueller` *eingetragen*.

Analog zu `A1`

Analog zu `A1`, nur mit Kunde `Knudsen`.

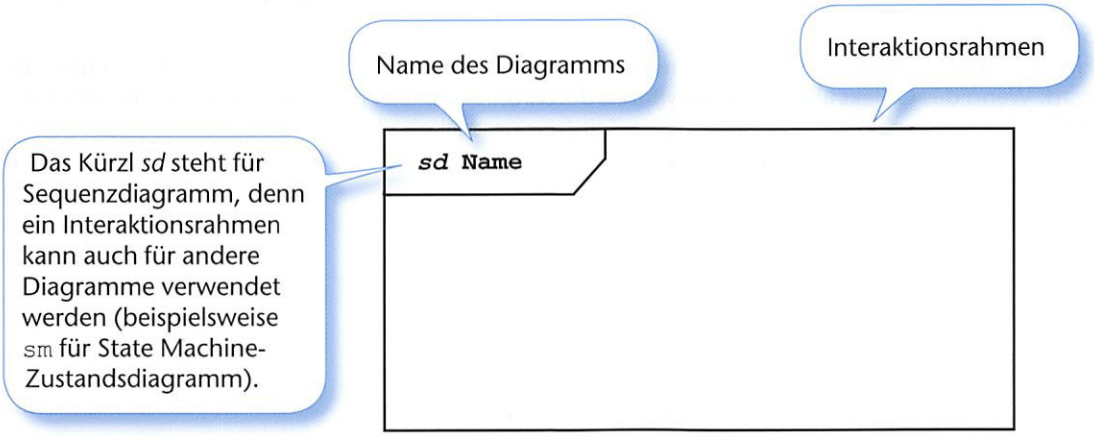
5 Das Sequenzdiagramm

Mit Sequenzdiagrammen wird die Kommunikation zwischen Objekten eines Softwaresystems dargestellt. Diese Kommunikation besteht in der Regel aus Nachrichten, die ein Objekt an ein anderes sendet. Sequenzdiagramme werden auch schon in der frühen Planungsphase eines Softwaresystems eingesetzt, um beispielsweise die Kommunikation eines Benutzers mit einer Benutzeroberfläche darzustellen. Ebenso können Sequenzdiagramme als zusätzliche Beschreibungen zu Anwendungsfällen und Klassendiagrammen dienen. Die Beschreibung eines Softwaresystems mit UML-Diagrammen ist deshalb auch ein Zusammenspiel der verschiedenen Diagrammtypen, welche sich gegenseitig ergänzen. Das statische Klassendiagramm ist eine hervorragende Basis für die Umsetzung in eine Programmiersprache, bietet jedoch wenig Informationen über dynamische Abläufe zwischen den Klassen bzw. Objekten. Diese Lücke kann das Sequenzdiagramm schließen.

5.1 Allgemeine Darstellung

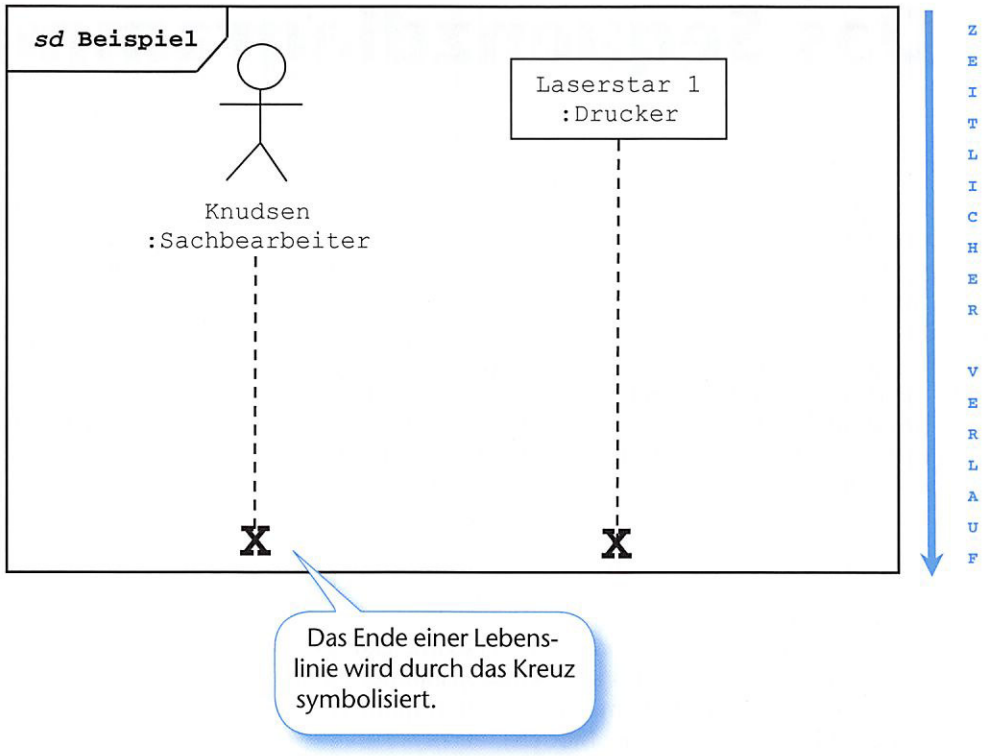
5.1.1 Der Interaktionsrahmen

Ein Sequenzdiagramm wird in der Regel innerhalb eines Interaktionsrahmen gezeichnet. Es bekommt einen eindeutigen Namen. Der Vorteil eines Interaktionsrahmens ist die Weiterverwendung in einem anderen Sequenzdiagramm, in welchem der Interaktionsrahmen einfach eingebettet wird (ähnlich dem Aufruf eines Unterprogramms oder Moduls).



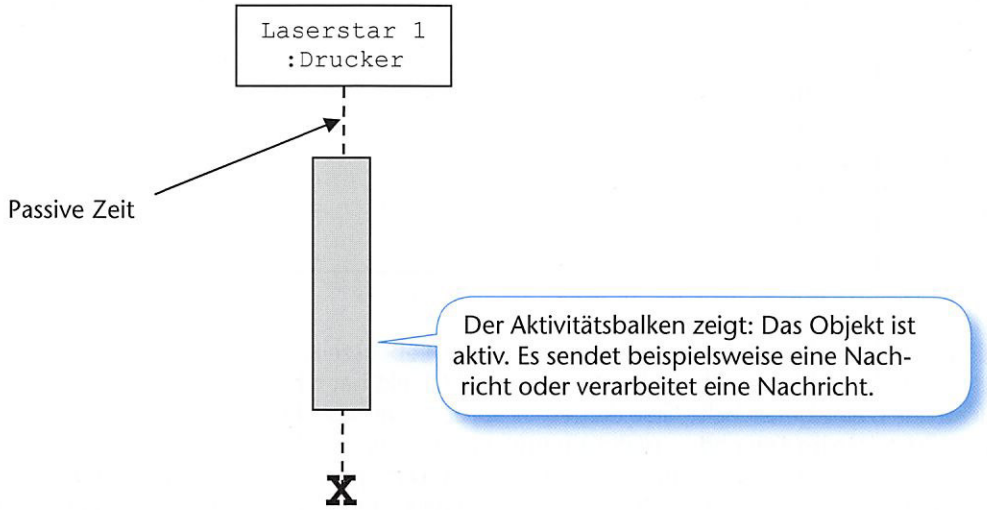
5.1.2 Lebenslinien

In einem Sequenzdiagramm werden Objekte, deren Lebenszeit und die Kommunikation untereinander in einem zeitlichen Ablauf betrachtet. Jedes Objekt hat dementsprechend eine Lebenslinie, welche durch eine gestrichelte Linie dargestellt wird. Die verschiedenen Objekte sind nebeneinander angeordnet. Wie im Anwendungsfalldiagramm werden Personen als Strichmännchen und Systeme (bzw. Objekte von Klassen) als Rechtecke dargestellt. Der Typ des Objektes (in der Regel die Klasse) wird unter der Objektbezeichnung mit einem vorangestellten Doppelpunkt angegeben. Die Gültigkeit (Lebenszeit) eines Objektes endet mit dem Stoppzeichen (einem Kreuz) am Ende der Lebenslinie.



5.1.3 Aktivitäten

Ein Objekt hat Gültigkeit, solange seine Lebenslinie nicht durch das Stoppzeichen (Kreuz) beendet wird. Wenn das Objekt allerdings Aktivitäten durchführt, dann ist das durch sogenannte Aktivitätsbalken kenntlich zu machen. Die restliche Zeit (nur gestrichelte Linie) ist sogenannte passive Zeit.



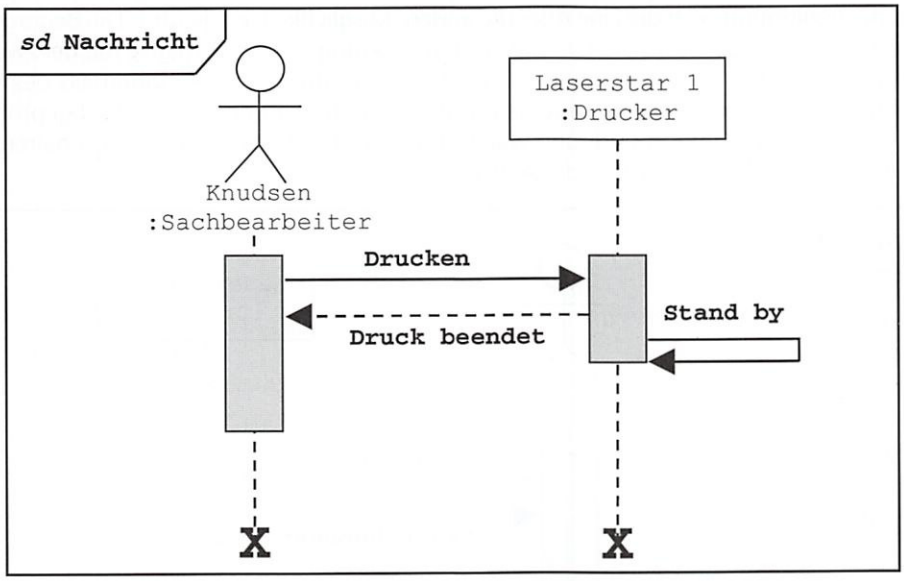
5.1.4 Nachrichten

Die Kommunikation zwischen den Objekten geschieht durch Nachrichten. Die Nachricht hat dabei eine Richtung. Das bedeutet, dass es eindeutig ist, von welchem Objekt eine Nachricht gesendet wird und wer der Empfänger der Nachricht ist. Man unterscheidet die Nachrichten weiterhin in **synchrone** und **asynchrone** Nachrichten. Bei den synchronen Nachrichten wartet der Sender auf eine Antwort, bevor er in seinen Aktionen fortfährt. Bei der asynchronen Nachricht wartet der Sender nicht auf die Antwort, sondern führt seine Aktivitäten weiter durch. Ein Objekt kann auch eine Nachricht an sich selbst senden (im Prinzip der Aufruf einer eigenen Methode).

Synchrone Nachrichten

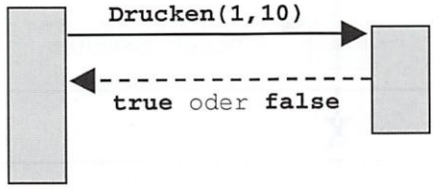
Die synchrone Nachricht wird durch eine Linie mit einer geschlossenen Pfeilspitze symbolisiert.

In dem folgenden Sequenzdiagramm sendet der Sachbearbeiter die Nachricht „Drucken" an den Laserdrucker. Der Drucker antwortet mit der Meldung „Druck beendet". Bis zu dieser Rückmeldung muss der Sachbearbeiter warten und kann keine weiteren Aktionen durchführen. Die Rückmeldung wird durch eine gestrichelte Linie mit einer geschlossenen Pfeilspitze symbolisiert. Im Anschluss sendet der Drucker sich selbst die Nachricht, in den „Stand by"-Modus zu wechseln.



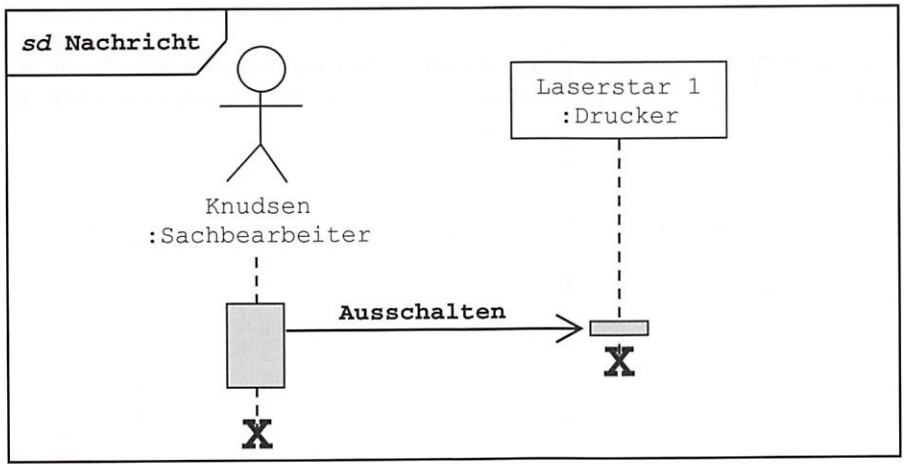
Hinweis:

Nachrichten können weiter spezifiziert werden, indem Übergabeparameter angegeben werden. Auch die Antworten können Rückgabewerte enthalten. Beispielsweise sollen die Seiten 1..10 gedruckt werden. Bei erfolgreichem Druck wird „`true`" zurückgegeben, ansonsten „`false`".



Asynchrone Nachrichten

Die asynchrone Nachricht wird durch eine Linie mit einer offenen Pfeilspitze symbolisiert. Der Sender der Nachricht wartet nicht auf eine Antwort, sondern führt seine Aktionen weiter durch.

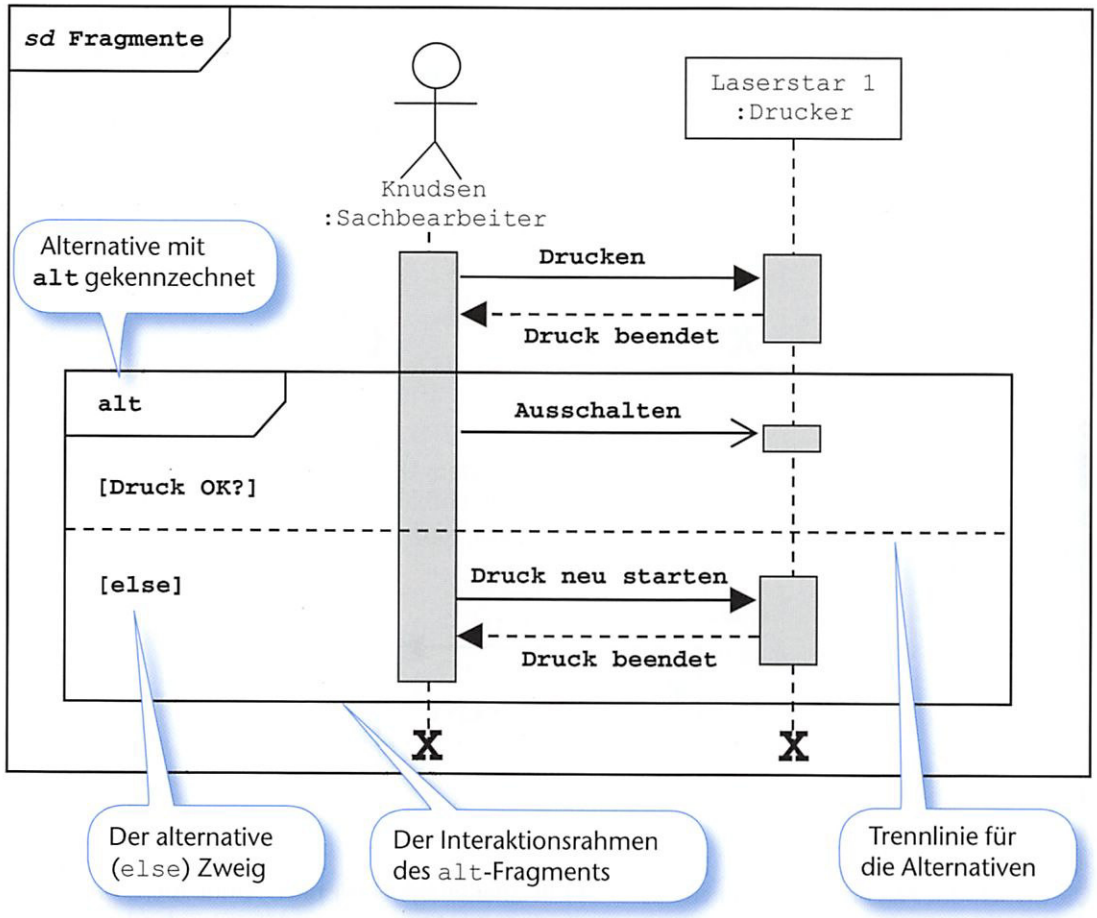


5.2 Fragmente

Innerhalb des Sequenzdiagramms können durch sogenannte Fragmente die Interaktionen zwischen den Akteuren (Objekten) zusätzlich mit Bedingungen oder Optionen versehen werden. Dazu wird ein Teil des Diagramms mit einem Interaktionsrahmen und mit einem Namen versehen. Im Folgenden werden die wichtigsten Fragmente vorgestellt.

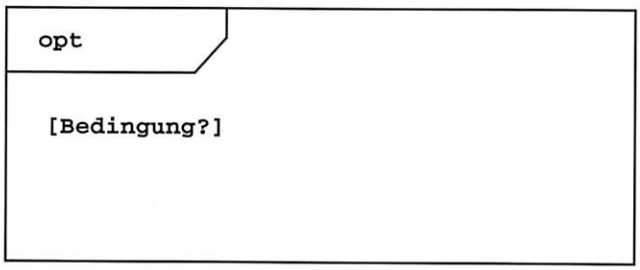
5.2.1 Alternativen

Die Alternative bietet zwei Möglichkeiten, die an eine bestimmte Bedingung geknüpft sind. Je nach Zutreffen der Bedingung wird die eine oder die andere Möglichkeit ausgeführt. Die Bedingung wird dabei in eckigen Klammern formuliert. In der Programmierung ist die Alternative mit der zweiseitigen Selektion (`if-else`-Logik) vergleichbar. Die beiden Alternativen werden innerhalb des Interaktionsrahmens durch eine gestrichelte Linie getrennt. In dem folgenden Beispiel wird geprüft, ob das Ergebnis eines Druckauftrags in Ordnung war. Falls ja, so wird die Nachricht „Ausschalten" an den Drucker gesendet. Falls nein, so wird der Auftrag wiederholt.



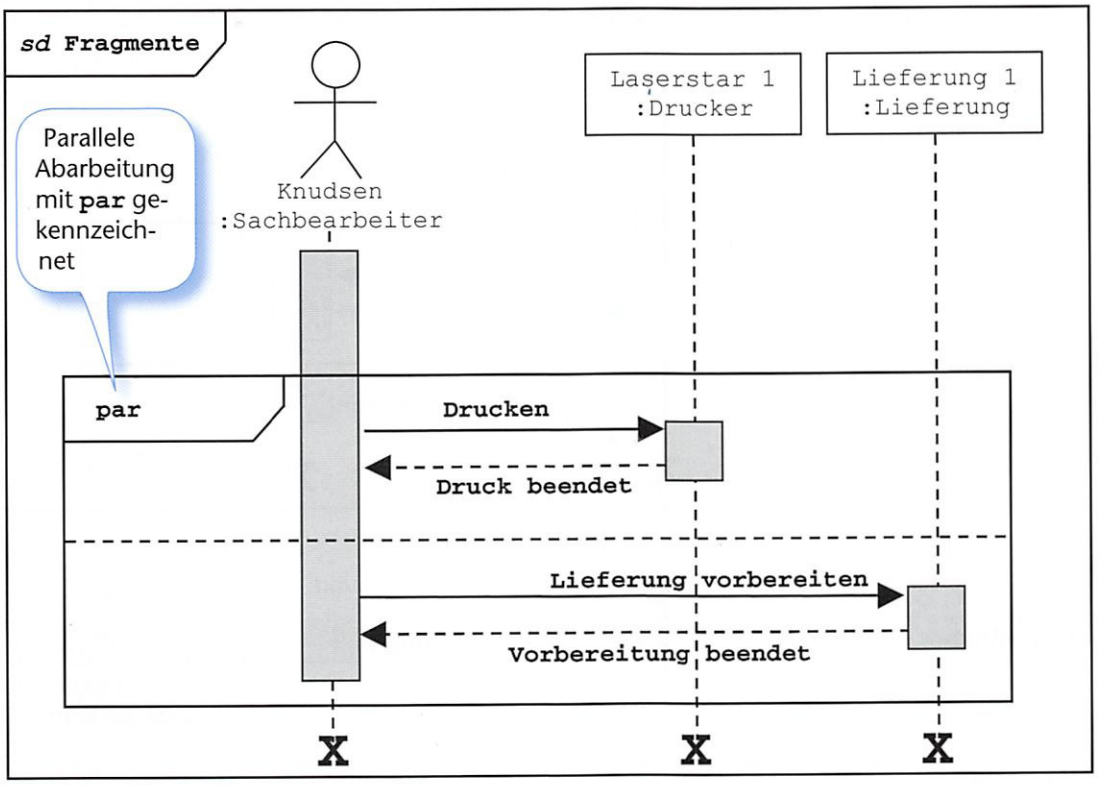
Hinweis:

In manchen Fällen ist der `else`-Zweig der Alternative nicht erforderlich. Es soll nur eine Bedingung geprüft werden und entweder kommt es zur Ausführung oder nicht. Ein solches Fragment nennt sich **Option** (gekennzeichnet mit `opt`).



5.2.2 Parallele Ausführung

Die Aktionen (Nachrichten senden) in einem Sequenzdiagramm laufen in der zeitlich festgelegten Reihenfolge hintereinander ab. In manchen Fällen ist es allerdings nicht wichtig, ob eine Nachricht vor einer anderen gesendet wird. Die Nachrichten können parallel (oder in beliebiger Reihenfolge) gesendet werden. In einem Sequenzdiagramm dient das Fragment **par** (für parallel) dazu, diese Prozesse darzustellen. In dem folgenden Beispiel sendet der Sachbearbeiter einen Druckauftrag und parallel dazu soll eine Lieferung vorbereitet werden.



Hinweis:

Das Senden der Nachricht „Drucken" bzw. „Lieferung vorbereiten" kann in beliebiger Reihenfolge oder auch parallel stattfinden. Allerdings muss die Reihenfolge innerhalb der Fragmentabschnitte eingehalten werden. Die Antwort „Druck beendet" muss natürlich nach der Nachricht „Drucken" erfolgen.

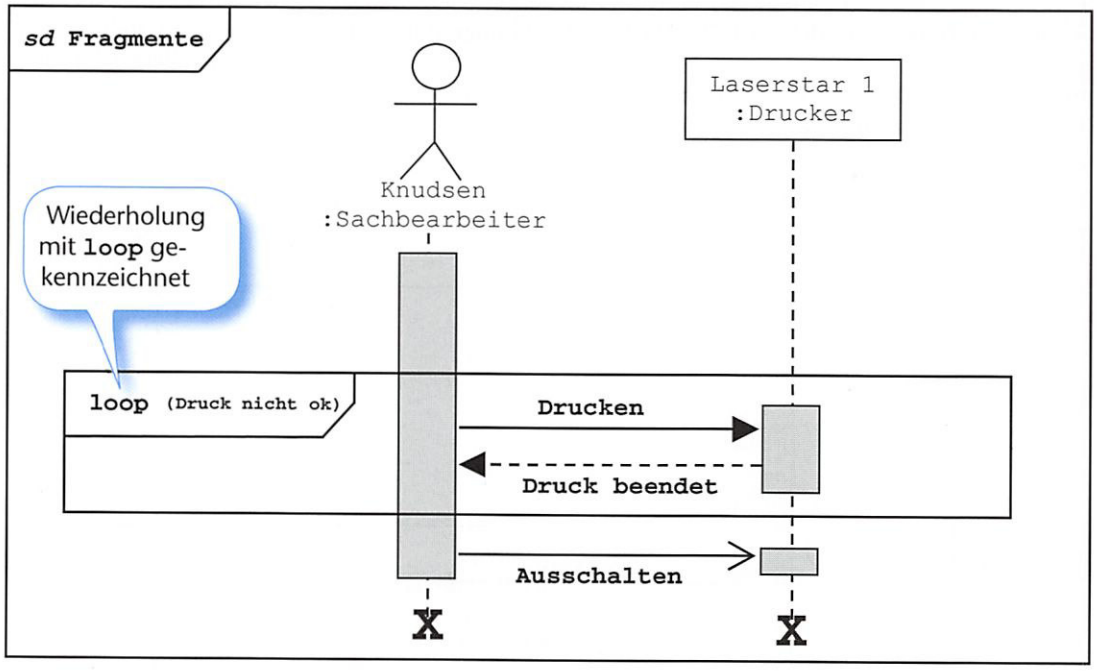
5.2.3 Schleifen

In manchen Fällen ist es sinnvoll, Abschnitte zu wiederholen (in der Programmierung durch Schleifen umgesetzt). Dazu dient das Fragment `loop` (Schleife). Dieses Fragment kann entweder eine bestimmte festgelegte Anzahl von Wiederholungen durchführen oder so lange wiederholen, bis eine Bedingung nicht mehr erfüllt ist.

Beispiele:

- `loop (*)` Beliebig viele Wiederholungen
- `loop (10)` Genau zehn Wiederholungen
- `loop (1 , 5)` Mindestens eine und maximal fünf Wiederholungen
- `loop ( Bedingung )` Solange die Bedingung wahr ist, wird wiederholt.

In dem folgenden Sequenzdiagramm wird der Druckauftrag so lange gesendet, bis das Ergebnis des Drucks zufriedenstellend ist.



5.2.4 Weitere Fragmente

Die folgende Tabelle zeigt weitere Fragmente und deren Verwendung.

Bezeichnung	Schlüsselwort	Beschreibung
Zusicherung	`assert`	Alle Nachrichten in einem `assert`-Fragment müssen unbedingt in der angegebenen Reihenfolge ablaufen.
Abbruchfragment	`break`	Dieses Fragment kann innerhalb eines anderen Fragments eingesetzt werden, um einen Abbruch zu erreichen (beispielsweise eine Schleife abbrechen).
Kritischer Bereich	`critical`	Alle Nachrichten in einem kritischen Bereich dürfen auf keinen Fall unterbrochen werden.

5.3 Umsetzung eines Sequenzdiagramms

Als Beispiel eines Sequenzdiagramms soll das oben beschriebene Diagramm mit dem `loop`-Fragment dienen. In der Umsetzung werden dazu die Klassen `CSachbearbeiter` und `CDrucker` implementiert. Der Sachbearbeiter kennt dabei den Drucker (unidirektionale Assoziation), sonst könnten keine Nachrichten gesendet werden. Das Ergebnis des Drucks wird hier der Einfachheit halber mit einem Zufallsgenerator simuliert. Der Druck ist dann zufriedenstellend, wenn bestimmte Zufallszahlen generiert werden (Simulation einer Wahrscheinlichkeit von 80 %, dass der Druck zufriedenstellend ist).

5.3.1 Umsetzung des Sequenzdiagramms in C#

```
using System;
namespace UML_IT_BERUFE
{
    class CDrucker
    {
        private bool ein = true;
        public CDrucker() { }
```

Simuliert den Einschaltzustand des Druckers

```
    public bool Drucken()
    {
        int wahrscheinlichkeit = 0;
        Random r = new Random();

        Console.WriteLine("Drucken....");
        wahrscheinlichkeit = r.Next(1, 1000);
        if (wahrscheinlichkeit > 200) return true;

        return false;
    }
    public void Ausschalten()
    {
        ein = false;
    }
}

class CSachbearbeiter
{
    private CDrucker vDrucker = null;
    public CSachbearbeiter() { }
    public void SetvDrucker(CDrucker vD)
    {
        vDrucker = vD;
    }
    public void StarteDruck()
    {
        bool ok = false;
        int anzahl = 1;

        while (ok == false)
        {
            Console.WriteLine("Druckvorgang: " + anzahl++); ;

            ok = vDrucker.Drucken();

            if (!ok) Console.WriteLine("Fehlerhafter Druck!");
        }
        Console.WriteLine();
        Console.WriteLine("Drucken beendet!\n");
    }
    public void DruckerAusschalten()
    {
        vDrucker.Ausschalten();
    }
```

Die Methode simuliert eine 80 %-Wahrscheinlichkeit, dass der Druck zufriedenstellend ist.

Die Methode antwortet mit der Nachricht `true` oder `false`.

Drucker ist ausgeschaltet.

Umsetzung der unidirektionalen Assoziation

Jetzt ist der Drucker bekannt.

Hier wird die Nachricht „Drucken" gesendet. Die Antwort des Druckers wird in der Variablen `ok` gespeichert. Die Nachrichten laufen **synchron**.

Senden der **asynchronen** Nachricht „Ausschalten"

```
    }
    class Program
    {
        static void Main(string[] args)
        {
            CDrucker LaserStar1 = new CDrucker();
            CSachbearbeiter Knudsen = new CSachbearbeiter();
            Knudsen.SetvDrucker(LaserStar1);

            Knudsen.StarteDruck();

            Knudsen.DruckerAusschalten();
        }
    }
}
```

Drucker und Sachbearbeiter anlegen

Umsetzung der Assoziation

Initiierung des Druckvorgangs

Initiierung des Ausschaltens

6 Das Aktivitäts-diagramm

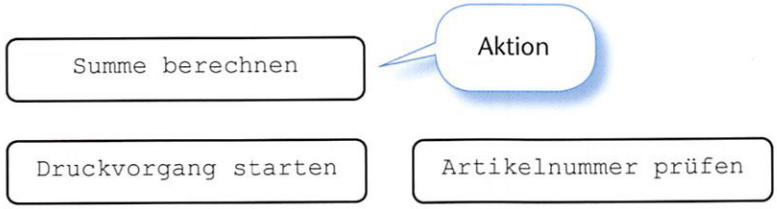
Mit Aktivitätsdiagrammen wird das Verhalten eines Softwaresystems dargestellt. Dabei geht es um Prozesse (oder auch Aktionen) und deren Reihenfolge (Ablauf). Das Aktivitätsdiagramm ähnelt deshalb auch einem **Programmablaufplan** (**PAP**), allerdings mit deutlich verfeinerten Darstellungsmöglichkeiten. Das Aktivitätsdiagramm kann in allen Phasen der Softwareentwicklung eingesetzt werden. Beispielsweise können in der Planungsphase die Geschäftsprozesse des zu entwickelnden Systems mit dem Aktivitätsdiagramm gut dargestellt werden. Das Aktivitätsdiagramm hat deshalb auch einige Gemeinsamkeiten mit den **ereignisgesteuerten Prozessketten** (**EPK**).

6.1 Allgemeine Darstellung

6.1.1 Die Aktion

Das zentrale Element des Aktivitätsdiagramms ist die Aktion. Eine Aktion ist eine Funktionalität des Systems, die atomar ist – also nicht weiter aufteilbar. Das kann die Berechnung eines Wertes oder der Aufruf einer Methode bzw. das Senden einer Nachricht sein. Die Aktion wird in ein Rechteck mit abgerundeten Ecken geschrieben.

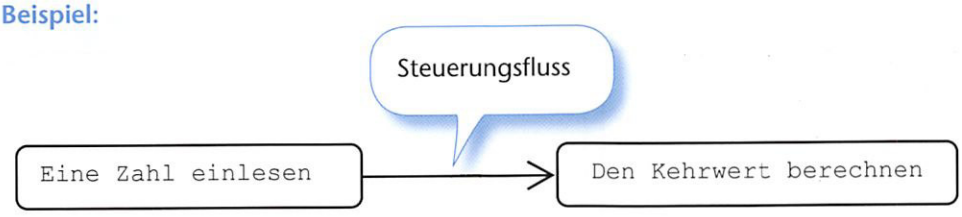
Beispiele:



6.1.2 Steuerungsfluss

Der Ablauf der einzelnen Aktionen wird durch den Steuerungsfluss gekennzeichnet. Damit wird klar vorgegeben, in welcher Reihenfolge Aktionen ausgeführt werden. Der Steuerungsfluss gilt auch für weitere Elemente wie Kontrollknoten und Objekte (siehe später). Der Fluss wird durch einen gerichteten Pfeil dargestellt.

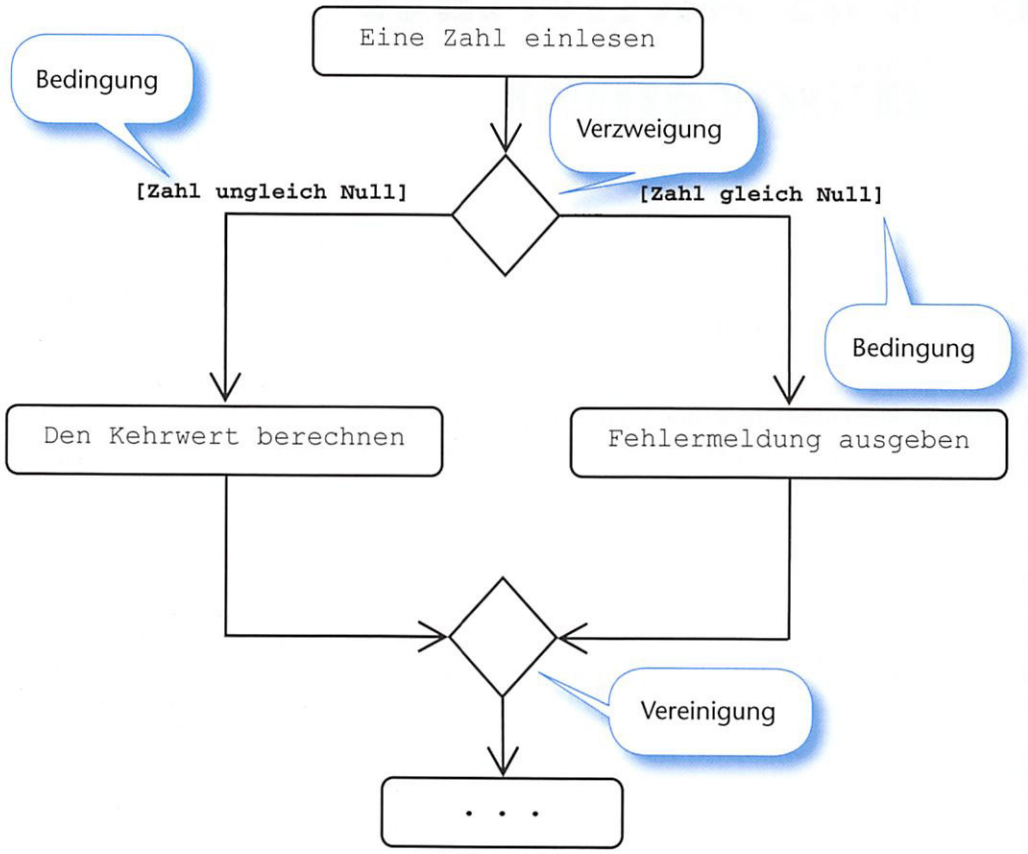
Beispiel:



6.1.3 Verzweigungen

In manchen Fällen ist es sinnvoll, dass nach einer **Aktion A** entweder die **Aktion B** oder die **Aktion C** ausgeführt wird. Der Steuerungsfluss verzweigt dann entweder in Richtung **Aktion B** oder in Richtung **Aktion C**. Die Verzweigung wird dazu mit Bedingungen (engl. *guards*) versehen, die den Fluss steuern. Die Verzweigung wird durch eine Raute gekennzeichnet (so wie im Programmablaufplan PAP). Ebenso kann die Raute aber auch genutzt werden, um die verzweigenden Flüsse wieder zu einem Steuerungsfluss zu vereinigen.

Beispiel:

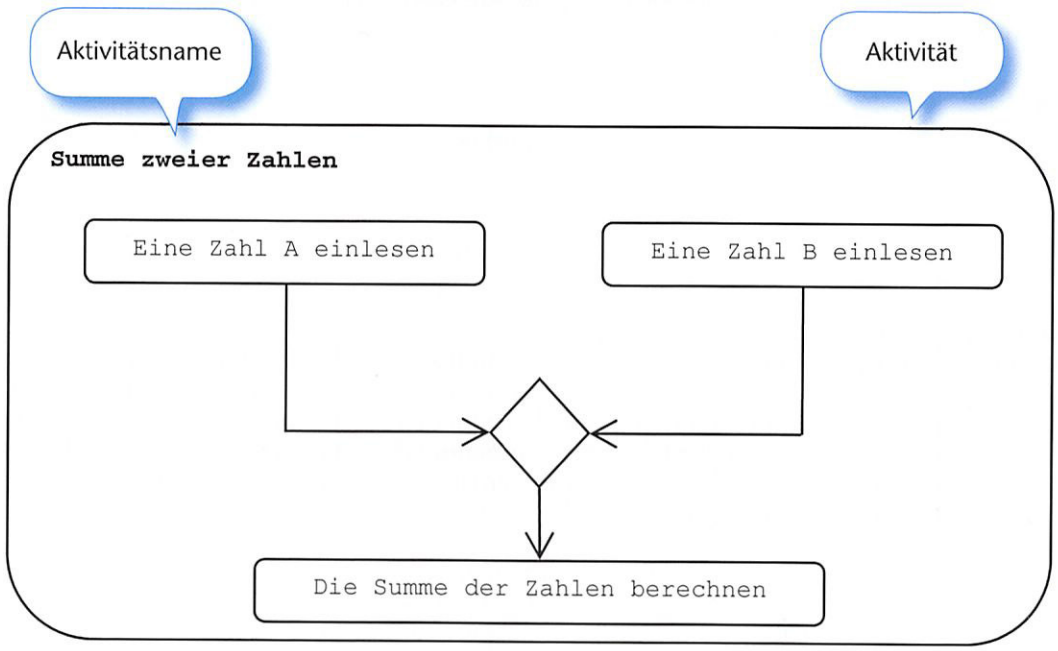


Hinweis:

Von einer Verzweigung können auch mehr als zwei Steuerungsflüsse ausgehen bzw. es können auch mehr als zwei Steuerungsflüsse vereinigt werden.

6.1.4 Aktivitäten

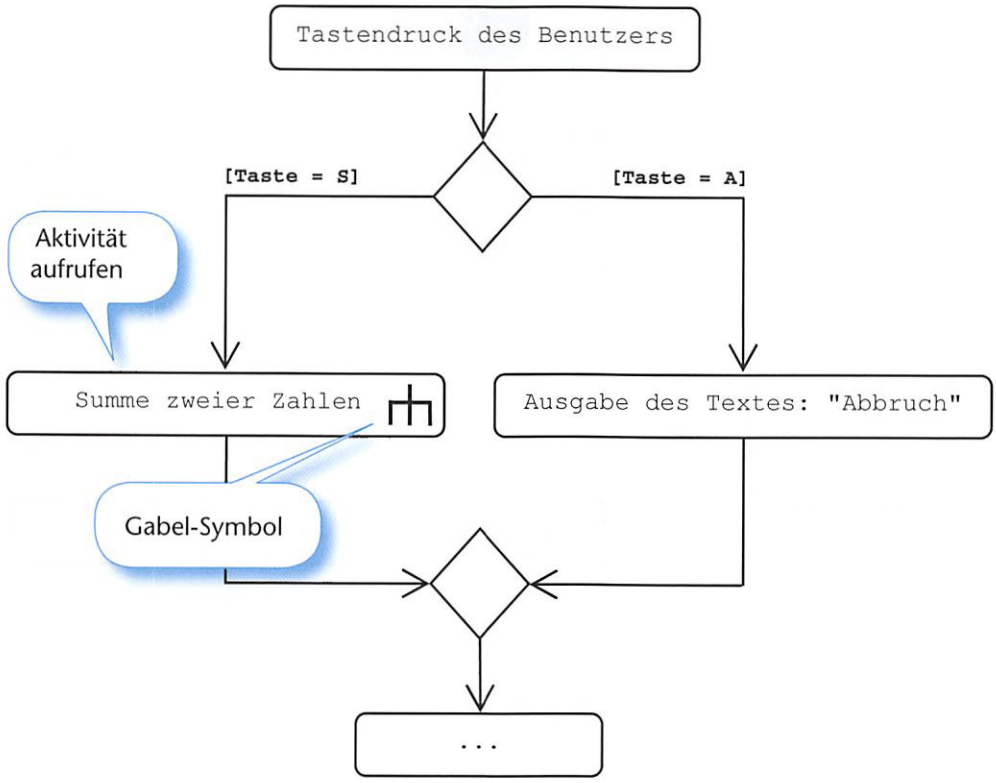
Eine Aktivität ist eine Einheit, die mehrere Aktionen und deren Steuerungsflüsse zusammenfasst. Die Aktivität hat einen eindeutigen Namen und wird durch ein Rechteck mit abgerundeten Ecken eingerahmt, beispielsweise die Summenberechnung zweier Zahlen:



Eine Aktivität kann (wie ein Unterprogramm) in einem anderen Aktivitätsdiagramm oder an einer anderen Stelle desselben Aktivitätsdiagramms aufgerufen werden. Dazu werden der Name der Aktivität und ein besonderes Symbol (**Gabel-Symbol**) angegeben.

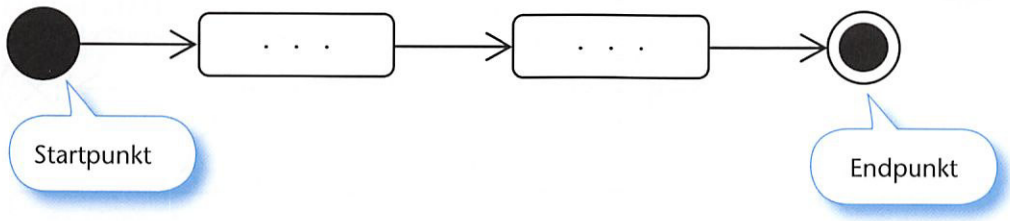
Beispiel:

Die Aktivität „Summe zweier Zahlen" wird in einem anderen Aktivitätsdiagramm genutzt. Ein Benutzer drückt auf Buchstaben der Tastatur und wählt damit verschiedene Funktionalitäten.



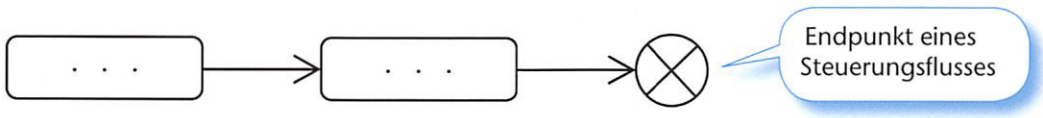
6.1.5 Start- und Endpunkte

Jede Aktivität hat mindestens einen Startpunkt und mindestens einen Endpunkt. Damit wird der Steuerungsfluss eindeutig festgelegt. In der Regel hat eine Aktivität bzw. ein Aktivitätsdiagramm einen Startpunkt und einen Endpunkt. Der Startpunkt wird durch einen ausgefüllten Kreis symbolisiert, der Endpunkt durch einen Kreis, in dem ein kleinerer ausgefüllter Kreis liegt. Wird der Endpunkt durch den Steuerungsfluss erreicht, so ist die Aktivität damit beendet.



Steuerungsfluss-Endpunkt

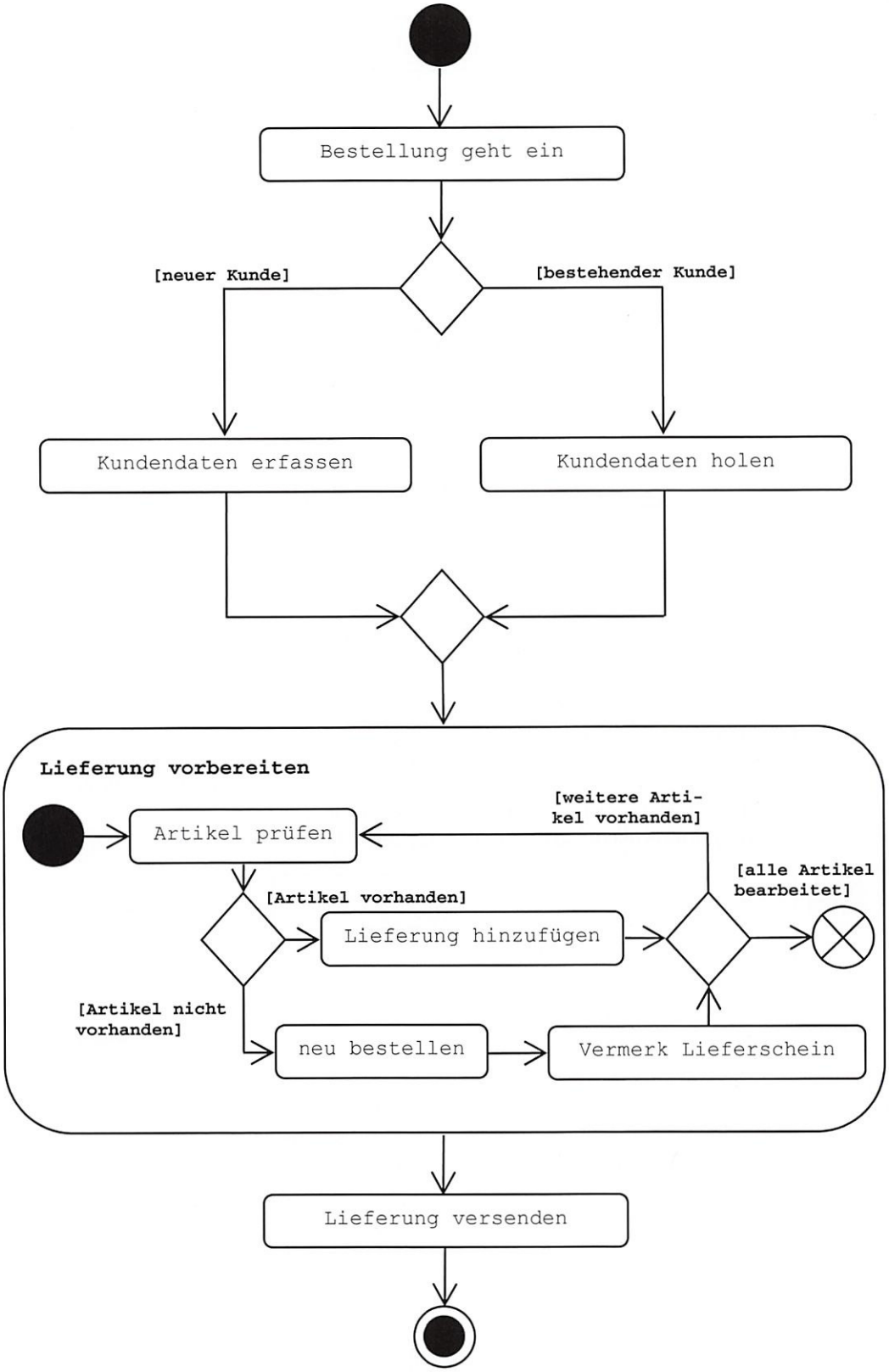
Neben dem Endpunkt für die Aktivität gibt es auch noch einen Endpunkt für Steuerungsflüsse. Damit werden abzweigende Steuerungsflüsse beendet (aber nicht die Aktivität). Das Symbol für diesen Endpunkt ist ein Kreis mit einem einbeschriebenen Kreuz.



Beispiel:

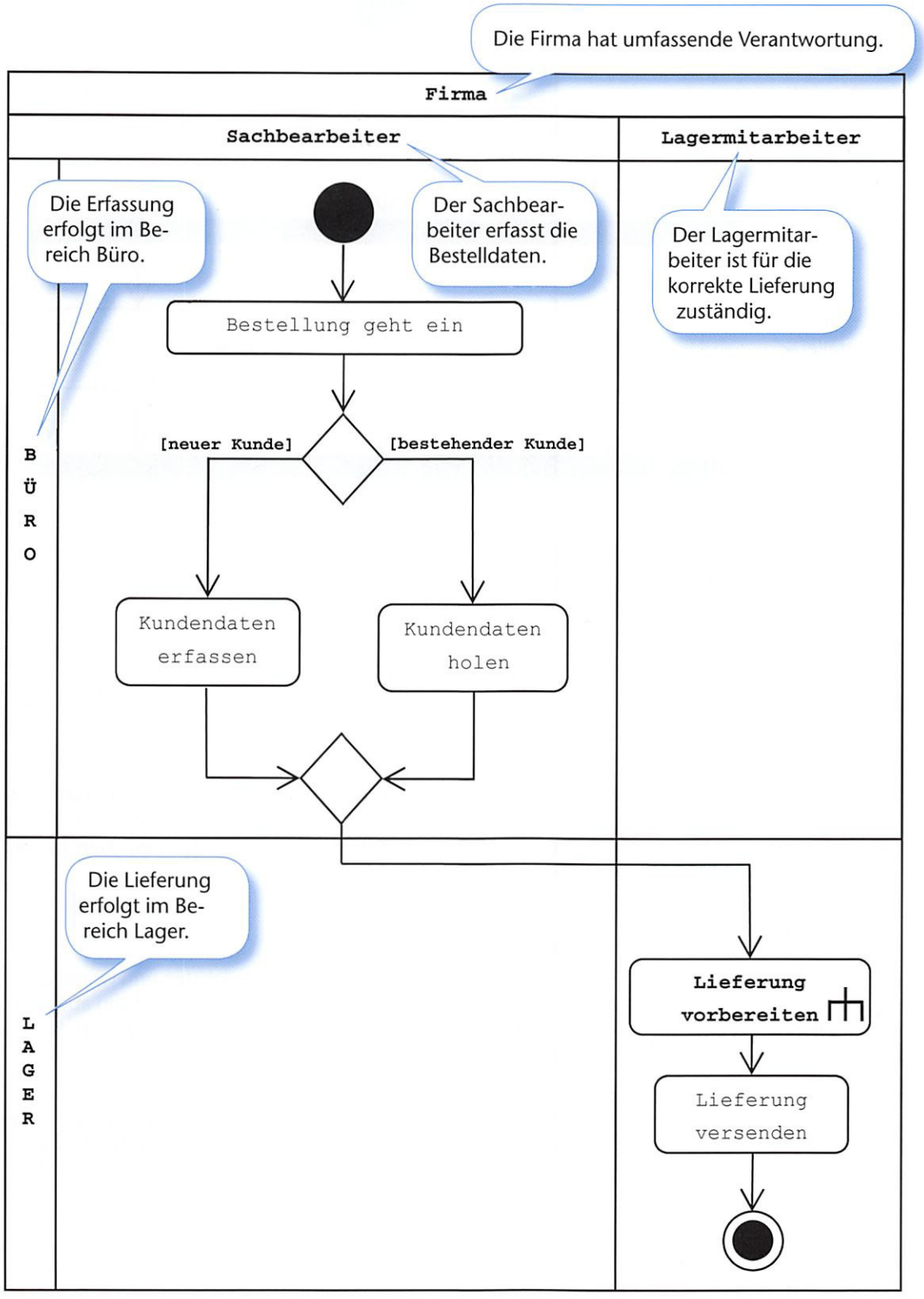
Das folgende Beispiel zeigt die Verwendung von mehreren Start- und Endpunkten in einem Aktivitätsdiagramm. Dabei wird ein Bestellvorgang beschrieben, bei dem einem bestehenden oder neuen Kunden die bestellten Artikel geliefert werden. Wenn ein Artikel nicht vorrätig ist, dann wird er bestellt und der Kunde erhält einen Vermerk auf dem Lieferschein, dass der Artikel nachgesendet wird. Dieser Vorgang findet in einer eigenen Aktivität (Lieferung vorbereiten)

statt. Der Steuerungsfluss endet in dieser Aktivität mit einem *Steuerungsfluss-Endpunkt*, wenn alle Artikel der Bestellung geprüft wurden. Danach übernimmt der Hauptsteuerungsfluss wieder die Kontrolle und endet nach der Versendung der Lieferung mit einem Aktivitäts-Endpunkt.



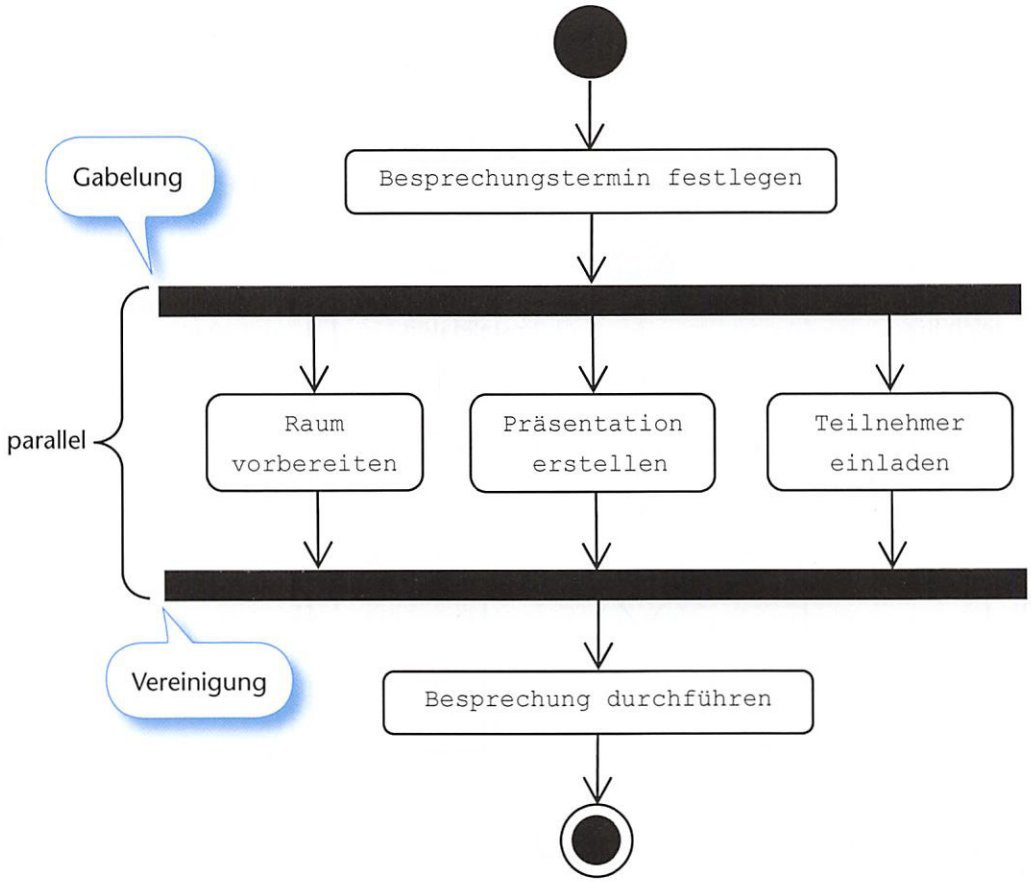
6.1.6 Verantwortungsbereiche

In einem Aktivitätsdiagramm kann es verschiedene Verantwortungsbereiche geben. In dem obigen Beispiel mit der Bestellung gibt es Sachbearbeiter, die die Kundendaten erfassen, und wiederum Arbeiter, die die Lieferung zusammenstellen. Die Mitarbeiter haben dabei verschiedene Verantwortungsbereiche. Das Diagramm wird dabei in vertikale Einheiten unterteilt, die diese Verantwortungsbereiche begrenzen. Die Einteilung erfolgt durch senkrechte Linien. Jeder Verantwortungsbereich erhält einen eindeutigen Namen. Neben einer vertikalen Einteilung kann zusätzlich eine horizontale Einteilung erfolgen, die weitere Verantwortungsbereiche kennzeichnet. Dabei überschneiden sich natürlich die einzelnen Verantwortungsbereiche. In dem folgenden Diagramm wird das Beispiel mit der Bestellung in verschiedene Verantwortungsbereiche eingeteilt.



6.1.7 Gabelungen und Vereinigungen

Der Steuerungsfluss kann durch Verzweigungen in verschiedene Richtungen gelenkt werden. Dabei wird allerdings durch die Bedingungen sichergestellt, dass der Fluss immer nur eine Richtung nimmt. Möchte man hingegen, dass der Fluss in mehrere Teilflüsse aufgeteilt wird, die dann auch parallel abgearbeitet werden, wird das in einem Aktivitätsdiagramm durch eine Gabelung ausgedrückt. Die Gabelung wird durch einen waagerechten Balken symbolisiert, von dem die weiteren Steuerungsflüsse parallel starten. Die parallelen Flüsse müssen auch wieder zusammenlaufen. Das geschieht mit einer Vereinigung, die ebenfalls durch einen waagerechten Balken symbolisiert wird. Das folgende Beispiel zeigt die Planung einer Besprechung, die organisiert werden muss.



Hinweis:

Eine echte parallele Abarbeitung findet in einem Softwaresystem in der Regel nicht statt, es sei denn, dass die Prozesse auf verschiedenen Prozessoren laufen. Das parallele Abarbeiten kann auf einem einfachen Rechner mithilfe von *Threads* simuliert werden. In den einzelnen *Threads* werden dann die parallelen Steuerungsflüsse abgearbeitet.

6.2 Besondere Kommunikation

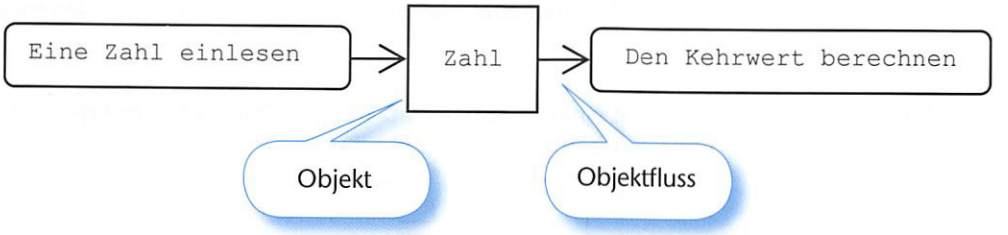
Der Steuerungsfluss lief bislang von einer Aktion (bzw. Aktivität) zu einer nächsten und wurde durch Verzweigungen oder Gabelungen entsprechend weitergeleitet. Der Steuerungsfluss kann aber noch weiter differenziert werden. Es können Objekte eingebunden und Signale versendet werden. Ebenso können Unterbrechungen den Steuerungsfluss kontrolliert abbrechen.

6.2.1 Objekte und Objektfluss

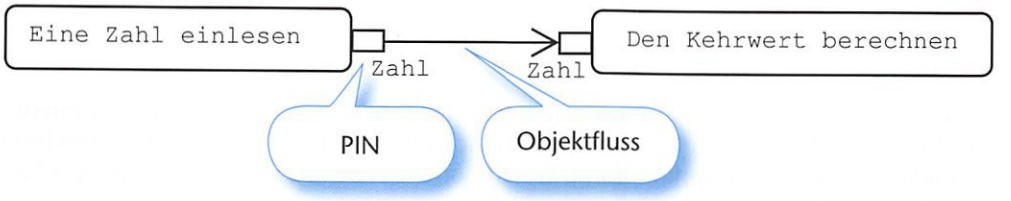
Zwischen zwei Aktionen (oder auch Aktivitäten) kann es zu einem Datenaustausch in Form eines Objektes kommen. Ein solches Objekt wird einfach in den Steuerungsfluss eingebettet. Es wird in einem Rechteck (ohne abgerundete Ecken) dargestellt.

Beispiel:

Das Objekt `Zahl` wird in den Steuerungsfluss eingebunden:



Alternativ kann diese Darstellung auch mit der sogenannten **PIN-Notation** erfolgen:

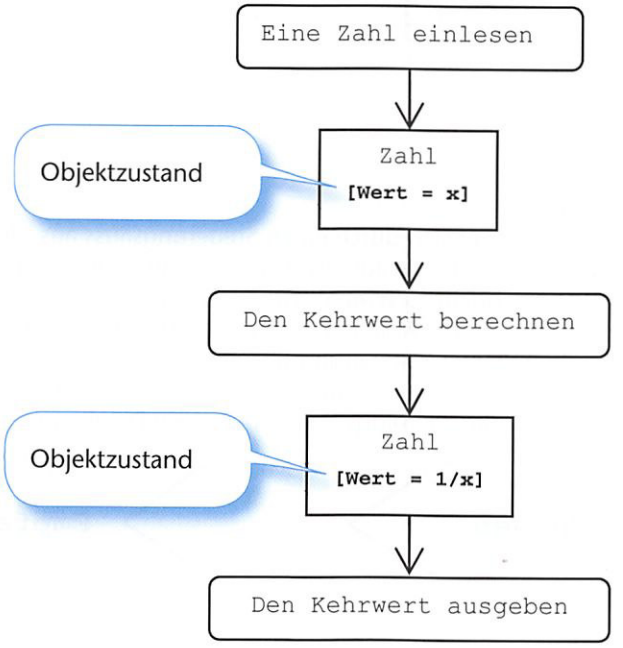


Hinweis:

Unter Objekten sind natürlich Instanzen von Klassen zu verstehen. Unter Objektfluss kann man sich beispielsweise die Übergabe eines Objektes an eine Methode (in Form eines Parameters) vorstellen. In dem obigen Beispiel könnte die Aktion „Eine Zahl einlesen" beispielsweise ein Objekt `Zahl` erzeugen und an eine Methode `Kehrwert` übergeben, die die Aktion „Den Kehrwert berechnen" umsetzt.

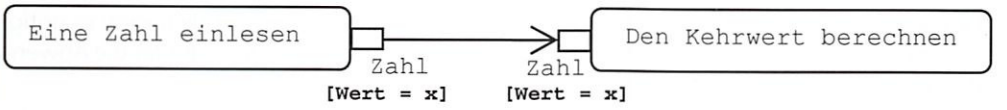
Zustände von Objekten

Die Objekte können im Laufe des Objektflusses ihren Zustand ändern (die Attributwerte ändern). Solche Änderungen können im Diagramm durch Zustandsangaben gekennzeichnet werden:



Hinweis:

In der PIN-Notation wird der Zustand unter dem Objektnamen angegeben:

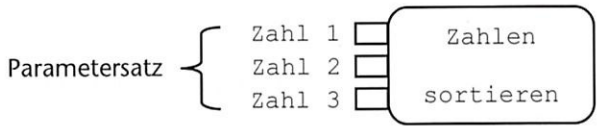


Parametersätze

Manchmal ist nicht nur ein Objekt an einer Aktion beteiligt. Es können beliebig viele Objekte mithilfe der PIN-Notation an eine Aktion (oder Aktivität) übergeben werden. Mehrere dieser Objekte werden als Parametersatz bezeichnet. Das bedeutet, dass alle Objekte übergeben werden müssen.

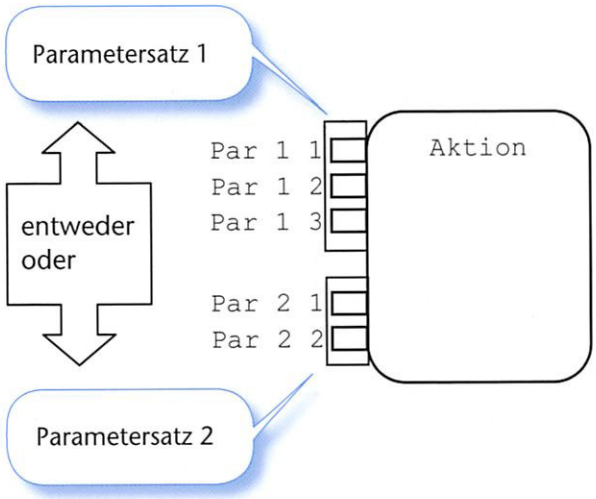
Beispiel:

Die Aktion „Zahlen sortieren" hat einen Parametersatz mit drei Objekten vom Typ `Zahl`. Der Aktion **müssen** alle drei Objekte übergeben werden.



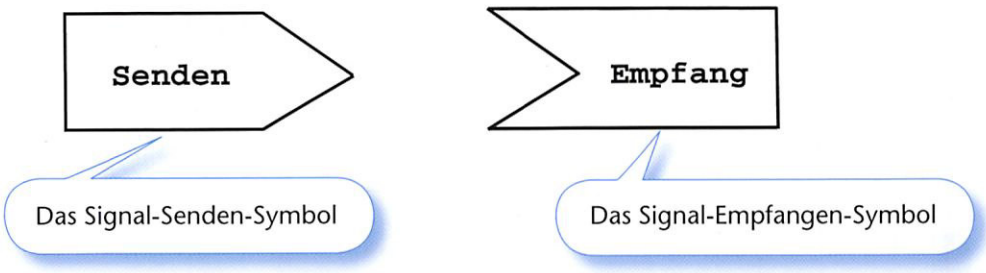
Hinweis:

Es können sogar mehrere Parametersätze übergeben werden. Die einzelnen Parametersätze müssen dann durch einen Rahmen gekennzeichnet werden. Bei mehreren Parametersätzen gilt eine Exklusive-Oder-Verknüpfung. Es darf nur entweder der eine oder der andere Parametersatz übergeben werden.



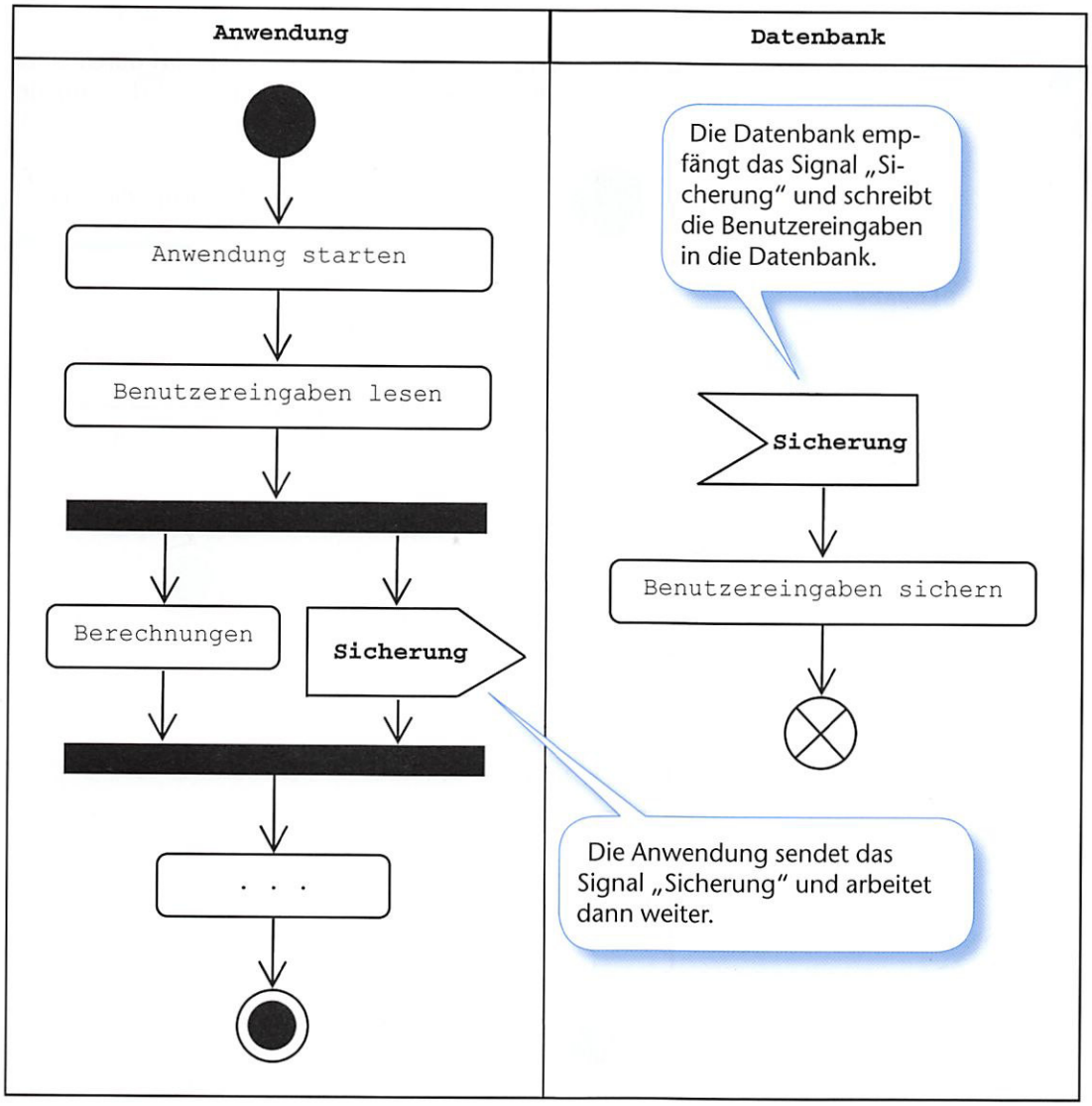
6.2.2 Signale senden

Das Aktivitätsdiagramm zeichnet sich durch einen Steuerungsfluss aus, der verzweigen oder auch durch eine Gabelung in parallele Flüsse aufgeteilt werden kann. Allerdings ist es bislang nicht möglich, eine Art asynchrone Nachricht zu senden (wie im Sequenzdiagramm), sodass der Steuerungsfluss unabhängig von der Nachricht weiter läuft. Aus diesem Grund gibt es das Signal-Senden und das Signal-Empfangen. Innerhalb des Steuerungsflusses wird ein Signal (Nachricht) gesendet. Dieses Signal wird empfangen und initiiert einen anderen Steuerungsfluss. Ein Signal kann an beliebigen Stellen gesendet werden, es darf allerdings immer nur einen Empfänger geben.



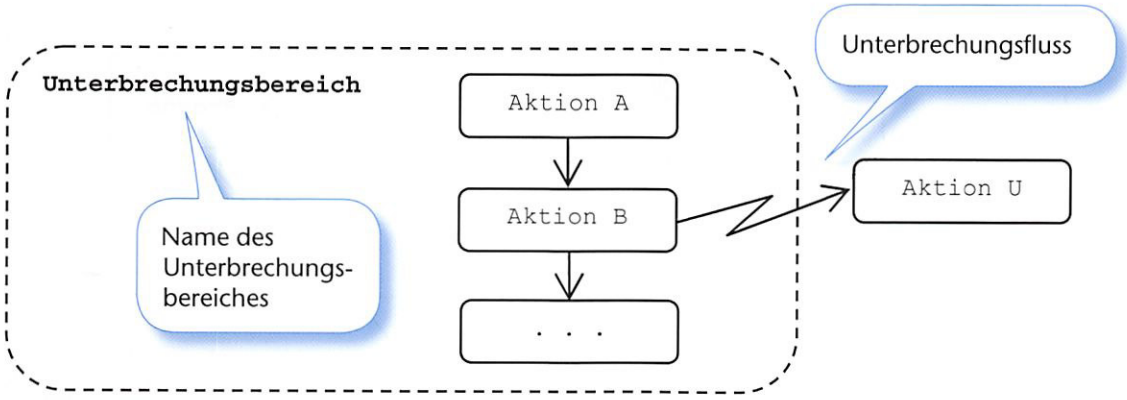
Beispiel:

Eine Anwendung arbeit mit einer Datenbank zusammen. In manchen Fällen kann es nötig sein, dass eine Datensicherung erfolgt, um Datenverlust vorzubeugen. Dazu sendet die Anwendung das Signal „Sicherung".



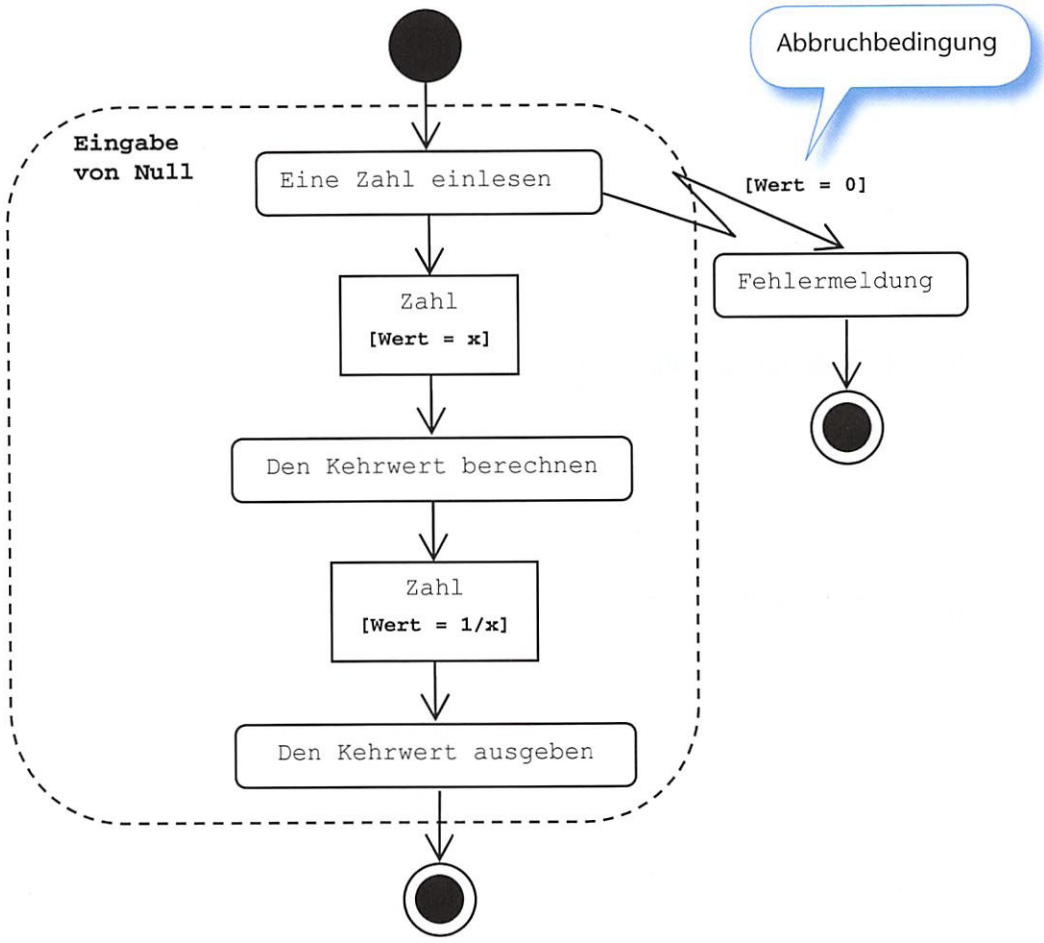
6.2.3 Unterbrechungen

In manchen Fällen kann es nötig sein, dass ein Steuerungsfluss kontrolliert abgebrochen wird – beispielsweise, wenn bestimmte Zustände (oder ein Fehler) eintreten. In vielen Programmiersprachen (C++, C# und auch Java) wird dieser kontrollierte Abbruch durch die Ausnahmebehandlung (*Exception-Handling*) geregelt. In einem Aktivitätsdiagramm werden dazu ein sogenannter Unterbrechungsbereich und ein Unterbrechungsfluss festgelegt. Der Unterbrechungsbereich wird wie eine Aktivität gekennzeichnet, aber mit gestrichelten Linien.



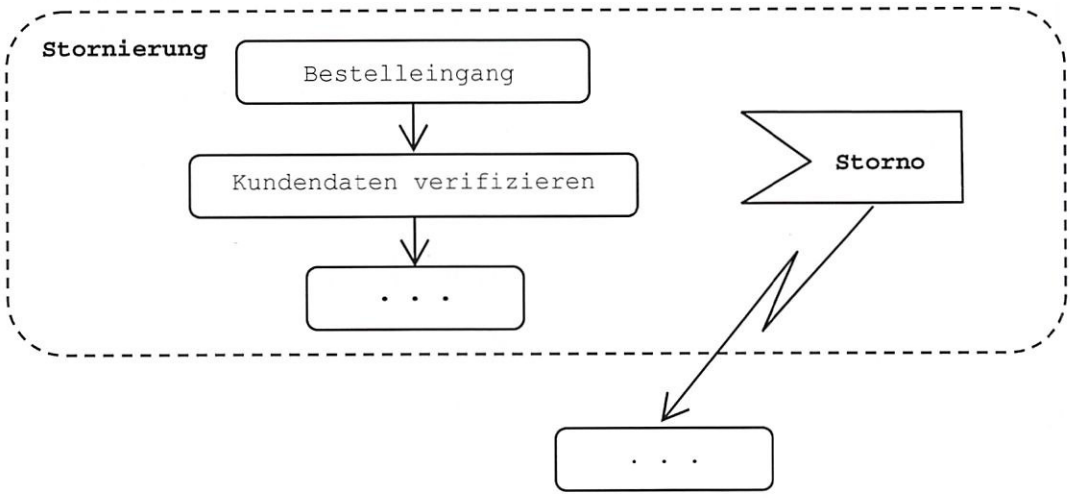
Beispiel:

Das bereits bekannte Beispiel zur Berechnung des Kehrwertes einer Zahl wird durch einen Unterbrechungsbereich ergänzt, der dann den Steuerungsfluss abbricht, wenn die eingegebene Zahl null ist.



Hinweis:

Oftmals ist es sinnvoll, dass ein Unterbrechungsbereich einen Signal-Empfang hat, der den Abbruch auslösen kann. Damit können kontrollierte Abbrüche über Signale gesteuert werden. In dem folgenden Beispiel wird die Bearbeitung einer Bestellung überflüssig, wenn der Kunde eine Stornierung verlangt.

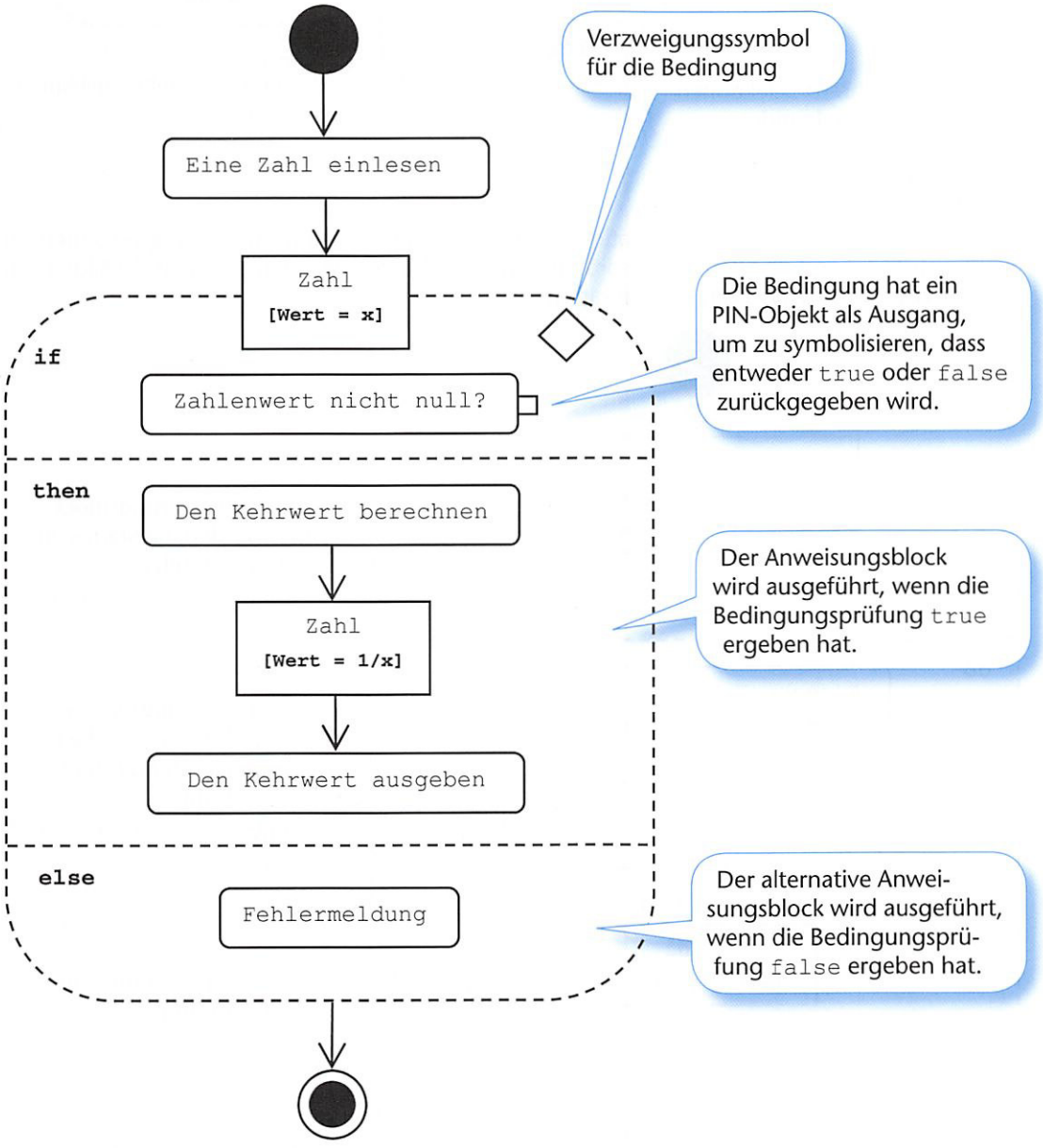


6.3 Selektion und Iteration

Mithilfe der Verzweigungen wurden bereits Selektion und Iteration (siehe Beispiel „Lieferung vorbereiten") umgesetzt. Allerdings bietet das Aktivitätsdiagramm weitere Möglichkeiten, solche Vorgänge zu symbolisieren. Mit einer kompakten Ausdrucksweise sind sowohl Selektion als auch Iteration *programmiersprachennäher* darstellbar.

6.3.1 Die Selektion

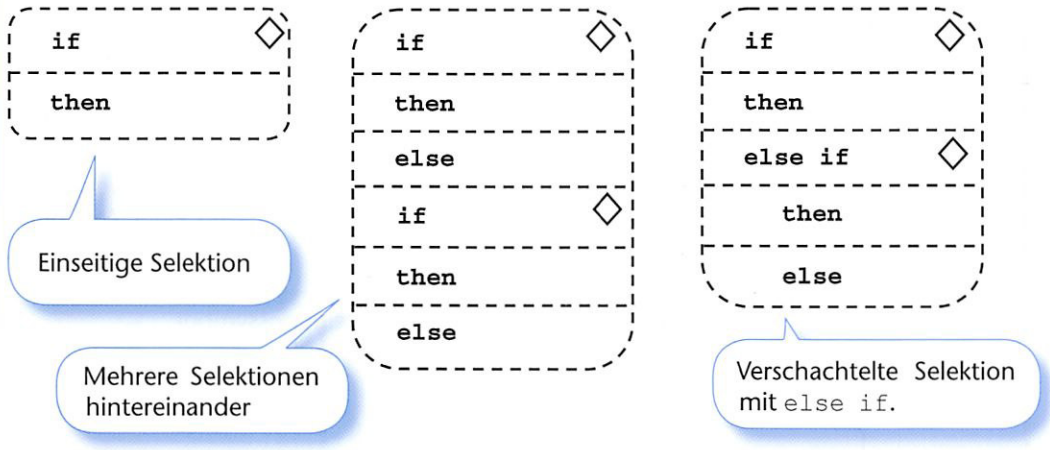
Die Selektion wird in drei Bereiche eingeteilt: die Bedingungsprüfung (**`if`**), die Anweisung (**`then`**) und die Alternative (**`else`**). Das folgende Beispiel berechnet den Kehrwert einer Zahl nur, wenn die Zahl ungleich null ist.



Hinweise:

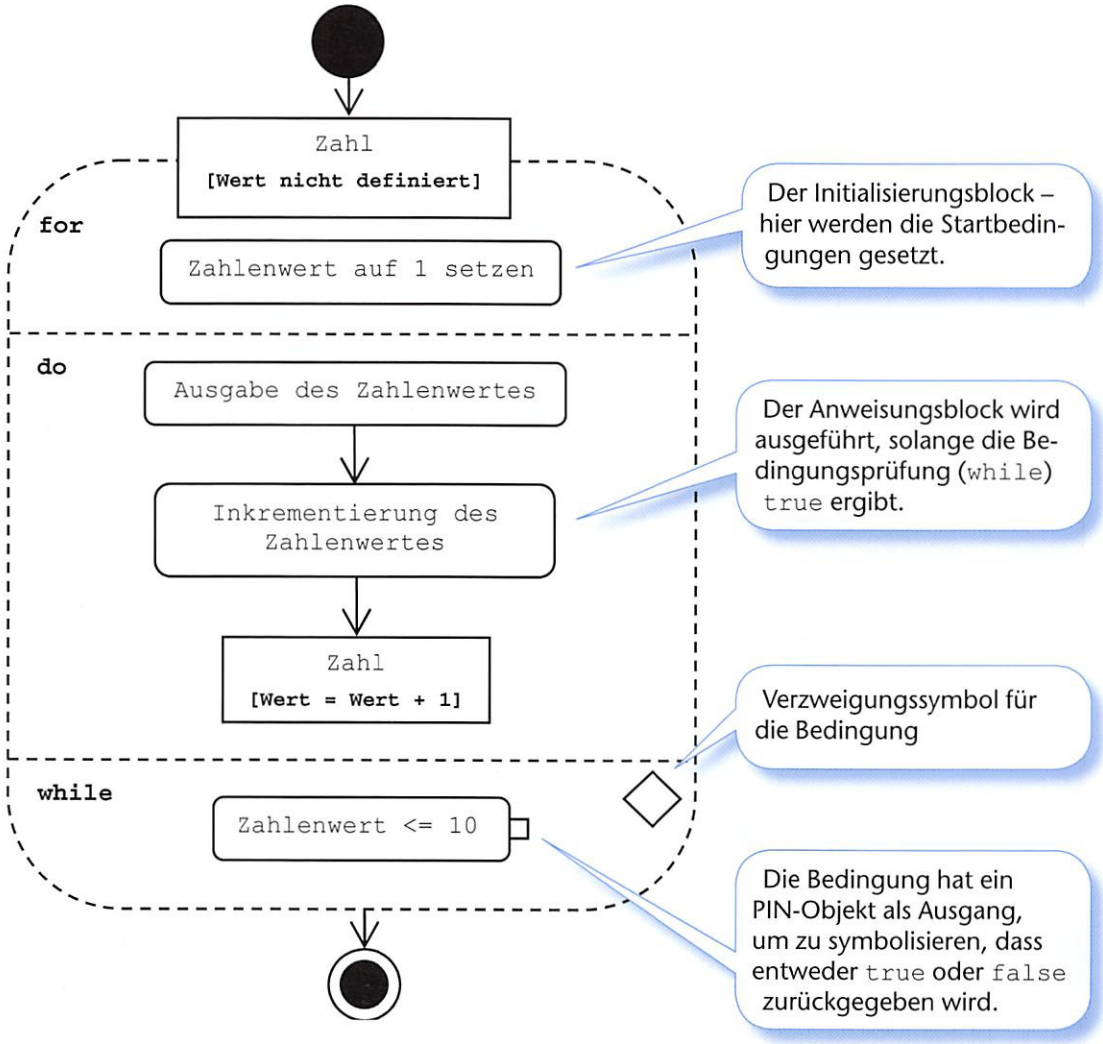
- Die Selektion kann auch ohne `else`-Zweig geschrieben werden. Dann spricht man von einer einseitigen Selektion.
- Mehrere Selektionen können auch hintereinandergeschrieben werden.
- Selektionen können auch verschachtelt werden (**`else if`**).

Beispiele:



6.3.2 Die Iteration

Ähnlich der Selektion ist auch die Iteration aufgebaut. Es gibt eine Initialisierung (**for**), einen Anweisungsteil (**do**) und eine Bedingungsprüfung (**while**). Das folgende Beispiel zeigt die Bildschirmausgabe der Zahlen 1 bis 10 in einer Iteration.

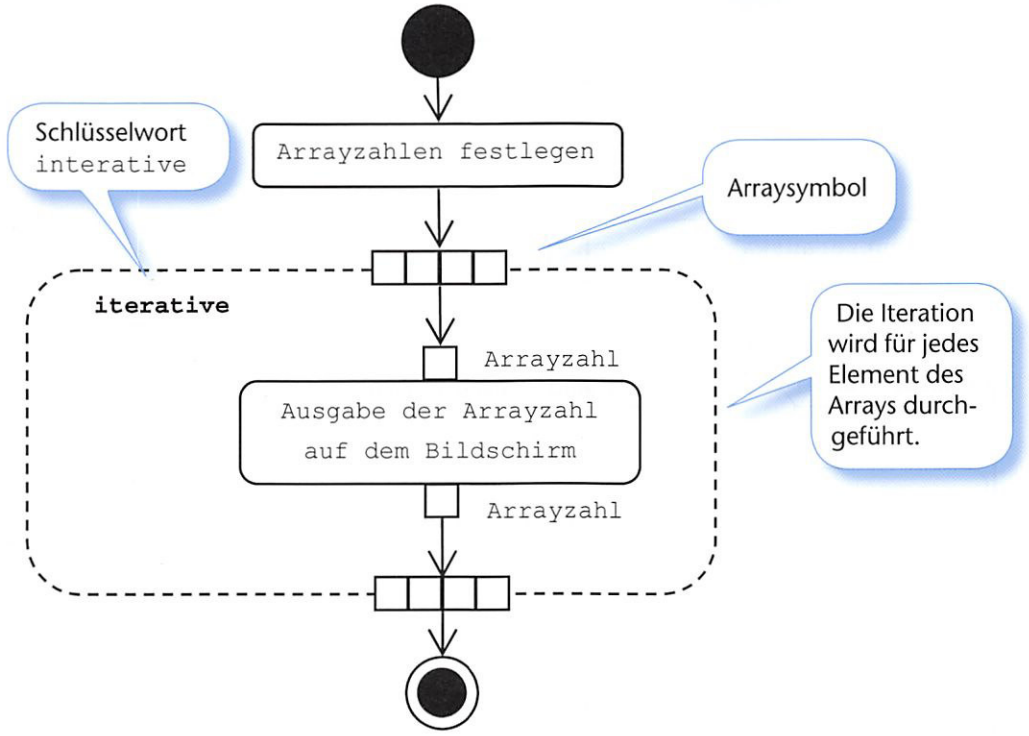


Hinweis:

Die obige Iteration ist eine sogenannte *fußgesteuerte* Iteration, da erst am Ende (nach mindestens einem Anweisungsblock) die Bedingung geprüft wird. Möchte man hingegen eine *kopfgesteuerte* Iteration, so sind einfach die Blöcke **do** und **while** zu vertauschen.

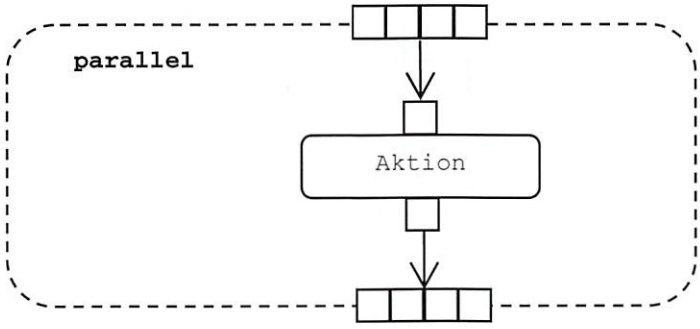
6.3.3 Expansionsbereiche

Eine besondere Form der Iteration ist der sogenannte Expansionsbereich. Sobald eine Iteration Aktionen oder Aktivitäten beinhaltet, die für jedes Element eines Arrays durchgeführt werden sollen, kann (statt der bislang vorgestellten Form der Iteration) der Expansionsbereich eingesetzt werden. Der Expansionsbereich wird durch eine Rechteck mit gestrichelten Linien dargestellt und besitzt einen Eingangs- und Ausgangsbereich für das Array (symbolisiert durch nebeneinander angeordnete Quadrate). Das folgende Beispiel zeigt die Ausgabe eines beliebigen Zahlenarrays.



Hinweise:

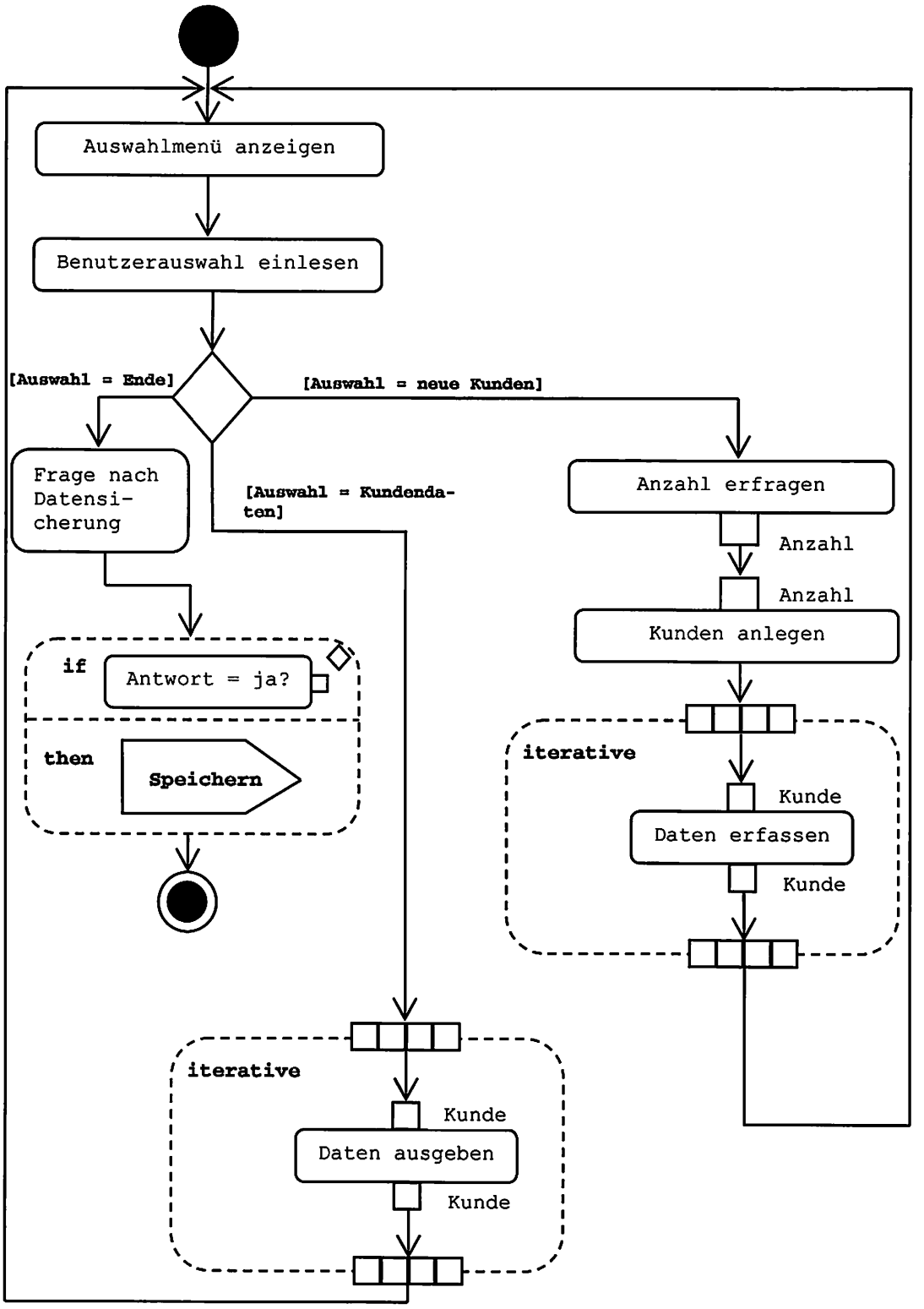
- In modernen Programmiersprachen (wie C#) gibt es besondere Iterationsanweisungen, die einen Expansionsbereich einfach umsetzen können (der Befehl **`foreach`** beispielsweise).
- Neben einer iterativen (hintereinander erfolgenden) Abarbeitung der Arrayelemente kann der Expansionsbereich auch eine parallele Verarbeitung vorgeben. Dazu muss das Schlüsselwort **`parallel`** benutzt werden. Die parallele Abarbeitung ist in sogenannten *Threads* vorstellbar. Das macht dann Sinn, wenn die Zeit für die Bearbeitung eines Elements sehr hoch ist. Das ist beispielsweise der Fall, wenn durch die Abarbeitung der Elemente die Antwortzeit eines Systems einige Sekunden dauert und damit benutzerunfreundlich wird.

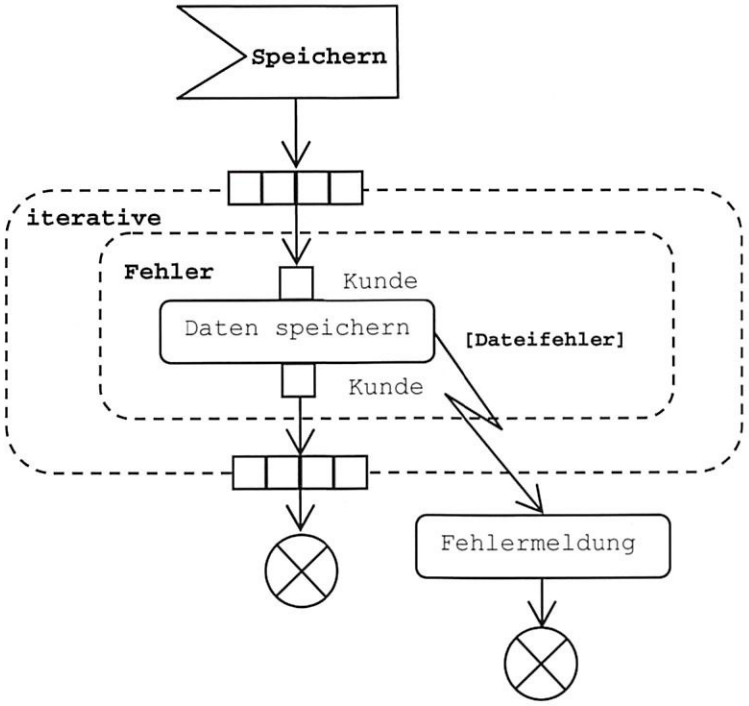


6.4 Umsetzung eines Aktivitätsdiagramms

Das folgende Beispiel soll ein Ausschnitt aus einer Software zur Verwaltung von Kunden sein. Der Benutzer erhält die Möglichkeit, neue Kunden anzulegen bzw. bestehende Kundendaten auf dem Bildschirm anzeigen zu lassen. Zusätzlich sollen die Daten gesichert werden können. Das Anlegen

und Ausgeben der Kunden wird mit Expansionsbereichen umgesetzt, die Sicherung wird durch ein Signal in Gang gesetzt. Ein Unterbrechungsbereich soll die Datensicherung kontrolliert beenden, wenn ein Fehler auftritt. Zusätzlich werden noch eine Verzweigung und eine einseitige Selektion verwendet, um das Auswahlmenü für den Benutzer zu gestalten und eine Abfrage vor dem Beenden des Programms durchzuführen.





6.4.1 Umsetzung des Aktivitätsdiagramms in C#

```
using System;
using System.Collections;
namespace UML_IT_BERUFE
{
   class CKunde
   {
         private string name;
         public CKunde(){ name = "NN"; }
         public void SetName(string n) { name = n; }
         public string GetName() { return name; }
   }
   class CFirma
   {
         private ArrayList kundenListe = new ArrayList();
         public CFirma() {}

         public void Einfuegen(CKunde vKunde)
         {
             kundenListe.Add(vKunde);
         }

         public void Menue()
         {
            string wahl, antwort;
            int neu;
            while (true)
```

Neue Kunden in die Liste einfügen

```
    {
        Console.WriteLine("Auswahlmenue:");
        Console.WriteLine("<1> Neue Kunden anlegen");
        Console.WriteLine("<2> Kundendaten ausgeben");
        Console.WriteLine("<3> Ende");
        Console.WriteLine("Ihre Wahl: ");
        wahl = Console.ReadLine();
        switch (wahl)
        {
        case "1":
            Console.WriteLine("Wie viele Kunden?");
            neu = Convert.ToInt32(Console.ReadLine());
            Anlegen(neu);
        break;

        case "2":
            Ausgabe();
        break;



        case "3":
            Console.Write("Daten sichern? (j/n)");
            antwort = Console.ReadLine();
            if (antwort == "j") Speichern();
            return;
        break;
        }
    }
}
public void Anlegen(int neu)
{
    string name;
    Console.WriteLine("Anlegen neuer Kunden:");
    for (int i = 0 ; i < neu; i++)
     {
        Console.WriteLine("Name von Kunde "
                + (i+1) + ": ");
        name = Console.ReadLine();
        CKunde vKunde = new CKunde();
        vKunde.SetName(name);
        Einfuegen(vKunde);
    }
}
```

Umsetzung der Verzweigung

Simulation „Signal senden“

Ende des Steuerungsflusses

Übergabe der Anzahl – Umsetzung des Objektflusses von Objekt `Anzahl`

```
        public void Ausgabe()
        {
            int i = 1;
            Console.WriteLine("Auflistung der Kunden:");
            foreach (CKunde kunde in kundenListe)
            {
                Console.WriteLine("Kunde " + (i++) + ": ");
                Console.WriteLine(kunde.GetName());
            }
        }
        public void Speichern()
        {
            int i = 0;

            string[] zeilen = new string[kundenListe.Count];
            foreach (CKunde kunde in kundenListe)
            {
                zeilen[i++] = kunde.GetName();
            }
            try
            {
                System.IO.File.WriteAllLines("Sicherung.txt", zeilen);
            }

            catch
            {
                Console.WriteLine("Fehler!");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            CFirma firma = new CFirma();
            firma.Menue();
        }
    }
}
```

Ausgabe der Kundendaten – Umsetzung des Expansionsbereiches

Umsetzung des „Signal-Empfangs" in Form einer Methode

Umsetzung des Expansionsbereiches

Umsetzung des Unterbrechungsbereiches durch Exception-Handling

Start des Steuerungsflusses

7 Beispiel einer Softwareentwicklung

In diesem Kapitel sollen nun die bisher gezeigten UML-Diagrammtypen in den Prozess einer objektorientierten Softwareentwicklung eingebunden werden. Alle wichtigen Schritte werden exemplarisch beschrieben – ein Schwerpunkt liegt dabei auf der Identifizierung der beteiligten Klassen des zu entwickelnden Softwaresystems. Zusätzlich soll ein Einblick in die Nutzung eines CASE-Tool gegeben werden. Dadurch soll die objektorientierte Softwareentwicklung professioneller werden.

7.1 Nutzung eines CASE-Tools

Die computergestützte Softwareentwicklung bedeutet, dass es Programme gibt, mit denen möglichst professionell und effizient Software (also auch Programme) entwickelt werden können. Solche Programme nennen sich (in englisch) CASE-Tools für **Computer-Aided-Software-Engineering-Tools**. Das Problem der CASE-Tools ist der Zeitaufwand, mit dem sich der Entwickler in dieses Programm einarbeiten muss. Dieser Aufwand macht sich jedoch bei allen weiteren Projekten bezahlt, da das Tool dann die Entwicklung strukturierter und effizienter gestalten kann. Auch die Dokumentation eines Software-Systems wird einfacher durch den Einsatz von CASE-Tools.

7.1.1 Aspekte von CASE-Tools

Es gibt die unterschiedlichsten CASE-Tools für die unterschiedlichsten Anforderungen an die Software-Entwicklung. Es gibt Tools für die strukturierte Softwareentwicklung, Tools für die Entwicklung von Datenbanken und natürlich die Tools für die objektorientierte Softwareentwicklung, die dann meistens auf der UML basieren.

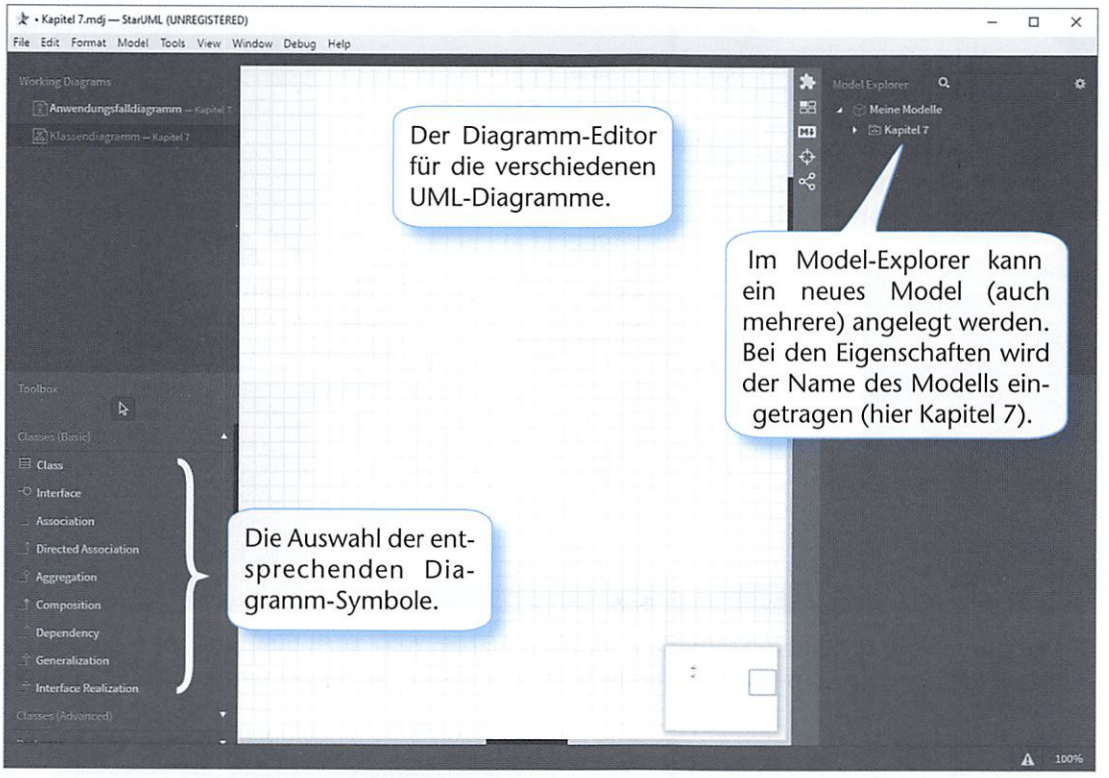
Viele Tools können an die Entwicklungsumgebungen der verwendeten Programmiersprache angebunden werden. Dadurch ist eine automatische Code-Generierung möglich. Beispielsweise kann ein Klassendiagramm von dem CASE-Tool direkt in den adäquaten Quellcode übersetzt werden. Die Code-Generierung ist auch nicht auf eine Sprache beschränkt – ein Tool wie *StarUML* kann den Code in Java, C++ oder auch C# generieren.

Auch der umgekehrte Weg ist möglich, das so genannte **Reverse-Engineering**. Dabei wird aus bestehendem Quellcode das zugrunde liegende Modell (beispielsweise ein Klassendiagramm) generiert. Auf diese Art können bestehende Programme noch einmal auf den Prüfstand der Konzeption.

7.1.2 Ein Modell mit StarUML anlegen

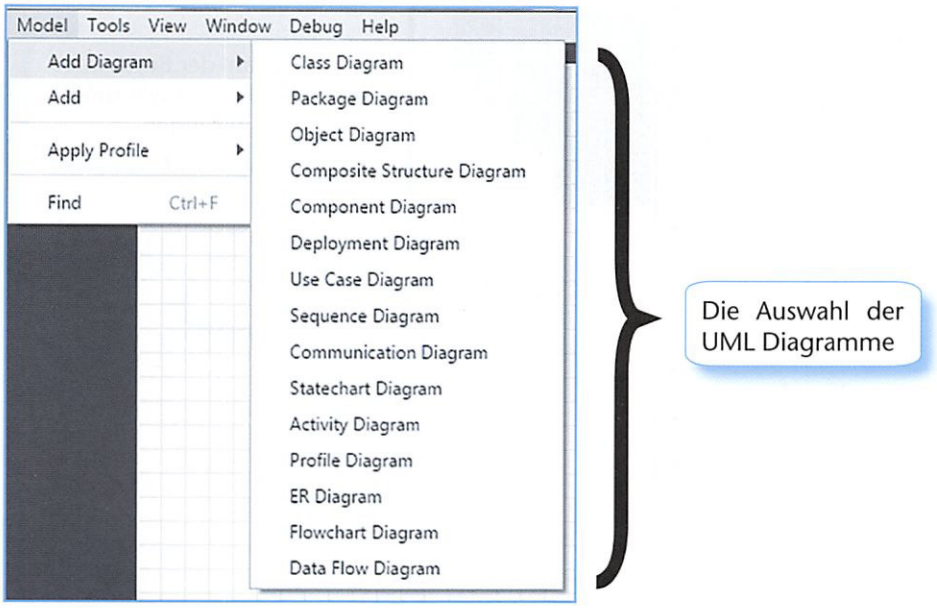
Als Beispiel für die Verwendung von CASE-Tools wird in den nächsten Schritten das Programm *StarUML* verwendet. Dieses Tool ist in einer Evaluationsversion kostenfrei und kann über die Internet-Seite der Firma *MKLab Co., Ltd.* heruntergeladen werden.

Am Anfang der Entwicklung eines neuen Software-Systems steht bei ***StarUML*** das Anlegen eines neuen Modells:

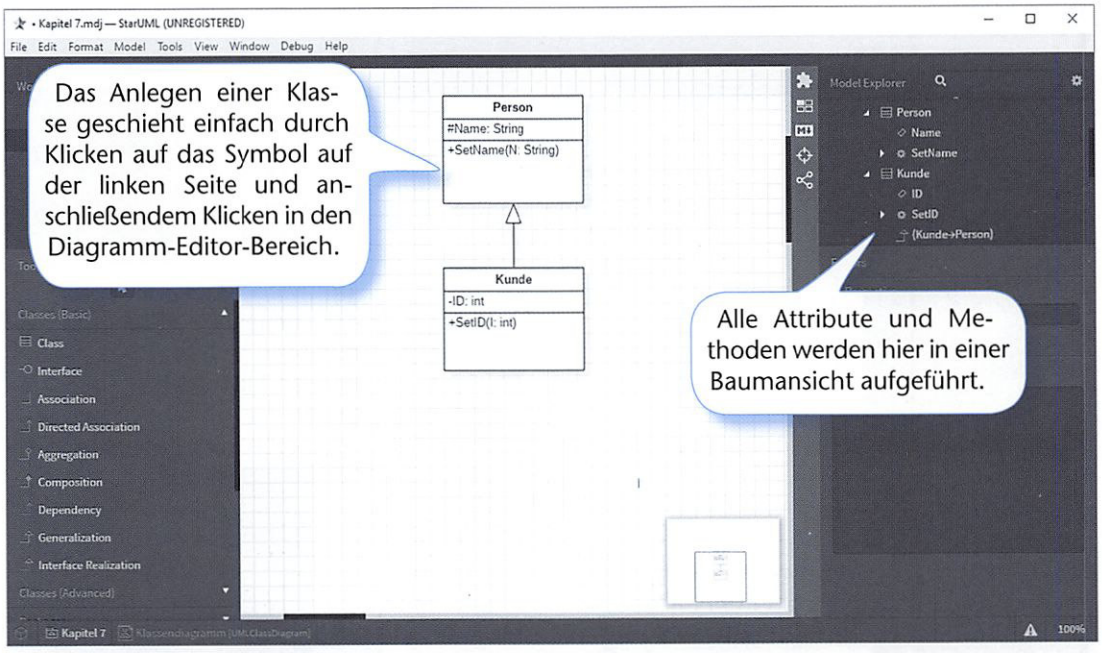


7.1.3 Diagramme in StarUML anlegen

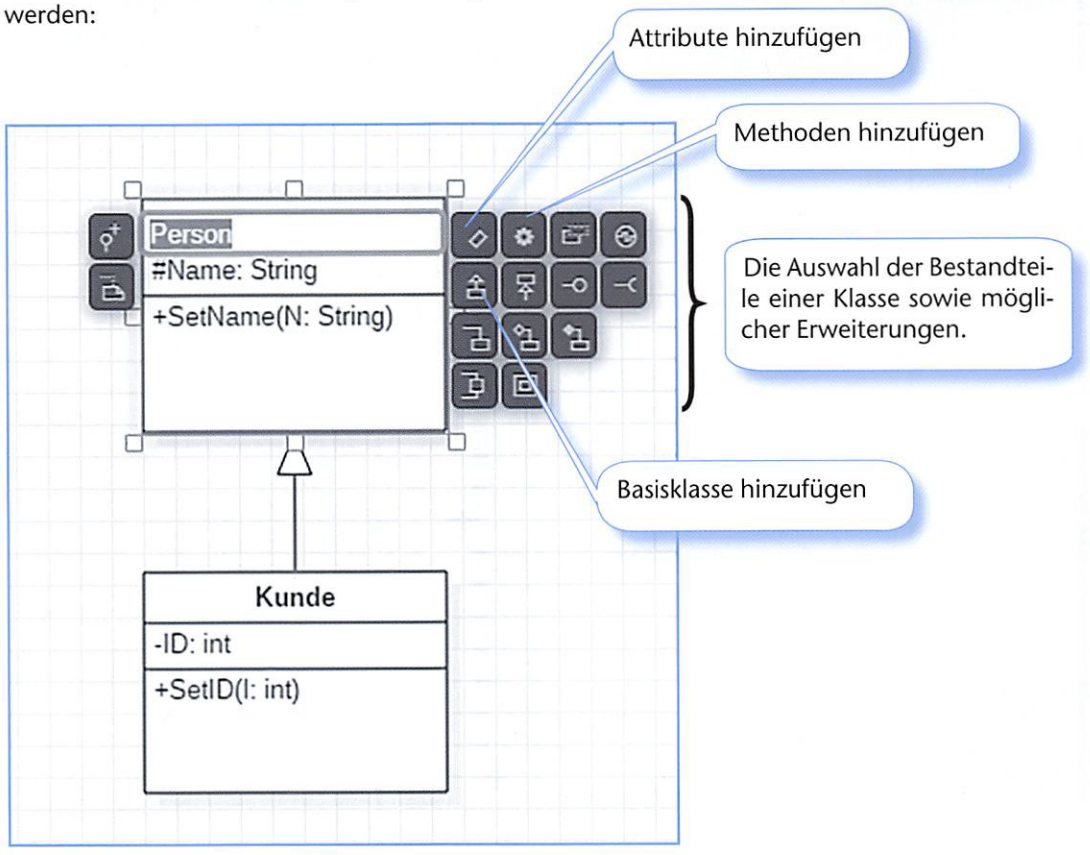
Über den Menüpunkt **`Model → Add Diagramm`** werden die verschiedenen Diagramme in das das Modell integriert:



Das folgende Beispiel zeigt die Integration eines Klassendiagramms:

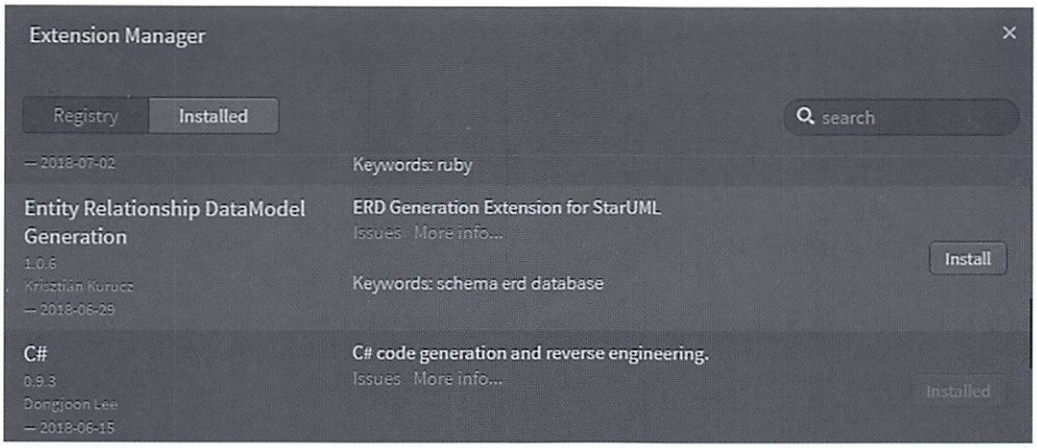


Das Ergänzen von Attributen, Methoden, weiteren Klassen oder auch Interfaces wird durch eine Direktauswahl an dem Klassensymbol möglich. Dazu muss auf das Klassensymbol doppelt geklickt werden:

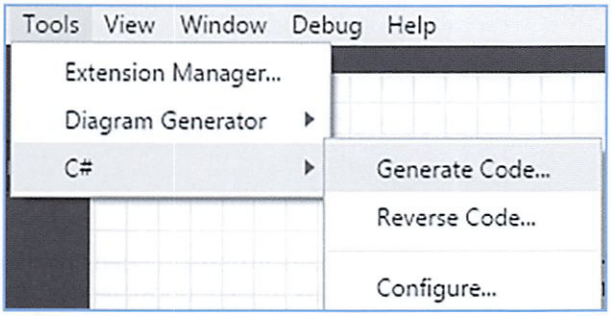


Hinweis:

Durch das Einbinden einer Programmiersprache über den Extension-Manager (Menüpunkt Tools) kann auch die automatische Code-Erzeugung genutzt werden. Hier wurde beispielsweise die Programmiersprache C# eingebunden:



Anschließend kann eine automatische Code-Generierung erfolgen:



Das Beispiel mit den Klassen `Person` und `Kunde` wird dann automatisch in C# umgesetzt:

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

public class Person
{

    public Person()
    {
    }

    protected String Name;

    /// <summary>
    /// @param N
    /// </summary>
    public void SetName(String N)
    {
        // TODO implement here
    }
}
```

```
public class Kunde : Person
{

    public Kunde()
    {
    }

    private int ID;

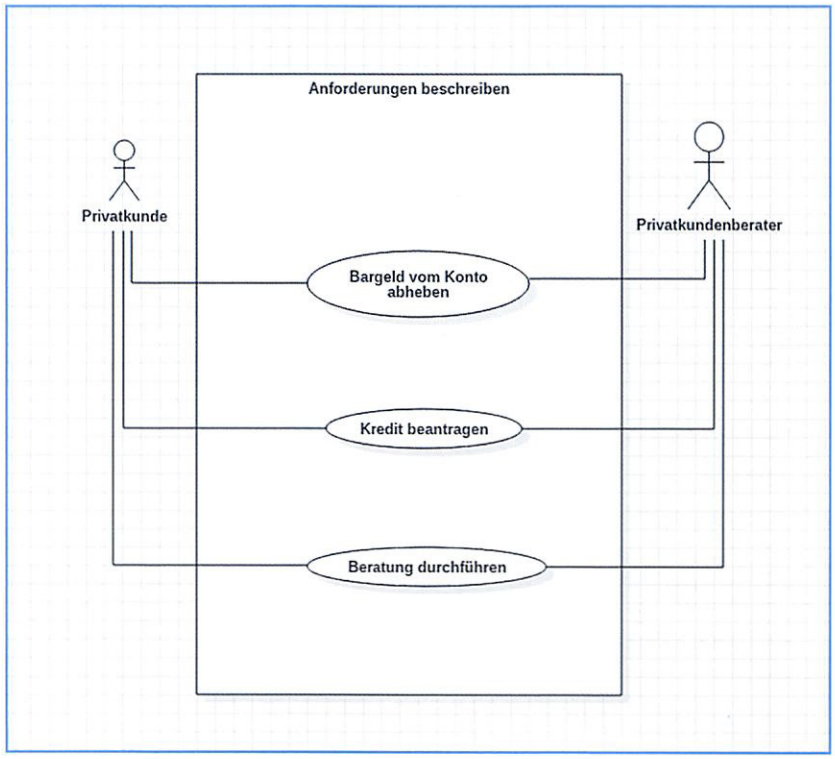
    /// <summary>
    /// @param I
    /// </summary>
    public void SetID(int I)
    {
        // TODO implement here
    }
}
```

7.2 Beispiel einer objektorientierten Softwareentwicklung mit UML und C#

Ausgangslage des Beispiels:
Für eine Bank soll eine Software entwickelt werden, die die Geschäftsprozesse mit den Privatkunden in einem EDV-System abbildet. In erster Linie geht es darum, dass Anwendungsfälle wie „Bargeld abheben" oder „Kredit beantragen" sowie „Beratungen durchführen" EDV-technisch erfasst und verwaltet werden. Die Diagramme werden mit dem CASE-Tool *StarUML* erstellt.

7.2.1 Anforderungen mit einem Anwendungsfalldiagramm beschreiben

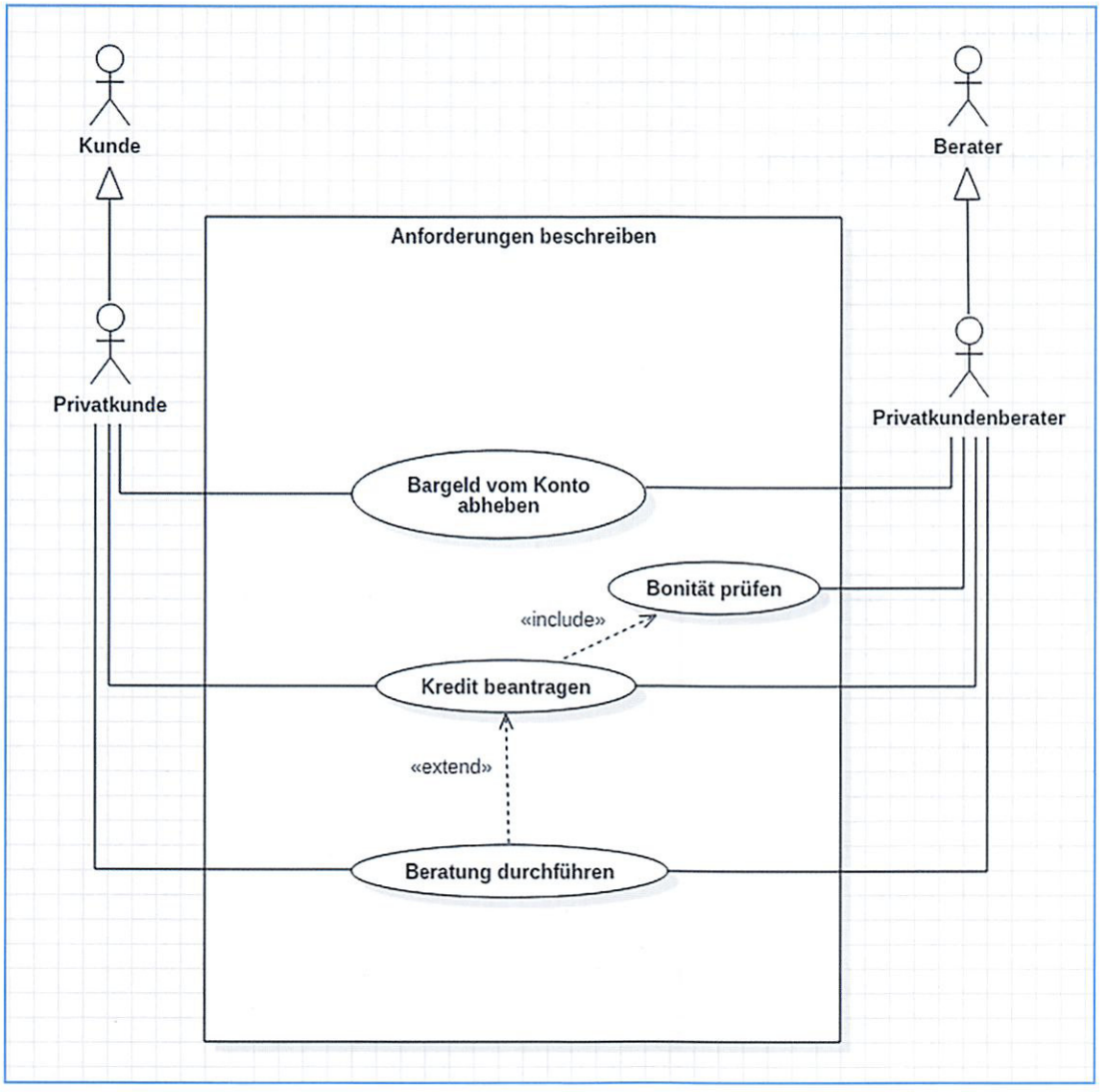
In einem ersten Schritt werden die Anforderungen an das System durch ein Anwendungsfalldiagramm beschrieben. In der Phase der Anforderungen wird das Diagramm noch allgemein gehalten. Weitere Details und Verfeinerungen werden dann in der objektorientierten Analyse (**OOA**) ergänzt bzw. durchgeführt.



Die Anwendungsfälle „Bargeld vom Konto abheben", „Kredit beantragen" und „Beratung durchführen" sind angelegt worden. Die beiden Akteure „Privatkunde" und „Privatkundenberater" sind mit den Anwendungsfällen assoziert.

7.2.2 Objektorientierte Analyse (OOA)

Das Ziel der objektorientierten Analyse ist das möglichst vollständige Erfassen der Anforderungen an das System. Dazu wird beispielsweise das Anwendungsfalldiagramm weiter spezifiziert. Die Beziehungen zwischen den Akteuren und Anwendungsfällen sowie mögliche Generalisierungen werden analysiert und ergänzt.



- Kunde und Berater können als Generalisierungen von Privatkunde bzw. Privatkundenberater identifiziert werden. Das ist sinnvoll, denn das System könnte zu einem späteren Zeitpunkt auch um Geschäftskunden erweitert werden.
- Der Anwendungsfall „Bonität prüfen" wurde als weiterer Anwendungsfall angelegt und hat eine `<<include>>`-Beziehung zum Anwendungsfall „Kredit beantragen". Bei jedem Kreditantrag muss die Bonität geprüft werden.
- Der Anwendungsfall „Kredit beantragen" hat eine `<<extend>>`-Beziehung zum Anwendungsfall „Beratung durchführen". Es ist nicht zwingend eine Beratung anzusetzen, nachdem ein Kunde einen Kreditantrag gestellt hat, aber durchaus möglich.

Identifizierung der Klassen

Mithilfe des Anwendungsfalldiagrammes sollen nun die beteiligten Klassen ermittelt. Damit soll ein erstes einfaches Klassendiagramm erstellt werden. Dieser Prozess ist relativ schwierig und hängt stark von der Erfahrung und dem Weitblick der Entwickler ab. Das bedeutet, dass es keine allgemein gültige und immer anwendbare Methode gibt, um die beteiligten Klassen zu ermitteln. An dieser Stelle soll deshalb ein einfaches Verfahren dargestellt werden.

Einfache Regeln zur Identifikation von Klassen:

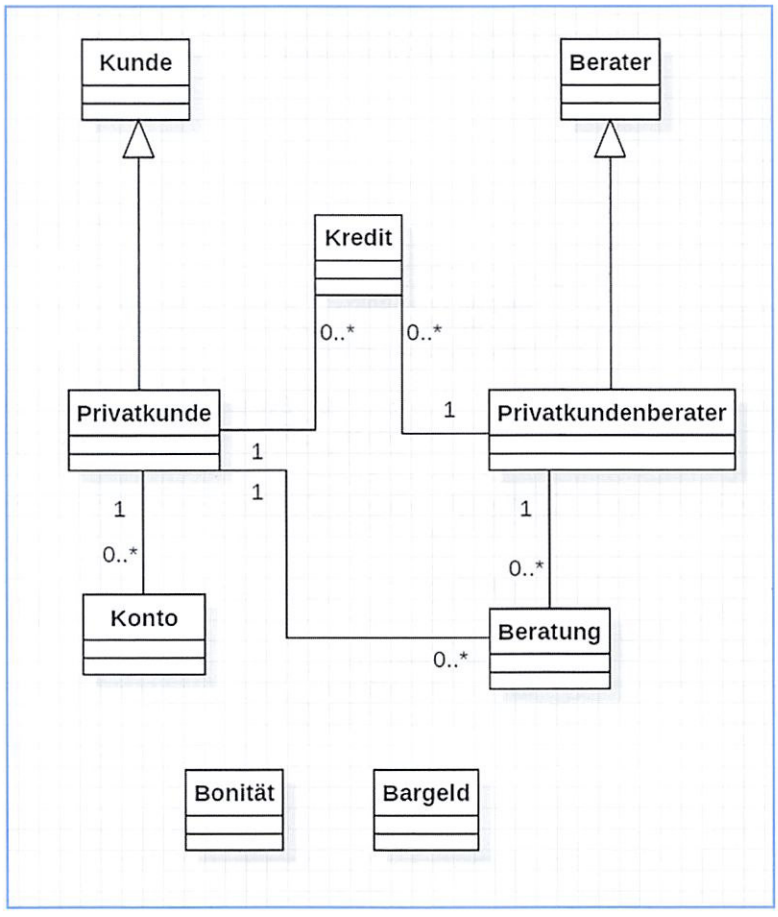
- Zuerst werden alle **Akteure** als Kandidaten für Klassen aufgenommen.
- Die Anwendungsfälle und deren zusätzliche Beschreibungen werden nach **Substantiven** durchsucht. Alle Substantive sind mögliche Klassenkandidaten.
- Kandidaten, die **mehrere Attribute** haben, sind mögliche **Klassen**. Kandidaten, die keine Attribute haben, sind oftmals selbst nur ein Attribut einer anderen Klasse.

Mithilfe dieser einfachen Regeln werden nun die Kandidaten für Klassen aufgelistet und anschließend in einem ersten Klassendiagramm festgehalten.

Kandidaten für die Klassen:

```
Privatkunde    Kunde      Privatkundenberater      Berater
Bargeld        Konto      Kredit       Bonität     Beratung
```

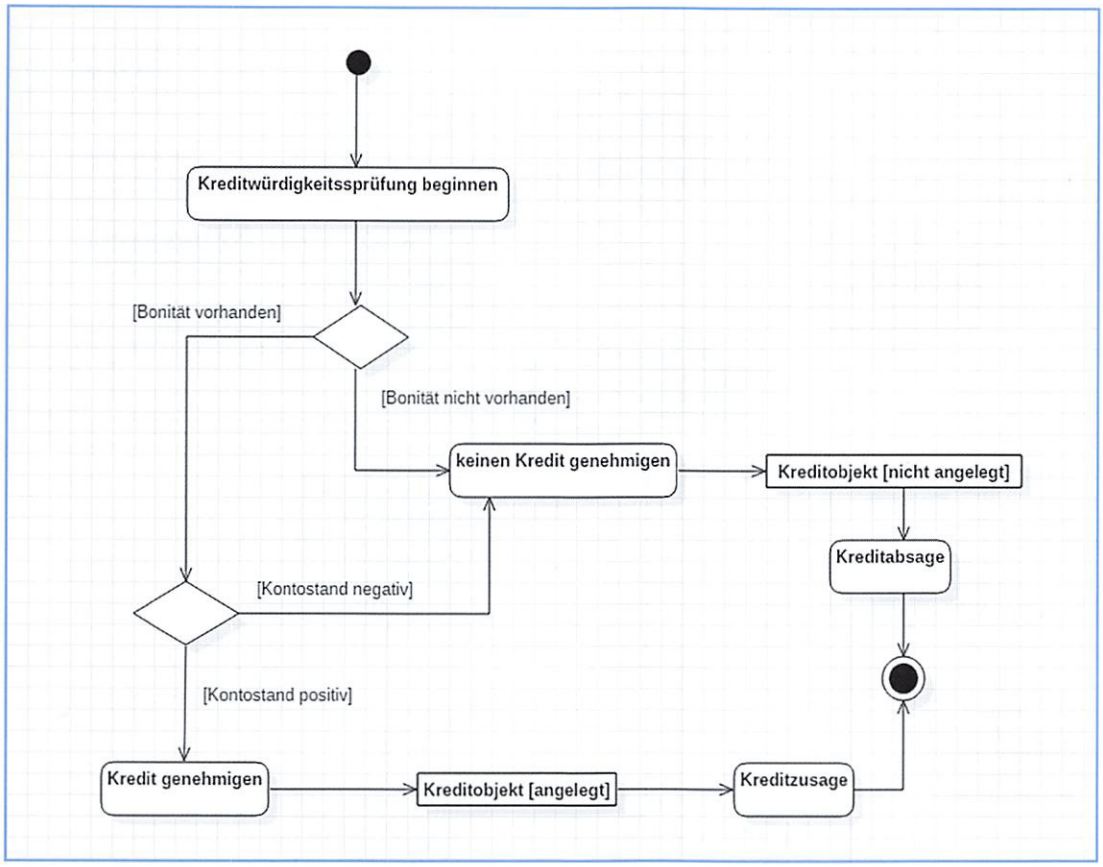
Ein erstes Klassendiagramm:



Es zeigt sich, dass die Kandidaten Bonität und Bargeld sich nicht wirklich in das Klassendiagramm einfügen. Die Bonität könnte als Attribut der Klasse Kunde angelegt werden und das Bargeld könnte als Attribut oder als Methode in der Klasse Konto eingefügt werden. Diese Arbeit wird dann im nächsten Schritt, dem objektorientierten Design, erledigt.

Einsatz des Aktivitätsdiagramms

Das Aktivitätsdiagramm kann beispielsweise eingesetzt werden, um einen Anwendungsfall näher zu beleuchten. In diesem Fall soll exemplarisch der Anwendungsfall „Kredit beantragen" mit dem Aktivitätsdiagramm beschrieben werden.



7.2.3 Objektorientiertes Design (OOD)

Das Ziel des objektorientierten Designs ist der Entwurf einer grundlegenden Architektur eines Softwaresystems. Wenn es sich in der Analysephase um das *Was* des Systems dreht, so dreht es sich in der Designphase um das *Wie*.

In der Designphase hat der Entwickler folgende Fragen zu berücksichtigen:

- Welche Plattform wird für die Umsetzung gewählt?
- Welche Entwicklungsumgebung wird eingesetzt?
- Welche Kosten sind in Betracht zu ziehen?

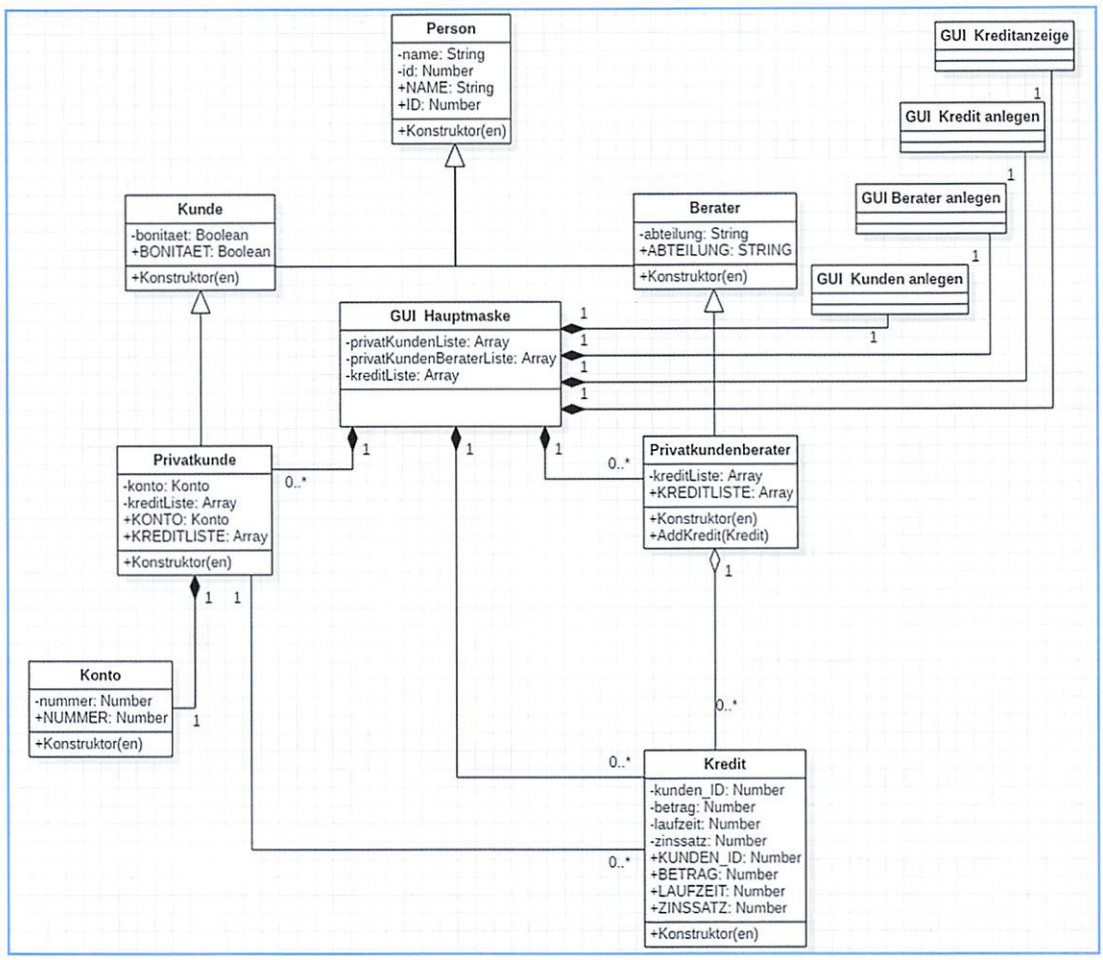
Weiterhin müssen folgende Aspekte beachtet werden:

- Schnittstellen zwischen Teilsystemen festlegen.
- Voraussetzungen für die Entwicklung (Programmierung) im Team schaffen.
- Konkrete Voraussetzungen für die Implementierung schaffen.

Je nach Softwaresystem müssen auch die Benutzeroberflächen entwickelt werden. Die meisten Aspekte sind für dieses Beispiel vorgegeben, deshalb konzentrieren sich die weiteren Ausführungen auf die konkreten Voraussetzungen für die Implementierung.

In einem ersten Schritt soll deshalb das Klassendiagramm verfeinert werden. Ein Sequenzdiagramm soll dann die Kommunikation der Klassen für den Prozess „Kredit beantragen" beleuchten. Anschließend werden Benutzeroberfächen entwickelt.

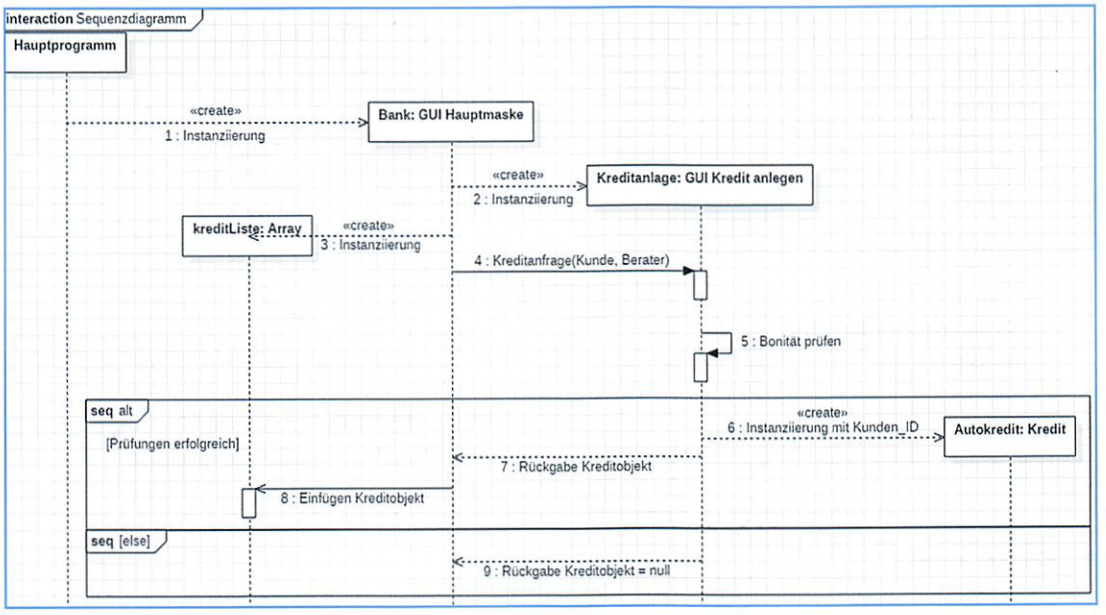
Verfeinerung des Klassendiagrammes:



Das Klassendiagramm zeigt nur eine exemplarische Teilverfeinerung des Diagramms aus der Analyse mit dem **Schwerpunkt auf der Kreditvergabe**. Die Klassen wurden konkretisiert und mit Attributen und Methoden bzw. Eigenschaften versehen. Die Beziehungen wurden detaillierter festgelegt – beispielsweise die Komposition zwischen Privatkunde und Konto sowie die Aggregation zwischen Privatkundenberater und Kredit. Ebenso wurden die Benutzeroberflächen als Klassen modelliert. Die Benutzeroberflächen sollen zur Eingabe neuer Kunden und Berater dienen sowie die Abwicklung einer Kreditvergabe simulieren. Die Hauptmaske dient dabei als Navigationsmaske (ähnlich einem Menü). Von der Hauptmaske sollen alle Funktionalitäten aufrufbar sein.

Einsatz des Sequenzdiagramms:

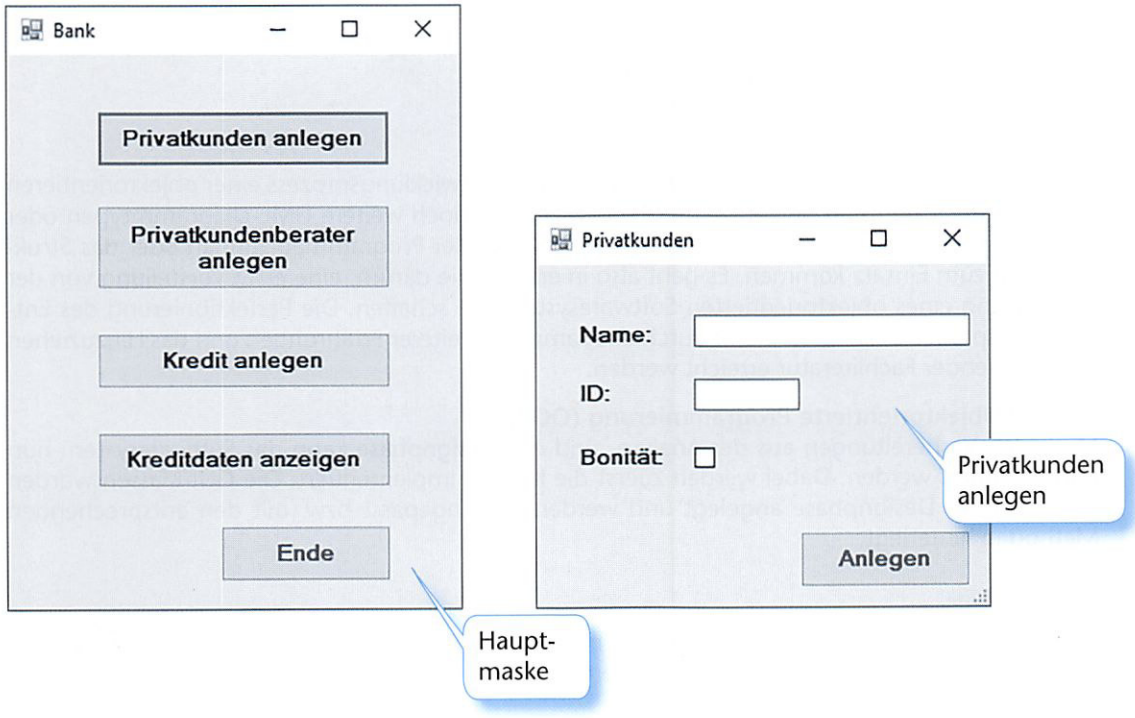
Mithilfe des Sequenzdiagramms wird nun exemplarisch der Prozess „Kredit beantragen" näher beleuchtet. Die beteiligten Objekte werden in ihrem Kommunikationsverhalten untereinander beschrieben.

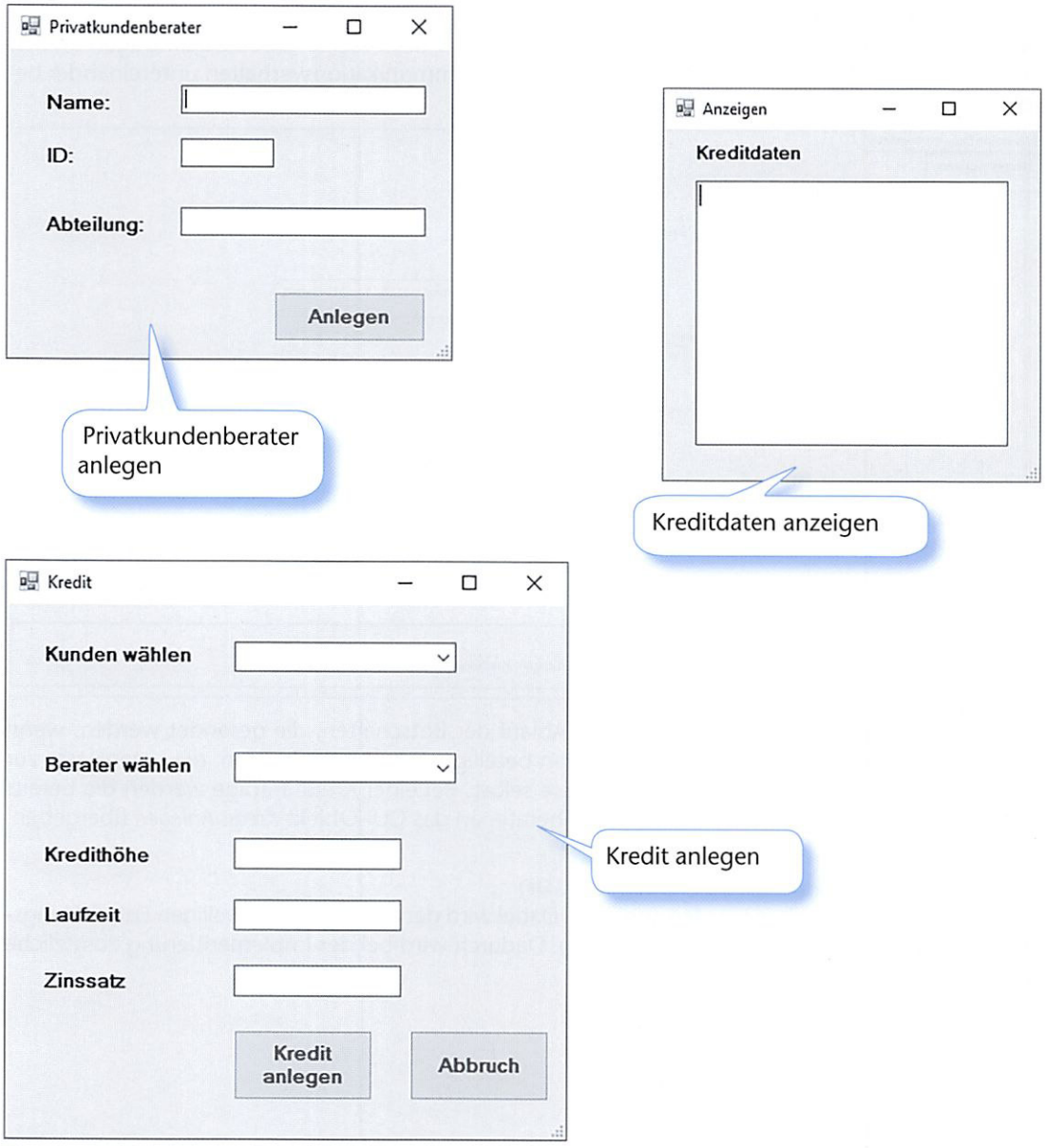


Das Sequenzdiagramm zeigt den zeitlichen Ablauf der Botschaften, die gesendet werden, wenn ein Privatkunde eine Kreditanfrage stellt. Daran beteiligt sind die GUI-Klassen, die Listenklasse zur Speicherung der Kredite sowie die Kreditklasse selbst. Bei einer Kreditanfrage werden die bereits bestehenden Privatkunden und Privatkundenberater an das GUI-Objekt *Kredit Anlegen* übergeben.

Skizzierung der Benutzeroberflächen (GUI)

Die möglichen Oberflächen werden angelegt. Dabei wird der Designer der jeweiligen Entwicklungsumgebung benutzt (hier: *Microsoft Visual C#*). Dadurch wird bei der Implementierung zusätzliche Arbeit gespart.





Die bisherige Darstellung zeigt einen exemplarischen Entwicklungsprozess einer objektorientieren Softwareentwicklung. In den einzelnen Phasen können noch weitere UML-Diagrammtypen oder auch Elemente der strukturierten Programmierung wie der Programmablaufplan oder das Struktogramm zum Einsatz kommen. Es geht also in erster Linie darum, eine erste Vorstellung von der Entwicklung eines objektorientierten Softwaresystems zu schaffen. Die Perfektionierung des Entwicklungsprozesses kann dann nur durch das Sammeln weiterer Erfahrungen und das Hinzuziehen entsprechender Fachliteratur erreicht werden.

7.2.4 Objektorientierte Programmierung (OOP)

Nach den Vorbereitungen aus der Analyse- und der Designphase kann das Softwaresystem nun implementiert werden. Dabei werden zuerst die Klassen implementiert. Die GUI-Klassen wurden bereits in der Designphase angelegt und werden nur angepasst bzw. mit den entsprechenden Methoden hinterlegt.

Hinweis:

Die Implementierung der Klassen kann mithilfe von *StarUML* geschehen. Dazu kann die automatische Codegenerierung benutzt werden.

Implementierung der Klassen:

```
public class CPerson
{
    protected string name;
    protected int id;
    public CPerson() { name = "NN"; id = -1; }
    public string NAME
    {
            get { return name;  }
            set { name = value; }
    }
    public int ID
    {
            get { return id; }
            set { id = value; }
    }
}

public class CKunde : CPerson
{
    protected bool bonitaet;
    public CKunde() : base() { bonitaet = false; }
    public bool BONITAET
    {
            get { return bonitaet; }
            set { bonitaet = value; }
    }
}

public class CPrivatkunde : CKunde
{
    private CKonto konto = new CKonto();
    public CPrivatkunde() : base() { }
    public int KONTO
    {
            get { return konto.NUMMER; }
            set { konto.NUMMER = value; }
    }
}

public class CKonto
{
    private int nummer;
    public CKonto() { nummer = -1; }
```

Erbt von CPerson

Erbt von CKunde

Umsetzung der Komposition

```
    public int NUMMER
    {
            set { if (value > 0 && value < 9999) nummer = value; }
            get { return nummer; }
    }
}
```

Erbt von CPerson

```
public class CBerater : CPerson
{
    protected String abteilung;
    public CBerater() : base() { abteilung = ""; }
    public string ABTEILUNG
        {
            get { return abteilung; }
            set { abteilung = value; }
    }
}
```

Erbt von CBerater

Umsetzung der Aggregation

```
public class CPrivatkundenberater : CBerater
{
    private System.Collections.ArrayList kreditListe =
                    new System.Collections.ArrayList();

    public CPrivatkundenberater() : base() { }
    public System.Collections.ArrayList KREDITLISTE
    {
            get { return kreditListe; }
    }
    public void AddKredit(CKredit vK)
    {
        kreditListe.Add(vK);
    }
}

public class CKredit
{
    private int kunden_ID;
    private double betrag;
    private int laufzeit;
    private double zinssatz;

    public CKredit()
    {
        kunden_ID = -1;
        betrag = 0;
```

Umsetzung der Assoziation

```
        laufzeit = 0;
        zinssatz = 0;
    }
    public int KUNDEN_ID
    {
            get { return kunden_ID; }
            set { if (value > 0) kunden_ID = value; }
    }
    public double BETRAG
    {
            get { return betrag; }
            set { if (value > 0) betrag = value; }
    }
    public int LAUFZEIT
    {
            get { return laufzeit; }
            set { if (value > 0 && value < 999) laufzeit = value; }
    }
    public double ZINSSATZ
    {
            get { return zinssatz; }
            set { if (value > 0 && value <= 100) zinssatz = value; }
    }
}
```

Implementierung der GUI-Klassen:

Der Hauptdialog

```
public partial class GUIHauptmaske : Form
{
```

Objekte für die „Unterformulare"

```
    private GUIKredit dlgKredit = new GUIKredit();
    private GUIPrivatkunden dlgPrivatkunden = new GUIPrivatkunden();
    private GUIPrivatkundenberater dlgPrivatkundenBerater
                = new GUIPrivatkundenberater();
    private GUIKreditAnzeige dlgAnzeige = new GUIKreditAnzeige();

    private System.Collections.ArrayList privatkundenListe
                = new System.Collections.ArrayList();
    private System.Collections.ArrayList privatkundenberaterListe
                = new System.Collections.ArrayList();
    private System.Collections.ArrayList kreditListe
                = new System.Collections.ArrayList();
```

Umsetzung der Aggregationen

```
public GUIHauptmaske()
{
    InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e) { }

private void btnKundenAnlegen_Click(object sender, EventArgs e)
{
    dlgPrivatkunden.ShowDialog();
    privatkundenListe.Add(dlgPrivatkunden.KUNDE);
}

private void btnBeraterAnlegen_Click(object sender, EventArgs e)
{
    dlgPrivatkundenBerater.ShowDialog();
    privatkundenberaterListe.Add(dlgPrivatkundenBerater.KUNDE);
}

private void btnKreditAnlegen_Click(object sender, EventArgs e)
{
    dlgKredit.Fuellen(privatkundenListe,
                      privatkundenberaterListe);

    dlgKredit.ShowDialog();
    kreditListe.Add(dlgKredit.KREDIT);
}

private void buttonEnde_Click(object sender, EventArgs e)
{
        this.Close();
}

private void buttonAnzeigen_Click(object sender, EventArgs e)
{
    dlgAnzeige.Fuellen(privatkundenListe,
                       privatkundenberaterListe);
    dlgAnzeige.ShowDialog();
}
}
```

„Kunden Anlegen"-Dialog aufrufen

Neuen Kunden hinzufügen

„Berater Anlegen"-Dialog aufrufen

Neuen Berater hinzufügen

„Berater Anlegen"-Dialog aufrufen

Die bestehenden Kunden und Berater mitteilen.

Neuen Kredit hinzufügen

„Kredite anzeigen"-Dialog aufrufen

Die bestehenden Kunden und Berater mitteilen.

Der „Kredit anlegen"-Dialog

```
public partial class GUIKredit : Form
{
    System.Collections.ArrayList kListe;
    System.Collections.ArrayList bListe;
    CKredit vKredit;
    public GUIKredit() { InitializeComponent(); }

    public void Fuellen(System.Collections.ArrayList kListe,
             System.Collections.ArrayList bListe)
    {
        comboBoxKunde.Items.Clear();
        comboBoxBerater.Items.Clear();

        this.kListe = kListe;
        this.bListe = bListe;

            foreach (CPrivatkunde pk in kListe)
            {
                comboBoxKunde.Items.Add(pk.NAME);
            }
            foreach (CPrivatkundenberater pb in bListe)
            {
                comboBoxBerater.Items.Add(pb.NAME);
            }
    }

    private void buttonAbbruch_Click(object sender, EventArgs e)
    {
            this.Close();
    }

    public bool DBABfrage(int kontonummer)
        {
            System.Random r = new Random();
            int zufall = r.Next(10);
            if (zufall > 5) return true;
            return false;
    }

    public CKredit KREDIT
    {
        get { return vKredit; }
    }
```

Verweise auf die bestehenden Berater und Kunden.

Die Kombinationsfelder leeren.

Die Verweise übernehmen.

Die Kombinationsfelder füllen.

Hier wird der Einfachheit halber eine Datenbankabfrage simuliert. Mit 50%-Wahrscheinlichkeit ist die Kontoauskunft positiv. An dieser Stelle könnte auch eine echte Datenbankanbindung stehen.

```
    private void buttonAnlegen_Click(object sender, EventArgs e)
    {

        CPrivatkunde k = null;
        CPrivatkundenberater b = null;
        vKredit = null;

        string kname = comboBoxKunde.SelectedItem.ToString();
        string bname = comboBoxBerater.SelectedItem.ToString();

        foreach (CPrivatkunde pk in kListe)
        {
            if (kname == pk.NAME) k = pk;
        }
        foreach (CPrivatkundenberater pb in bListe)
        {
            if (bname == pb.NAME) b = pb;
        }

        if (DBABfrage(k.KONTO) == true)
        {
            if (k.BONITAET == true)
            {
                vKredit = new CKredit();
                vKredit.KUNDEN_ID = k.ID;
                vKredit.BETRAG = Convert.ToDouble(textBoxKreditHoehe.Text);
                vKredit.LAUFZEIT = Convert.ToInt32(textBoxLaufzeit.Text);
                vKredit.ZINSSATZ = Convert.ToDouble(textBoxZinssatz.Text);

                b.AddKredit(vKredit);
            }
            else MessageBox.Show("Kein Kredit möglich - fehlende Bonität");
        }
        else MessageBox.Show("Kein Kredit möglich - Konto überzogen");
        this.Close();
    }
}
public partial class GUIPrivatkunden : Form
{
    private CPrivatkunde vPrivatkunde;
    public GUIPrivatkunden()
    {
        InitializeComponent();
    }
```

Kredit auf `null` setzen – noch kein Kredit

Namen aus den Kombinationsfeldern merken

Kunden zu dem Namen finden

Berater zu dem Namen finden

Kreditwürdigkeit prüfen – nach Vorgabe im Aktivitätsdiagramm

Neuen Kredit anlegen

Kreditdaten zuweisen

Umsetzung der Aggregation

Der „Kunden anlegen"-Dialog

Verweis für das neue Kundenobjekt

```
    public CPrivatkunde KUNDE
    {
        get { return vPrivatkunde; }
    }
    private void buttonAnlegen_Click(object sender, EventArgs e)
    {
        vPrivatkunde = new CPrivatkunde();
        vPrivatkunde.NAME = textBoxName.Text;
        vPrivatkunde.ID = Convert.ToInt32(textBoxID.Text);
        if (checkBoxBonitaet.Checked == true)
                vPrivatkunde.BONITAET = true;
        else
                vPrivatkunde.BONITAET = false;
        this.Close();
    }
}
```

Neuen Kunden anlegen

Der „Berater anlegen"-Dialog

```
public partial class GUIPrivatkundenberater : Form
{

    //analog zum Privatkunden
}
```

Der „Kredite anzeigen"-Dialog

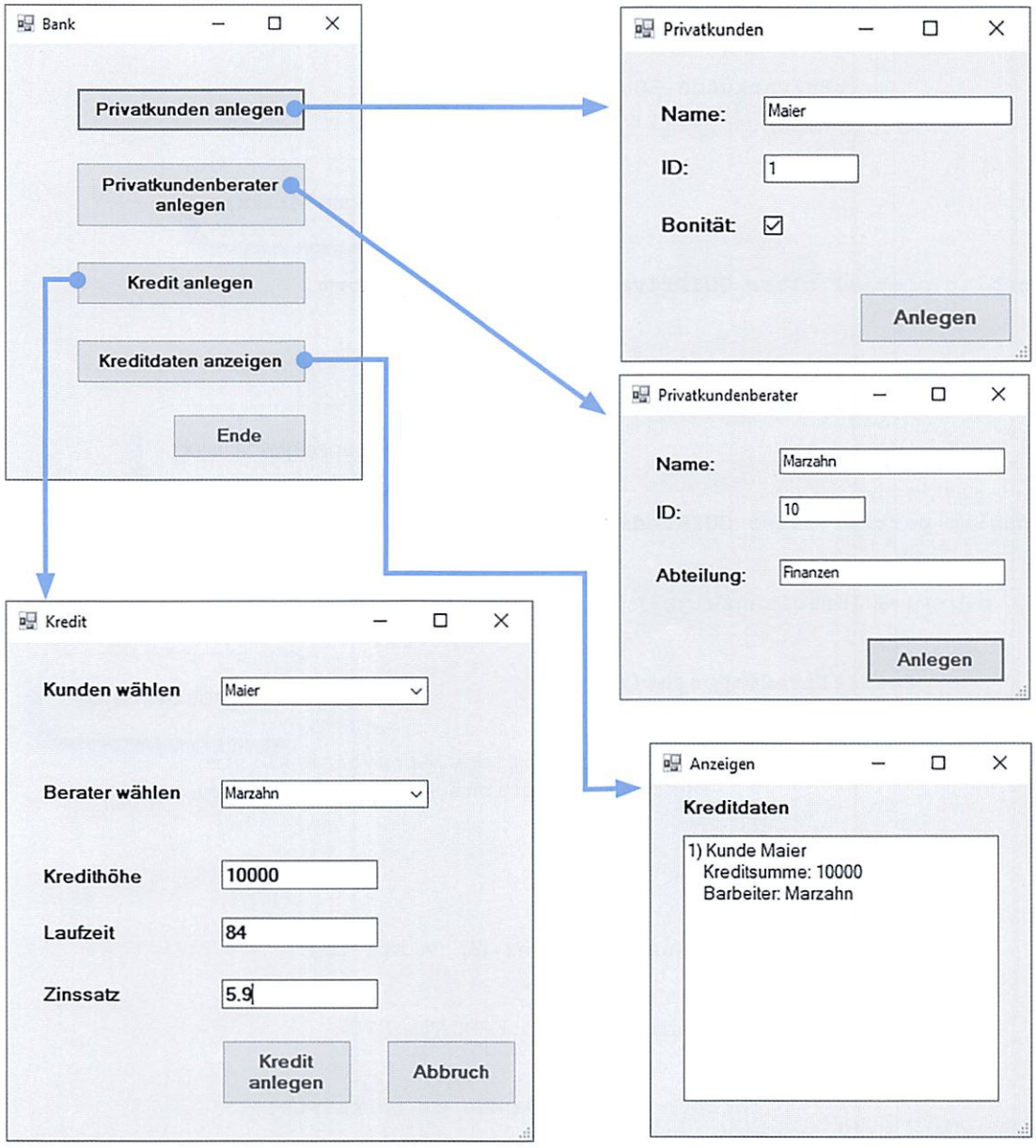
```
public partial class GUIKreditAnzeige : Form
{
    public GUIKreditAnzeige()
    {
        InitializeComponent();
    }
    public void Fuellen(System.Collections.ArrayList kListe,
                        System.Collections.ArrayList bListe)
    {
        textBoxDaten.Text = "";
        int zaehler = 1;
        foreach (CPrivatkundenberater pb in bListe)
        {
            foreach (CKredit k in pb.KREDITLISTE)
            {
                foreach (CPrivatkunde pk in kListe)
                {
                    if (k.KUNDEN_ID == pk.ID)
                    {
                        textBoxDaten.Text +=
                        zaehler.ToString() + ") Kunde " +
                        pk.NAME + "  Kreditsumme: " +
```

Übergabe der Listen

Hier wird anhand der Kreditlisten der Berater und der jeweiligen ID des Kunden eine Informationszeile zu dem Kredit zusammengestellt.

```
                                        k.BETRAG + "   Bearbeiter " +
                                        pb.NAME + "\r\n\r\n";
                                        zaehler++;
                                    }
                                }
                                }
                            }
                        }
                    }
```

Das neue *Softwaresystem* im *Einsatz*



8 Weitere UML-Diagramme

Neben den vorgestellten Diagrammen gibt es neun weitere Diagramme, die in den folgenden Unterkapiteln kurz vorgestellt werden. Einige Diagramme sind den bereits behandelten sehr ähnlich, zeigen das zu entwickelnde Softwaresystem allerdings aus anderen Blickwinkeln.

Strukturdiagramme

- ✓ Klassendiagramm
- ✓ Objektdiagramm
- ✓ **Kompositionsstrukturdiagramm**
- ✓ **Komponentendiagramm**
- ✓ **Verteilungsdiagramm**
- ✓ **Paketdiagramm**
- ✓ **Profildiagramm**

Verhaltensdiagramme

- ✓ Anwendungsfalldiagramm
- ✓ Sequenzdiagramm
- ✓ Aktivitätsdiagramm
- ✓ **Zustandsdiagramm**
- ✓ **Kommunikationsdiagramm**
- ✓ **Zeitverlaufsdiagramm**
- ✓ **Interaktionsübersichtsdiagramm**

8.1 Strukturdiagramme

8.1.1 Das Kompositionsstrukturdiagramm

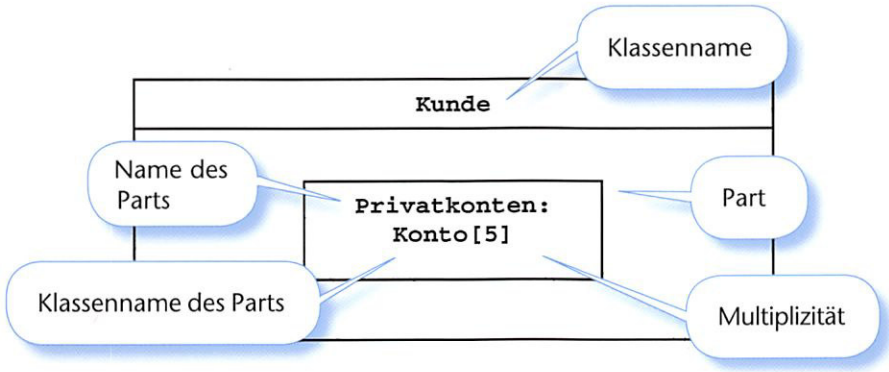
Das Kompositionsstrukturdiagramm dient zur Darstellung von internen Strukturen des zu entwickelnden Softwaresystems. Auf der einen Seite können Klassen mit zugehörigen Kompositionen (und auch Assoziationen) in eine kompakte Darstellung gebracht werden. Auf der anderen Seite kann die Arbeitsweise eines Systems (bzw. Teilsystems) durch sogenannte Kollaborationen deutlich gemacht werden. Dabei wird auch die Kommunikation zwischen einzelnen Objekten beleuchtet.

Klassen und Parts

Durch Kompositionen (oder auch Aggregationen) besitzt eine Klasse Attribute vom Datentyp einer anderen Klasse. Diese Attribute (Objekte der anderen Klasse) können als sogenannte **Parts** dargestellt werden.

Beispiel:

Die Klasse Kunde hat bis zu fünf Konten.

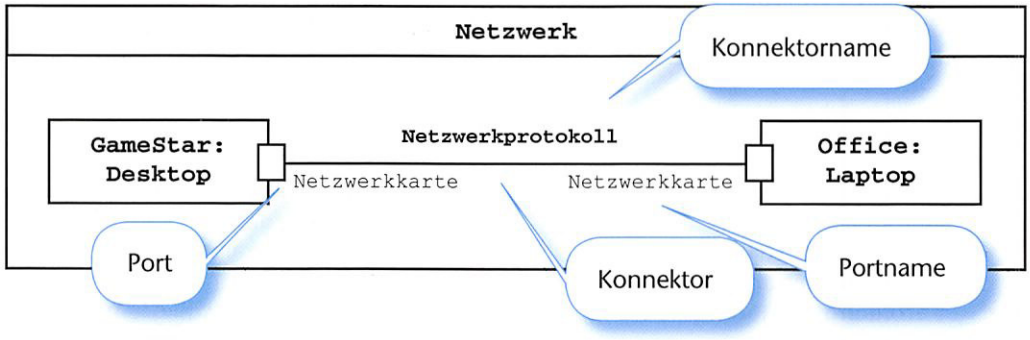


Ports und Konnektoren

Wenn eine Klasse oder ein Part eine Beziehung zu einer anderen Klasse oder einem Part hat, so kann das mit den Symbolen für Ports und Konnektoren dargestellt werden. Dabei kennzeichnen Ports, dass die Klasse oder der Part eine Interaktionsmöglichkeit nach außen anbietet. Mithilfe der Konnektoren können Ports miteinander verbunden werden. Damit ist eine Kommunikationsmöglichkeit zwischen den Ports (und damit auch zwischen den zugehörigen Klassen oder Parts) gegeben.

Beispiel:

In einem Netzwerk sind zwei Computer (Desktop und Laptop) über Ports (Netzwerkkarten) und einen Konnektor (Netzwerkprotokoll) verbunden.

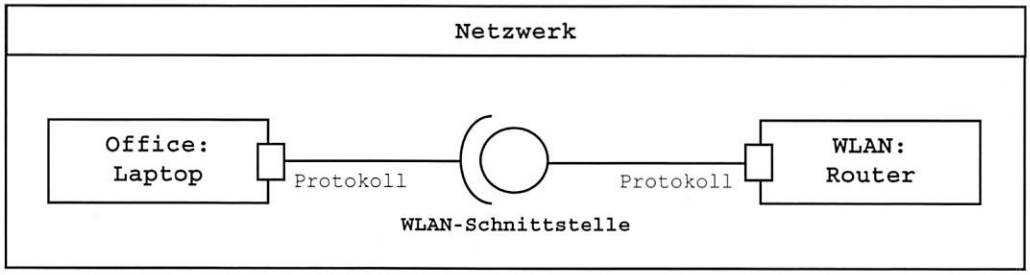


Ports und Schnittstellen

Über Ports kann eine Klasse auch Schnittstellen implementieren bzw. nutzen. Schnittstellen wurden bereits im Kapitel über Klassendiagramme behandelt.

Beispiel:

Ein Laptop nutzt eine WLAN-Schnittstelle, die ein WLAN-Router anbietet und vorher implementiert hat.

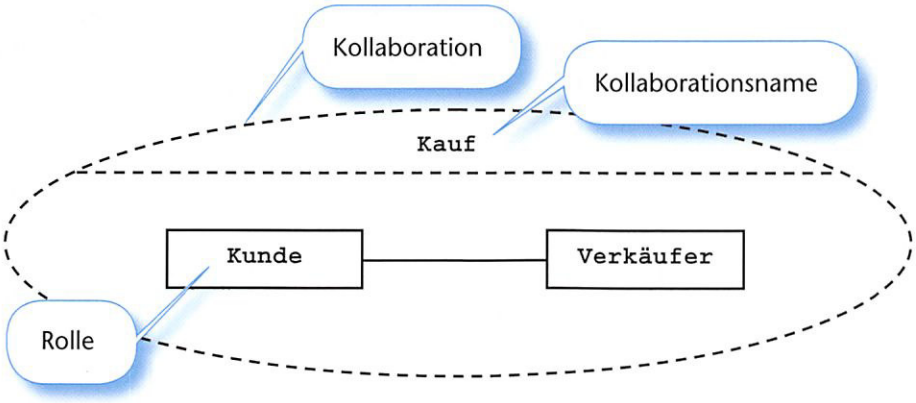


Kollaborationen

Mit Kollaborationen kann die Kommunikation zwischen Objekten beschrieben werden. Die Objekte treten in der Kollaboration in einer Rolle auf. In verschiedenen Kollaborationen können Objekte derselben Klasse auch mit verschiedenen Rollen auftreten. Die Rollen werden durch Konnektoren verbunden, um die Kommunikation zwischen ihnen darzustellen. Die Darstellung einer Kollaboration ist eine Ellipse mit gestrichelten Linien.

Beispiel:

Zwei Objekte einer Klasse `Person` haben die Rollen `Kunde` und `Verkäufer`. In der Kollaboration „Kauf" wird die Beziehung dargestellt.

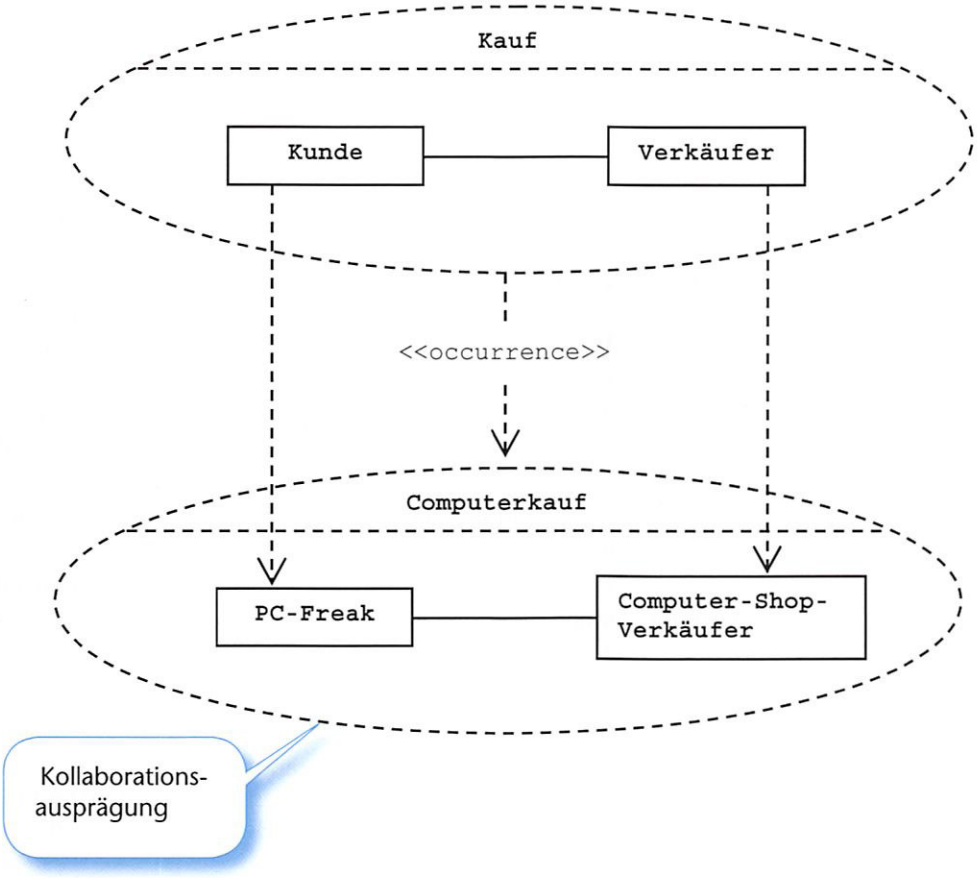


Kollaborationsausprägung

Ein bestehende Kollaboration kann weiter spezifiziert werden, um Merkmale besonders hervorzuheben. Damit erhält man eine andere Ausprägung der bestehenden Kollaboration.

Beispiel:

Die Kollaboration „Kauf" wird zu einer Kollaboration „Computerkauf" ausgeprägt. Dabei wird durch die `<<occurrence>>` -Beziehung die Ausprägung deutlich gemacht.



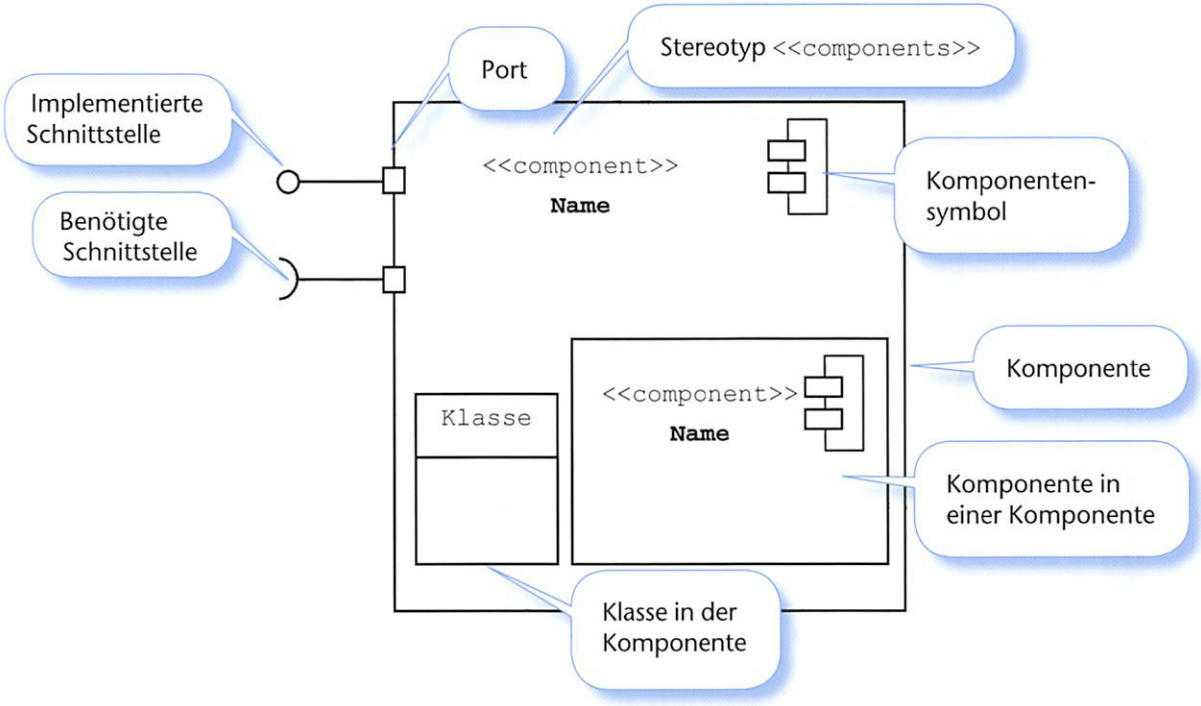
8.1.2 Das Komponentendiagramm

Die Entwicklung großer Softwaresysteme beinhaltet nicht nur viele Klassen, sondern kann auch in verschiedene Komponenten aufgeteilt werden (siehe auch Architekturmuster in Kapitel 1). Die Darstellung aller beteiligten Klassen mit einem einzigen Klassendiagramm würde sehr schnell den Rahmen sprengen. Das Komponentendiagramm ermöglicht deshalb einen Blick auf das ganze Softwaresystem. Dabei können einzelne Komponenten, Schnittstellen, Klassen und sogenannte

Artefakte dargestellt werden. Einzelne Komponenten können wiederum aus Komponenten (Klassen, Schnittstellen, Artefakten) bestehen.

Darstellung einer Komponente

Eine Komponente kann aus einer bis beliebig vielen Klassen bestehen oder auch ein ganzes Softwaresystem darstellen. Sie kommuniziert über Ports und entsprechende Schnittstellen mit anderen Komponenten, Klassen oder Artefakten.



Weitere Stereotype für Komponenten

Neben dem Stereotyp `<<component>>` kann eine Komponente auch mit anderen Stereotypen bezeichnet werden. Diese anderen Stereotype spezifizieren die Komponente noch näher. Die folgende Tabelle zeigt einige wichtige Stereotype:

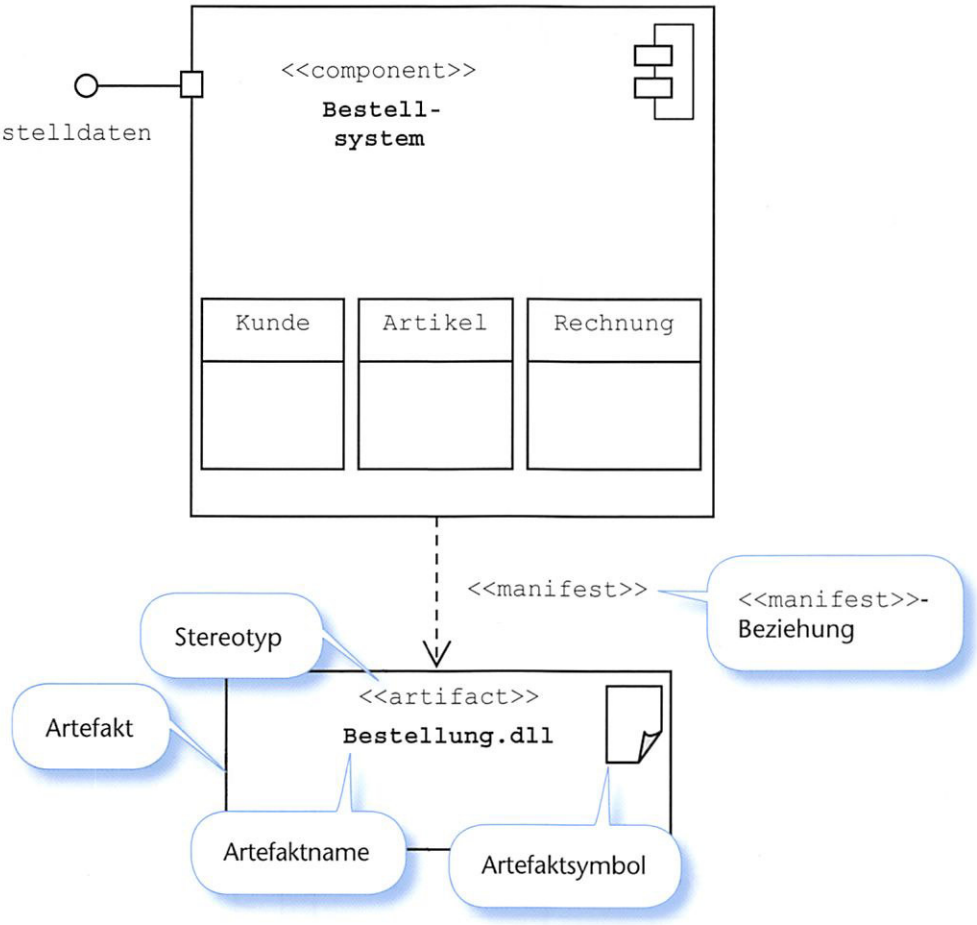
Stereotyp	Beschreibung
`<<subsystem>>`	Eine Komponente wird mit dem Stereotyp `<<subsystem>>` gekennzeichnet, wenn die Komponente zwar ein eigenes abgeschlossenes System darstellt, aber eine Art Untersystem (Bauteil) eines komplexeren Systems ist.
`<<service>>`	Eine `<<service>>`-Komponente stellt Dienste für andere Komponenten zu Verfügung.
`<<specification>>` und `<<implements>>`	Eine `<<specification>>`-Komponente deklariert nur Schnittstellen. Die Implementierung erfolgt in der entsprechenden `<<implements>>`-Komponente.

Die Artefakte

Bei der Entwicklung eines Softwaresystems entstehen konkrete Bestandteile wie Datenbanktabellen, Quellcode-Dateien oder natürlich auch UML-Diagramme. Solche (physisch existenten) Bestandteile werden **Artefakte** genannt. Sie stehen zu den Komponenten in einer Beziehung – der sogenannten `<<manifest>>`-Beziehung.

Beispiel:

Die Komponente „Bestellsystem" stellt eine komplette Abwicklung von Bestellungen zur Verfügung. Sie manifestiert sich in dem Artefakt „Bestellung.dll".



Weitere Stereotypen für Artefakte

Neben dem Stereotyp `<<artifact>>` kann ein Artefakt auch mit anderen Stereotypen bezeichnet werden. Diese anderen Stereotypen spezifizieren das Artefakt noch näher. Die folgende Tabelle zeigt einige wichtige Stereotype:

Stereotyp	Beschreibung
`<<executable>>`	Das `<<executable>>`-Artefakt steht für eine ausführbare Datei. In der Regel hat eine solche Datei die Endung *exe*.
`<<script>>`	Das `<<script>>`-Artefakt steht für eine Skript-Datei, die auf dem Server interpretiert werden muss. Das kann beispielsweise eine Perl- oder PHP-Datei sein.
`<<source>>`	Dieses Artefakt stellt eine Quellcode-Datei dar.

8.1.3 Verteilungsdiagramm

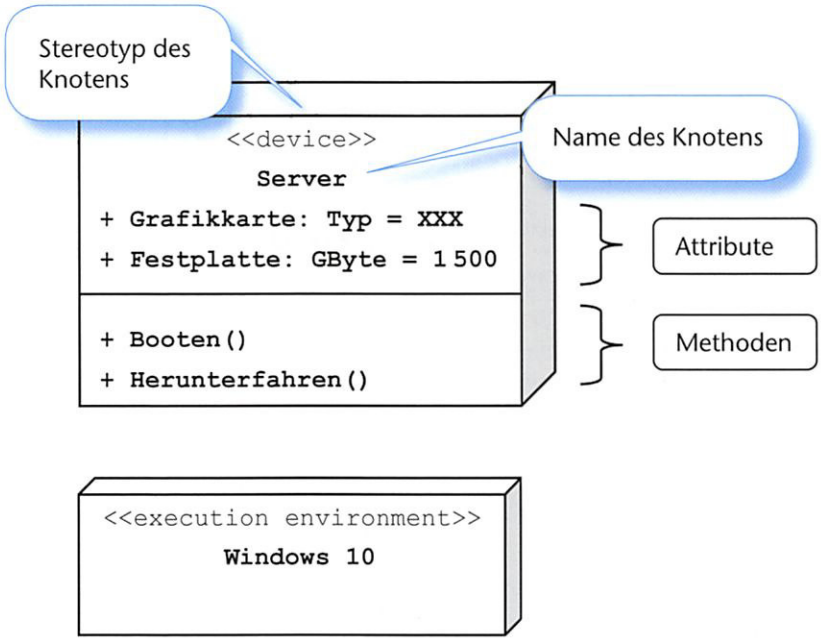
Das Verteilungsdiagramm zeigt den Zusammenhang zwischen Hardware- und Softwarekomponenten in einer bestimmten Umgebung (beispielsweise einem Firmennetzwerk mit Servern und Clients). In der Planungsphase eines Softwaresystems können dadurch die notwendigen Hardwarekomponenten identifiziert und in Beziehung mit den zu entwickelnden Softwarekomponenten gebracht werden. Damit lassen sich die Entwicklungskosten präziser einschätzen. Das Verteilungsdiagramm wird oft als Ergänzung zum Komponentendiagramm eingesetzt. Das zentrale Element des Verteilungsdiagramms ist der **Knoten**.

Der Knoten

Mit einem Knoten werden beliebige Hardwarekomponenten dargestellt. Ebenso können aber auch allgemeine Systemressourcen dargestellt werden. Einem Knoten können Attribute und Methoden zugewiesen werden. Mithilfe der Stereotypen werden verschiedene Knotenarten unterschieden:

Stereotyp	Beschreibung
`<<device>>`	Ein `<<device>>`-Knoten ist ein allgemeiner Knoten zur Darstellung einer physischen Systemressource wie einem Server.
`<<application server>>`	Der `<<application Server>>`-Knoten steht für eine Anwendung, die den Rahmen für die Ausführung weiterer Applikationen bietet (beispielsweise der WebLogic-Server von Oracle).
`<<execution environment>>`	Dieser Knoten stellt eine Umgebung für die Ausführung von Programmen dar. Die Umgebung kann beispielsweise ein Betriebssystem sein.
`<<client workstation>>`	Dieser Knoten greift auf die Applikationen zu, die der `<<application Server>>`-Knoten anbietet. Deshalb werden die beiden Knoten auch oft zusammen verwendet.
`<<mobile device>>` und `<<embedded device>>`	Diese Konten stehen für mobile Geräte (Handy etc.) oder eingebettete Ressourcen (Mikrocontroller in Haushaltsgeräten etc.).

Beispiele:

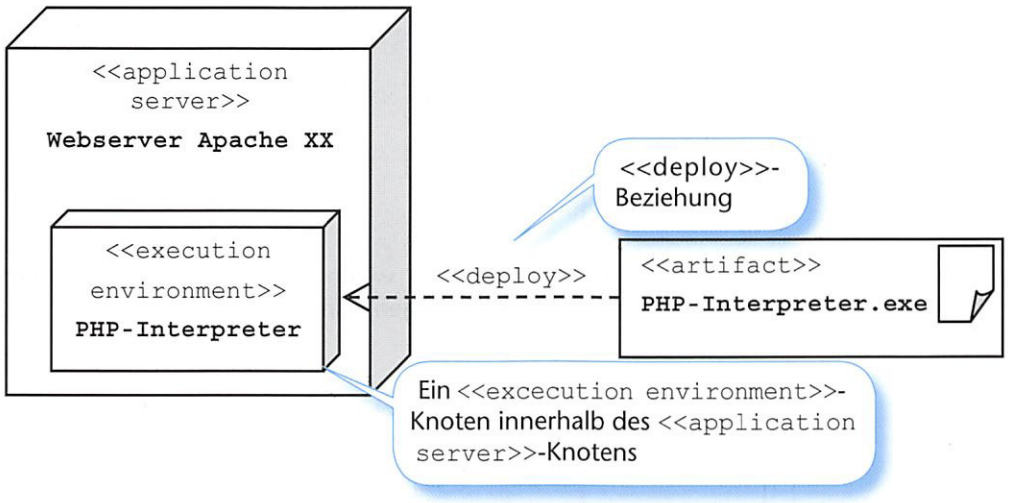


Die <<deploy>>-Beziehung

Wenn in einem Knoten ein Artefakt (beispielsweise eine Skript-Datei) eingesetzt werden soll, dann wird das durch die `<<deploy>>`-Beziehung dargestellt. Unter Deployment (deutsch: Softwareverteilung) versteht man einen Prozess zur Installation von Software auf Computern.

Beispiel:

Auf einem Webserver läuft ein PHP-Interpreter, der PHP-Skripte ausführen kann.

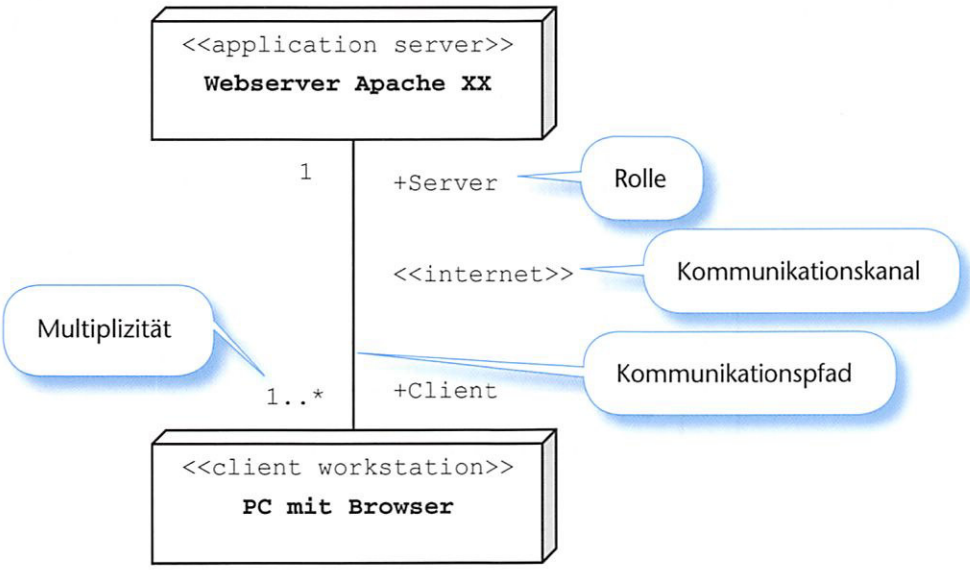


Kommunikationspfad

Zwei Knoten können über einen Kommunikationspfad miteinander kommunizieren. Dabei können Knoten Rollen einnehmen und es können Multiplizitäten angefügt werden. Die Beziehung ist deshalb vergleichbar mit einer Assoziation im Klassendiagramm.

Beispiel:

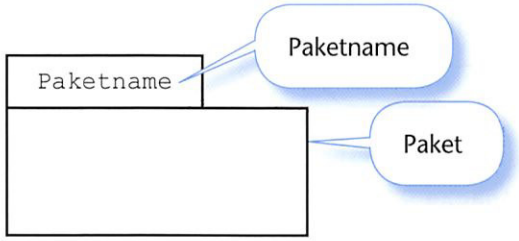
Ein Webserver kommuniziert über das Internet mit einem Browser, der auf einem Client-PC installiert ist.



8.1.4 Paketdiagramm

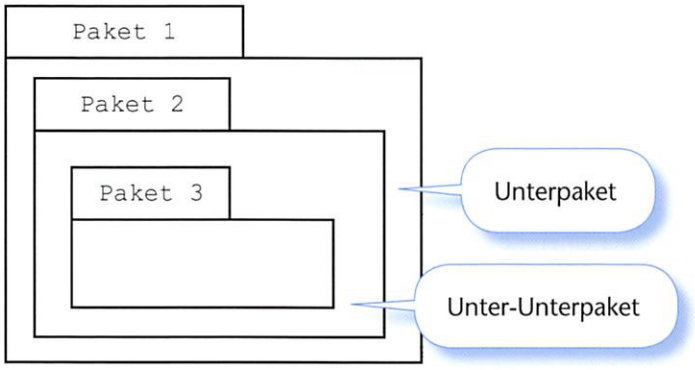
Das Paketdiagramm ermöglicht eine Gruppierung von zusammengehörenden Funktionalitäten in Form eines Pakets. Ein Paket definiert dabei auch einen Namensraum, zu dem alle diese Funktionalitäten dann gehören. Das Paketdiagramm wird eingesetzt, um eine Strukturierung des zu entwickelnden Softwaresystems zu ermöglichen. Beispielsweise können alle Klassen, die zu einer Datenbankanbindung gehören, zu einem Paket „Datenbankanbindung" zusammengefasst werden.

Darstellung eines Pakets



Hinweis:

Pakete können andere Pakete enthalten, die wiederum andere Pakete enthalten usw. Pakete müssen allerdings immer einen eindeutigen Namen haben.

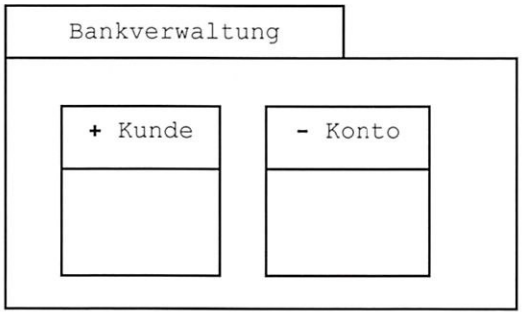


Sichtbarkeiten von Klassen in Paketen

In den Paketen können auch einzelne Klassen dargestellt werden. Dabei können die Klassen mit den bekannten Sichtbarkeitsmodifizierern (`private`, `public` und `protected`) versehen werden. Wenn nichts angegeben wird, dann handelt es sich um eine `public`-Klasse.

Beispiel:

Ein Paket `Bankverwaltung` beinhaltet die Klassen `Kunde` und `Konto`. Die Klasse `Kunde` ist öffentlich, da sie auch von anderen Paketen genutzt werden soll. Die Klasse `Konto` hingegen soll nur innerhalb des Pakets `Bankverwaltung` genutzt werden und ist deshalb als privat deklariert.

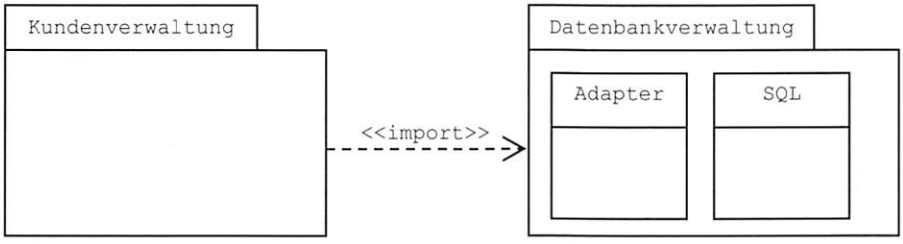


Beziehungen zwischen Paketen

Zwischen Paketen können verschiedene Beziehungen definiert werden, die dafür sorgen, dass die Elemente eines Paketes in einem anderen verwendbar (zugreifbar) sind. Die folgenden Beispiele zeigen die verschiedenen Möglichkeiten und deren Konsequenzen.

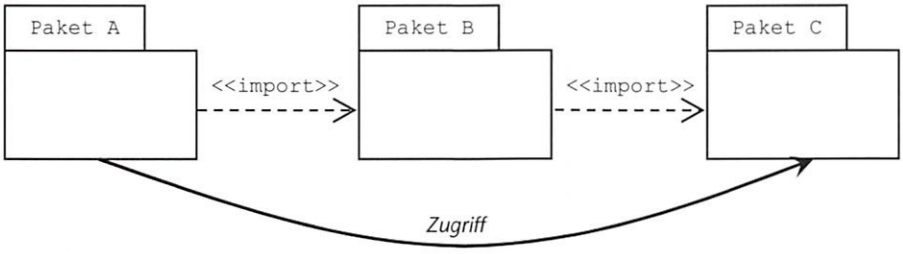
Beispiel 1: <<import>>-Beziehung

Die `<<import>>`-Beziehung sorgt dafür, dass die Elemente eines Paketes in einem anderen Paket so zugreifbar sind, als wären sie dort selbst mit `public` deklariert worden. In dem folgenden Beispiel importiert das Paket „Kundenverwaltung" das Paket „Datenbankverwaltung" und kann damit auf die öffentlichen Klassen `Adapter` und `SQL` zugreifen.



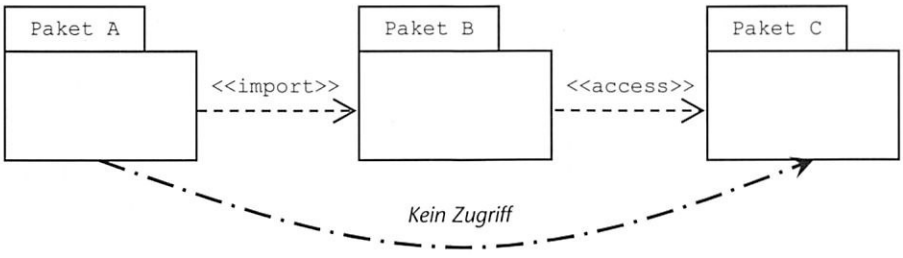
Hinweis:

Die `<<import>>`-Beziehung sorgt dafür, dass importierte Klassen von einem weiteren Paket ebenfalls importiert und damit aufrufbar sind.



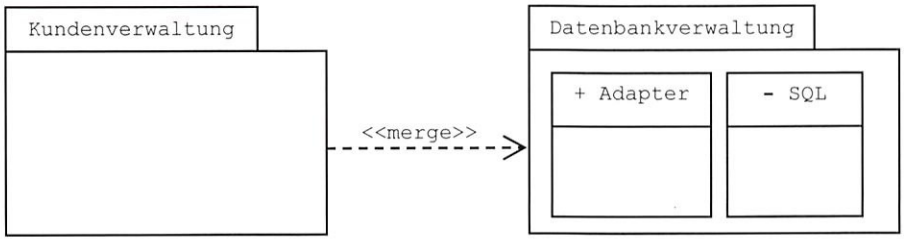
Beispiel 2: <<access>>-Beziehung

Die `<<access>>`-Beziehung sorgt dafür, dass die Elemente eines Paketes in einem anderen Paket nur so zugreifbar sind, als wären sie dort mit `private` deklariert worden. Paket A hat in dem folgenden Beispiel keinen Zugriff auf die Elemente von Paket C.



Beispiel 3: <<merge>>-Beziehung

Die `<<merge>>`-Beziehung sorgt dafür, dass nur die öffentlichen Elemente eines Paketes in einem anderen Paket zugreifbar sind. In dem folgenden Beispiel kann in dem Paket „Kundenverwaltung" nur auf die Klasse `Adapter` zugegriffen werden, da die `SQL`-Klasse als `private` deklariert wurde.

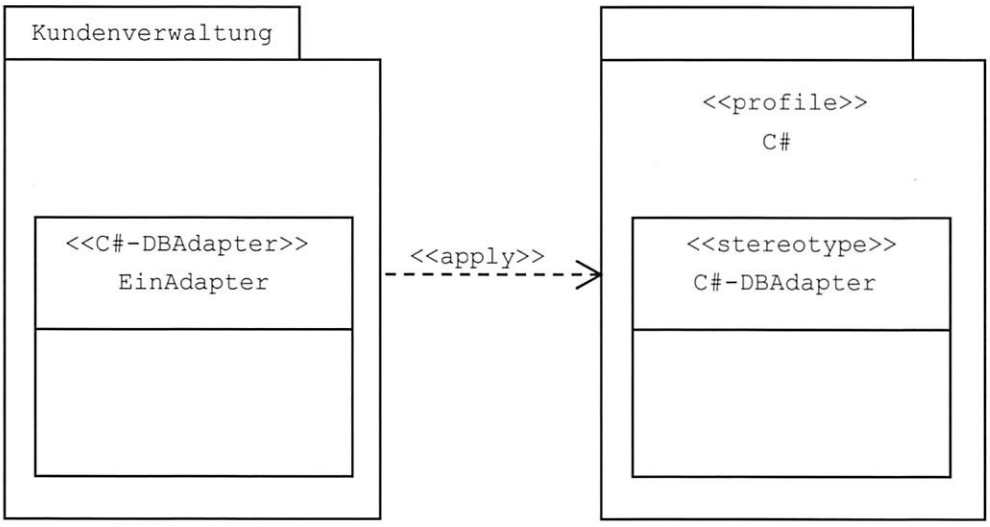


8.1.5 Profildiagramm

Das Profildiagramm wurde mit der UML-Version 2.2 eingeführt und ist deshalb das jüngste Diagramm innerhalb der UML. Es dient zur Darstellung von sogenannten Profilen, die im Prinzip eine Möglichkeit eröffnen, die Sprache UML auf seine eigenen spezifischen Bedürfnisse anzupassen. Beispielsweise kann durch diese Profile ein Anpassung an eine bestimmte Implementierung vorgenommen werden. Die UML wird dabei nicht verändert, sondern durch eigene Stereotype erweitert. Die Entwickler dieser Diagramme bewegen sich damit auf einer übergeordneten Ebene (Meta-Ebene) und entwickeln dadurch spezifische „UML-Sprachvarianten". Solche „Meta-Modelle" können beispielsweise bei der Entwicklung von CASE-Tools hilfreich sein. Profile werden in Form von Paketen dargestellt und mit dem Stereotyp `<<profile>>` versehen. Andere Pakete können durch diese Profile erweitert werden. Dazu wird die `<<apply>>`-Beziehung genutzt.

Beispiel:

Das Paket „Kundenverwaltung" wird durch das Profil „C#" erweitert. Der neu angelegte Stereotyp „C#-DBAdapter" wird in der Kundenverwaltung verwendet – in Form der Klasse `EinAdapter`.



8.2 Verhaltensdiagramme

8.2.1 Zustandsdiagramm

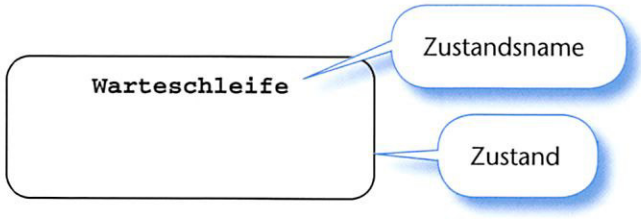
Das Zustandsdiagramm ähnelt dem Aktivitätsdiagramm mit seiner Symbolik sehr. Es beschreibt ebenso wie das Aktivitätsdiagramm das dynamische Verhalten eines Systems. Allerdings wird der Schwerpunkt auf die Zustände und nicht auf die Aktionen gelegt. Damit beschreibt das Diagramm eher den Zustand eines Objektes (die Werte der Attribute) als die Aktionen, die das Objekt (über Methoden) ausführen kann. Das Zustandsdiagramm basiert auf den sogenannten **endlichen Automaten (EA)**, die auch Zustandsmaschinen genannt werden. Ein solcher Automat heißt endlich, weil er nur endlich viele Zustände annehmen kann.

Der Zustand

Ein Zustand beschreibt eine Situation mit genau definierten Bedingungen. Der Zustand wird durch ein Rechteck mit abgerundeten Ecken dargestellt.

Beispiel:

Eine Benutzeroberfläche ist in einer sogenannten Warteschleife und wartet auf die Aktion eines Benutzers.

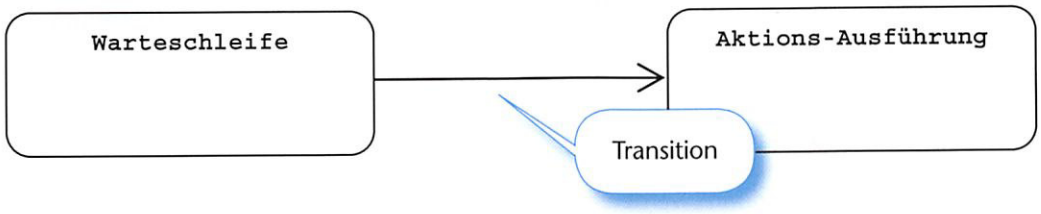


Transitionen

Die Übergänge von einem Zustand in einen anderen Zustand werden Transitionen genannt. Sie werden mit einem gerichteten Pfeil gekennzeichnet, der die Richtung des Übergangs beschreibt.

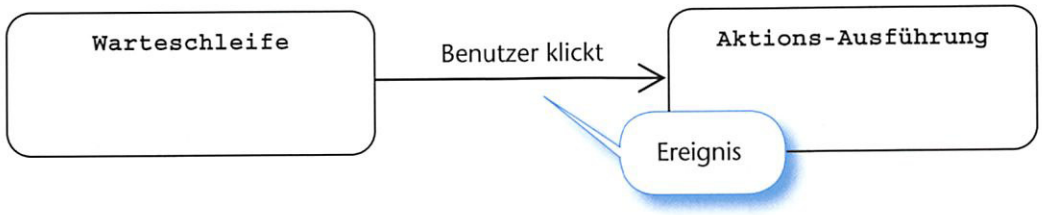
Beispiel:

Der Zustand „Warteschleife" der Benutzeroberfläche geht in den Zustand „Aktions-Ausführung" über.



Hinweis:

Die Transitionen werden in der Regel durch Ereignisse (*Events*) ausgelöst. Dabei können verschiedene Ereignisse unterschieden werden (beispielsweise einfache Signale oder zeitgesteuerte Ereignisse). Die Ereignisse werden einfach über den Transitionspfeil geschrieben.



Überwachte Bedingungen (Guards)

Die Ereignisse können zusätzlich durch Bedingungen ergänzt werden – durch die sogenannten **Guards**. Die Transition wird nur ausgeführt, wenn die Bedingung erfüllt ist (also den booleschen Wert `true` hat).

Beispiel:

Der Zustand „Warteschleife" der Benutzeroberfläche geht nur in den Zustand „Aktions-Ausführung" über, wenn die Bedingung „Datenbankverbindung verfügbar" zutrifft (also den booleschen Wert `true` hat).



Effekte

Bei einer Transition können zusätzliche Aktionen ausgelöst werden, die auch an Bedingungen (Guards) geknüpft werden können. Diese zusätzlichen Aktionen werden Effekte genannt und nach dem Ereignis (oder auch dem Guard) angegeben (getrennt durch einen Schrägstrich).

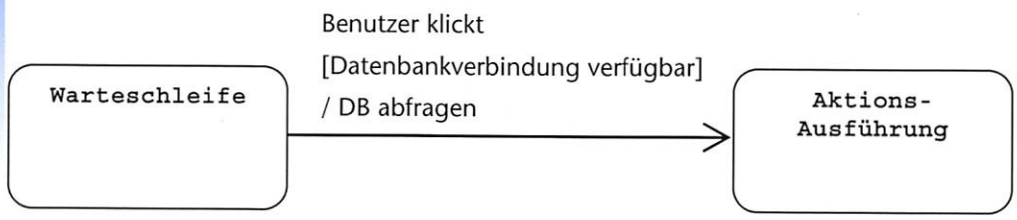
Beispiel 1:

Nach dem Ereignis „Benutzer klickt" wird der Effekt „Datenbank abfragen" ausgeführt.



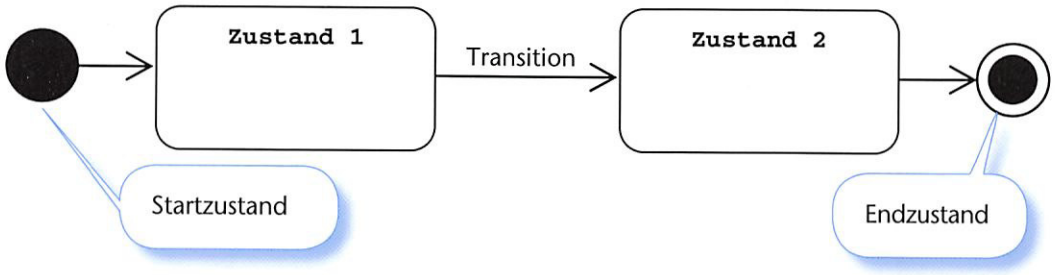
Beispiel 2:

Nach dem Ereignis „Benutzer klickt" wird der Effekt „Datenbank abfragen" ausgeführt, allerdings nur, wenn die Bedingung „Datenbankverbindung verfügbar" erfüllt ist.



Start- und Endzustände

So wie im Aktivitätsdiagramm werden auch im Zustandsdiagramm Start- und Endzustände symbolisiert. Dabei kann es durchaus sein, dass ein Zustandsdiagramm keinen Endzustand hat, da es in einer Art Schleife immer wieder zu einem Anfangszustand zurückkehrt.

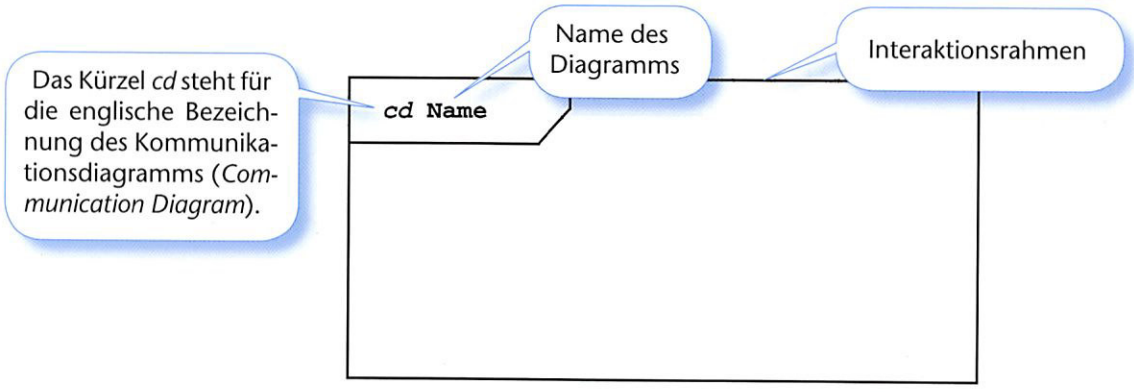


8.2.2 Kommunikationsdiagramm

Das Kommunikationsdiagramm stellt wie das Sequenzdiagramm die Kommunikation zwischen Objekten dar. Im Gegensatz zum Sequenzdiagramm steht allerdings der genaue Ablauf (also die Reihenfolge der Nachrichten) nicht im Vordergrund, sondern die umfassende Darstellung der Kommunikation aller beteiligten Objekte. Daher dient das Kommunikationsdiagramm eher der Übersicht und ergänzt dadurch das Sequenzdiagramm sehr gut.

Der Interaktionsrahmen

Ein Kommunikationsdiagramm wird wie ein Sequenzdiagramm innerhalb eines Interaktionsrahmen gezeichnet. Es bekommt einen eindeutigen Namen. Der Vorteil eines Interaktionsrahmens ist die Weiterverwendung in einem anderen Kommunikationsdiagramm, in welchem der Interaktionsrahmen einfach eingebettet wird (ähnlich dem Aufruf eines Unterprogramms oder Moduls).

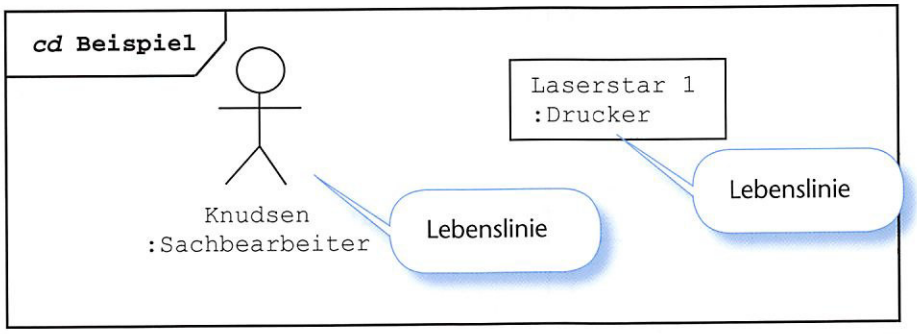


Hinweis:

Die Diagrammtypen Sequenzdiagramm, Kommunikationsdiagramm, Zeitverlaufsdiagramm und Interaktionsübersichtsdiagramm werden auch manchmal zu einer Gruppe zusammengefasst – der Gruppe der **Interaktionsdiagramme**.

Lebenslinien

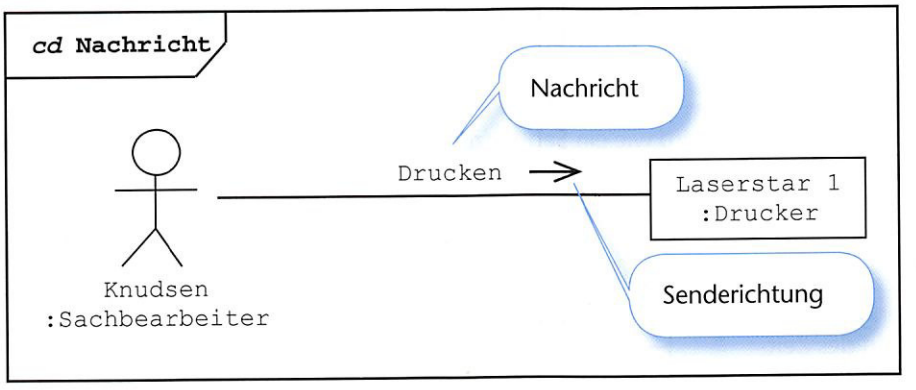
In einem Kommunikationsdiagramm werden Objekte als Lebenslinien dargestellt. Im Gegensatz zum Sequenzdiagramm besteht die Lebenslinie in dem Kommunikationsdiagramm aber nur aus der Darstellung des Objekts in Form eines Rechtecks oder Akteur-Symbols. Die Bezeichnung Lebenslinie scheint deshalb etwas irreführend zu sein – sie wurde wahrscheinlich nur aus Konsistenzgründen aus dem Sequenzdiagramm übernommen.



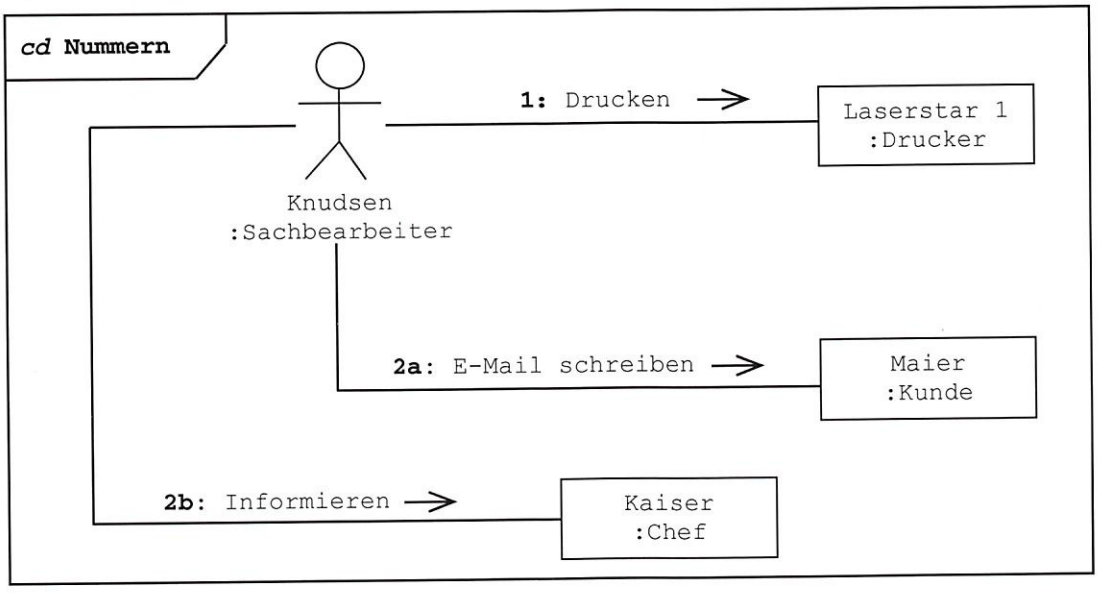
Nachrichten

Zwischen den Lebenslinien können Nachrichten gesendet werden. Dabei können auch Reihenfolgen durch Nummerierungen festgelegt werden. Es gibt aber keine eigenen Antwort-Nachrichten wie im Sequenzdiagramm.

Die Nachricht wird durch eine einfache Linie (wie bei der Assoziation) symbolisiert. Durch einen kleinen Pfeil wird die Senderichtung angegeben.



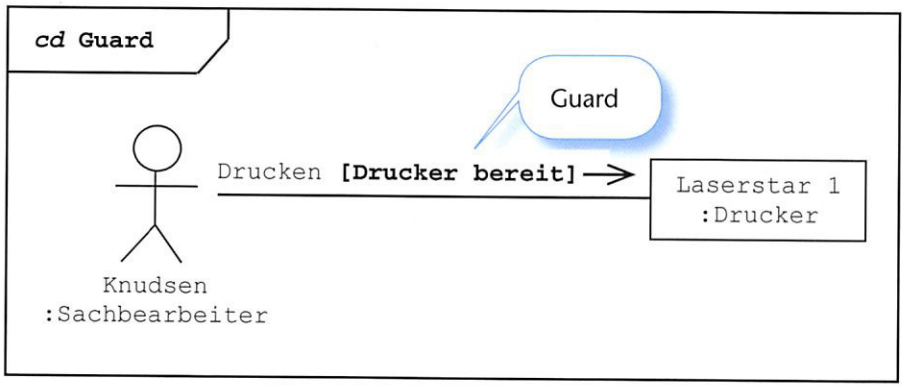
Durch eine Nummerierung können die Nachrichten in eine Reihenfolge gebracht werden, wie das folgende Diagramm zeigt:



Die Nummerierung vor der Nachricht gibt vor, welche Nachricht zuerst gesendet wird. Wenn eine Nummerierung zusätzlich mit einer Gliederung durch **Buchstaben** versehen wird, dann sollen diese Nachrichten **parallel** gesendet werden.

Hinweis:

Nachrichten können mit Bedingungen (**Guards**) versehen werden. Nur wenn die Bedingung den booleschen Wert `true` hat, wird die Nachricht versendet.

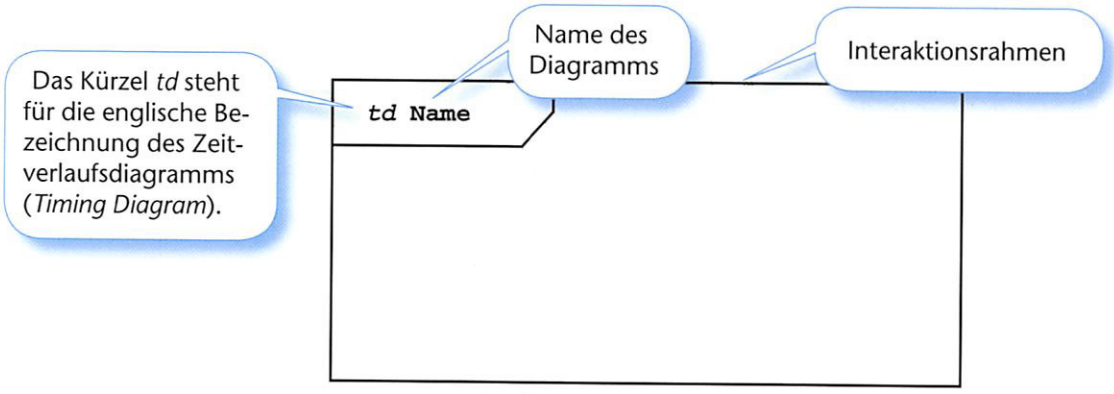


8.2.3 Zeitverlaufsdiagramm

Das Zeitverlaufsdiagramm zeigt sehr genau den Zustandswechsel von Objekten auf einer vorgegebenen Zeitachse. Damit ergänzt es einerseits das Zustandsdiagramm und andererseits das Kommunikations- bzw. Sequenzdiagramm um eine konkrete zeitliche Komponente. Die Darstellung erinnert an ein Oszilloskop aus der Elektrotechnik, welches eine optische Darstellung einer (oder mehrerer) elektrischer Spannung(en) und des zeitlichen Verlaufs auf einem Bildschirm bietet.

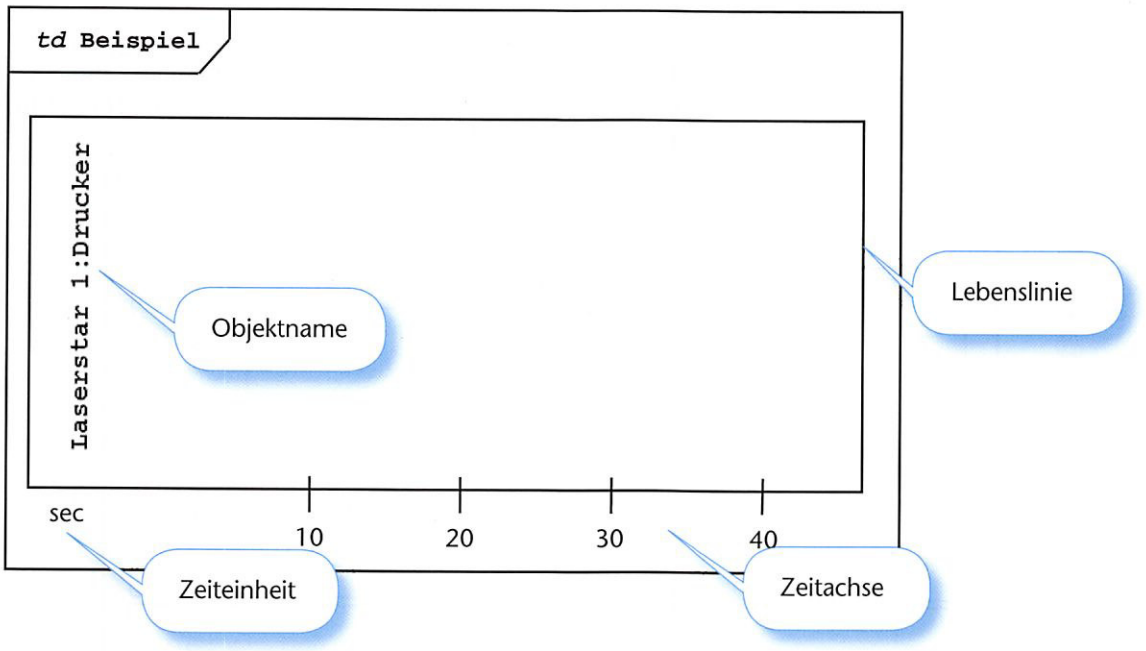
Der Interaktionsrahmen

Ein Zeitverlaufsdiagramm wird wie ein Sequenzdiagramm (oder Kommunikationsdiagramm) innerhalb eines Interaktionsrahmen gezeichnet. Es bekommt einen eindeutigen Namen.



Lebenslinien

In einem Zeitverlaufsdiagramm werden Objekte als Lebenslinien dargestellt. Eine Lebenslinie besteht aus der Bezeichnung des Objekts, einem Rahmen und einer Zeitachse (inkl. Zeiteinheit).

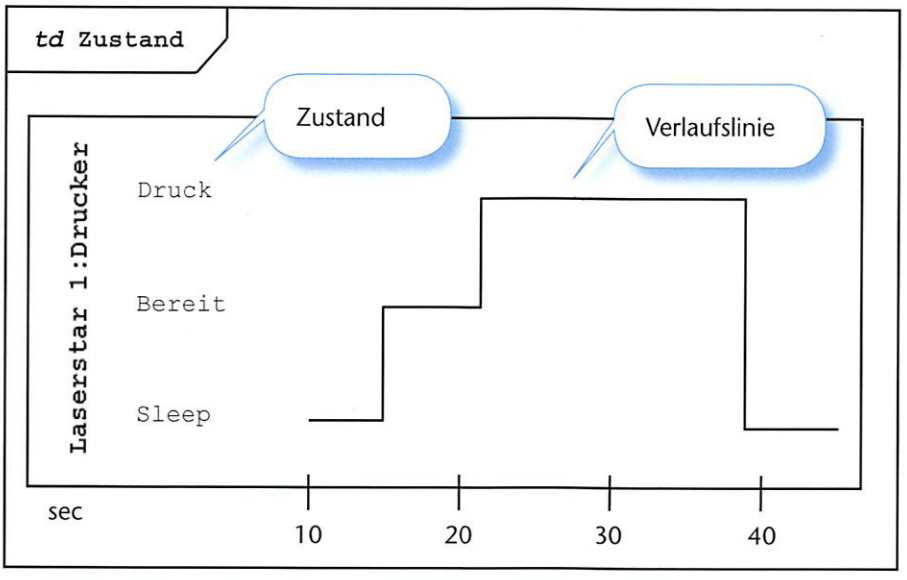


Zustände und Verlaufslinien

Die Zustände eines Objektes werden nun in einen zeitlichen Zusammenhang gebracht. Dabei zeigt die sogenannte Verlaufslinie an, zu welchem Zeitpunkt das Objekt sich in welchem Zustand befindet. Diese Darstellung erinnert dann an das oben bereits beschriebene Oszilloskop. Das folgende Beispiel zeigt das Objekt „Laserstar 1" in drei verschiedenen Zuständen, die sich im Laufe der Zeit ändern.

Beispiel:

Der Drucker „Laserstar 1" wechselt vom *Sleep*-Modus in den *Bereit*-Modus sowie in den *Druck*-Modus.

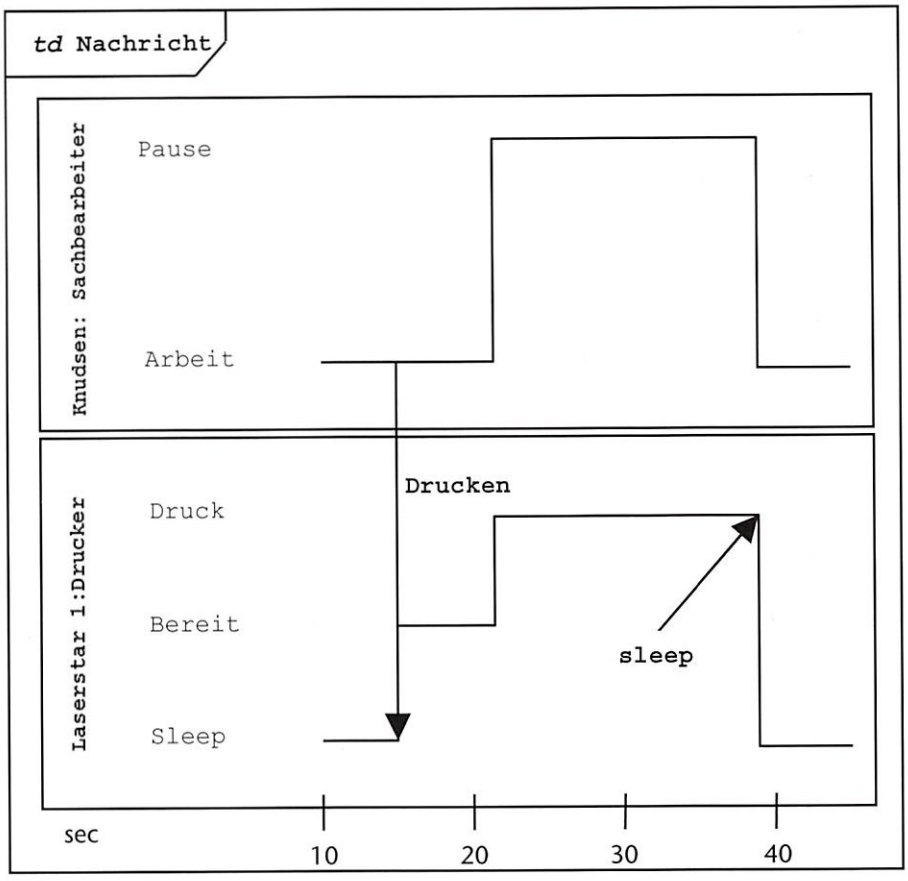


Nachrichten

In einem Zeitverlaufsdiagramm können ebenfalls Nachrichten zwischen Lebenslinien gesendet werden. Eine Nachricht wird dann im Gegensatz zum Sequenzdiagramm vertikal gesendet.

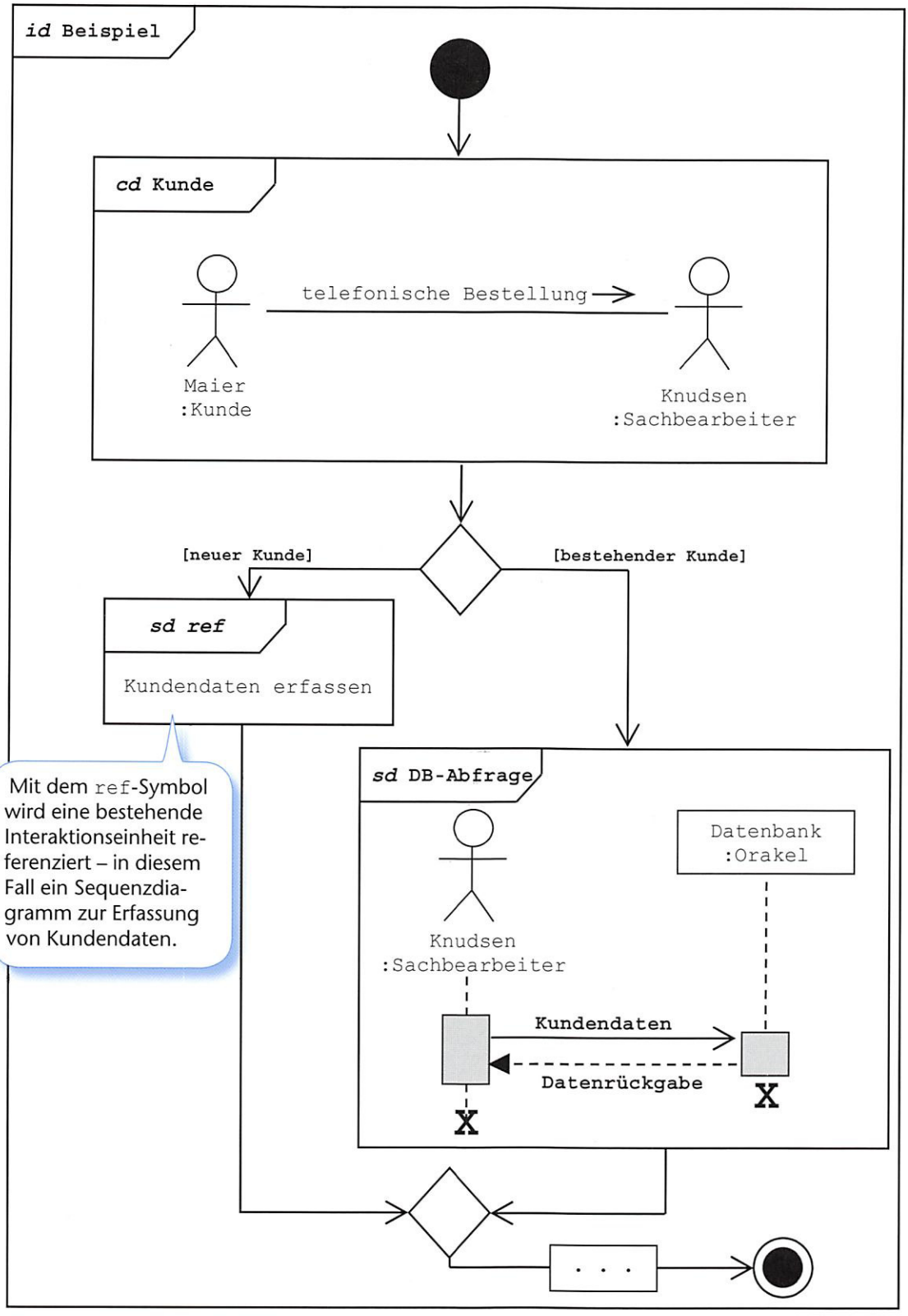
Beispiel:

Der Sachbearbeiter sendet dem Drucker die Nachricht „Drucken". Der Drucker wechselt daraufhin seinen Zustand. Nach dem Drucken sendet der Drucker sich selbst die Nachricht, wieder in den *Sleep*-Modus zu wechseln.



8.2.4 Interaktionsübersichtsdiagramm

Das Interaktionsübersichtsdiagramm kann als ein übergeordnetes Diagramm innerhalb der Gruppe der Interaktionsdiagramme bezeichnet werden. Es dient zur Darstellung von Interaktionen zwischen verschiedenen Interaktionseinheiten. Diese Einheiten können einzelne Interaktionsrahmen aus den bereits besprochenen Interaktionsdiagrammen sein (Sequenzdiagramm, Kommunikationsdiagramm und Zeitverlaufsdiagramm). Die einzelnen Interaktionen werden mithilfe der **Symbolik** des **Aktivitätsdiagramms** in einen größeren Rahmen integriert. Das folgende Beispiel zeigt das Zusammenspiel verschiedener Interaktionen im Zusammenhang mit einer Kundenbestellung:



Teil 2
Aufgabenpool

1 Aufgaben zu den Grundbegriffen UML / OOP 126
2 Aufgaben zum Anwendungsfalldiagramm 127
3 Aufgaben zum Klassendiagramm 129
4 Aufgaben zum Objektdiagramm 131
5 Aufgaben zum Sequenzdiagramm 134
6 Aufgaben zum Aktivitätsdiagramm 136
7 Aufgaben zur Softwareentwicklung 138
8 Aufgaben zu den weiteren Diagrammen 138

Aufgabenpool

1 Aufgaben zu den Grundbegriffen UML / OOP

Aufgabe 1.1
Entwerfen Sie eine Skizze, die den Zusammenhang zwischen Klassen, Objekten, Attributen und Methoden anschaulich darstellt. Beachten Sie dabei auch, wie Attribute und Methoden zu den Objekten gespeichert werden.

Aufgabe 1.2
Informieren Sie sich (im Internet oder mit anderer Fachliteratur) darüber, wie Objekte konkret im Speicher angelegt und auch wieder entfernt werden. Unterscheiden Sie dabei nach Programmiersprachen wie C++, C# und Java.

Aufgabe 1.3
Setzen Sie das Entwurfsmuster „Observer" für die folgende Problemstellung in die Sprache C# um:

Eine Klasse `CMesswerte` soll beliebig viele ganzzahlige Messwerte speichern. Die Klasse bietet dazu entsprechende Methoden an. Die Klasse soll dabei von zwei Ansichts-Klassen beobachtet werden. Die Ansichts-Klassen bieten verschiedene Sichten auf die Daten. Die eine Klasse soll bestimmte statistische Kennzahlen anzeigen (Minimum, Maximum und Mittelwert der Messwerte). Die andere Klasse soll die Messwerte in leserlicher Form auf dem Bildschirm darstellen. Sobald die Messwerte sich ändern oder neue Messwerte eingefügt werden, sollen die Beobachter benachrichtigt werden und ihre Anzeige aktualisieren.

Hinweis:

Nutzen Sie der Einfachheit halber eine Konsolenanwendung für die Darstellung.

Ein mögliches „Hauptprogramm" in C# könnte so aussehen:

```
static void Main(string[] args)
{
   CMesswerte messwerte = new CMesswerte();
   CStatistik statistik = new CStatistik();
   CAnzeige anzeige = new CAnzeige();
   messwerte.AddBeobachter(statistik);
   messwerte.AddBeobachter(anzeige);
   System.Console.WriteLine("**************Neuer
                            Wert**************");
   messwerte.AddWert(10);
   System.Console.WriteLine("**************Neuer
                            Wert**************");
   messwerte.AddWert(15);
   System.Console.WriteLine("**************Neuer
                            Wert**************");
   messwerte.AddWert(5);
}
```

Eine mögliche Bildschirmausgabe könnte so aussehen:

```
C:\WINDOWS\system32\cmd.exe
**************Neuer Wert**************
Statistik-Beobachter:
Maximum: 10
Minimum: 10
Mittelwert: 10

Anzeige-Beobachter:
Wert Nr: 1 = 10

**************Neuer Wert**************
Statistik-Beobachter:
Maximum: 15
Minimum: 10
Mittelwert: 12,5

Anzeige-Beobachter:
Wert Nr: 1 = 10
Wert Nr: 2 = 15

**************Neuer Wert**************
Statistik-Beobachter:
Maximum: 15
Minimum: 5
Mittelwert: 10

Anzeige-Beobachter:
Wert Nr: 1 = 10
Wert Nr: 2 = 15
Wert Nr: 3 = 5

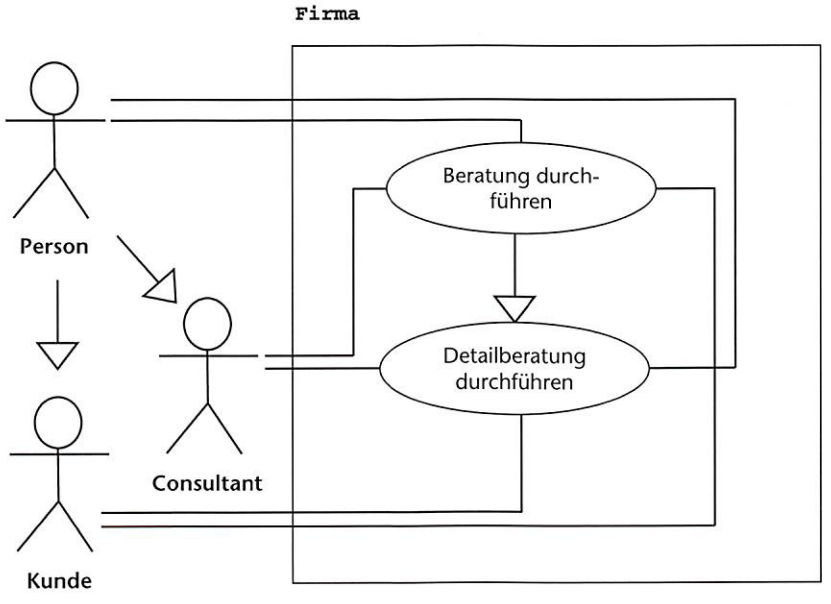
Drücken Sie eine beliebige Taste . . .
```

2 Aufgaben zum Anwendungsfalldiagramm

Aufgabe 2.1

Eine Firma für objektorientierte Softwareentwicklung wurde damit beauftragt, ein Softwaresystem für die Abwicklung von Kundenberatungen für eine große Consulting-Agentur zu entwickeln. In einem ersten Schritt wurde ein Anwendungsfalldiagramm erstellt.

Analysieren Sie das Diagramm auf inhaltliche Korrektheit und korrigieren Sie es gegebenenfalls.



Aufgabe 2.2

Entwickeln Sie ein Anwendungsfalldiagramm zu der folgenden Problemstellung:

Ein Schulungsunternehmen bietet EDV-Kurse in verschiedenen Bereichen an. Jeder Kurs kann maximal 14 Teilnehmer haben. Es gibt fest angestellte Dozenten und freie Mitarbeiter, die als Dozenten tätig sind.

Falls es zu Spannungen zwischen Dozenten und Teilnehmern kommt, bietet der Schulungsleiter ein verbindliches Vermittlungsgespräch an.

Muss der Kurs aus Krankheitsgründen des Dozenten abgebrochen werden, dann erhalten die Teilnehmer Gutscheine für spätere Kurstage.

Aufgabe 2.3

Ausgangssituation:

Der Kfz-Händler Knauser möchte ein neues EDV-System einführen. Das neue System soll sowohl die Kundenberatung als auch den Verkauf von neuen Kfz abbilden. Bislang wurden Beratung und Verkauf mit einem Formular handschriftlich dokumentiert.

Analysieren Sie das folgende Formular und erstellen Sie ein Anwendungsfalldiagramm, welches die gesamte Problematik erfasst.

Kundenberatung und Verkauf

Kfz-Knauser
Autostraße 12
12345 Autostadt
Tel: 123-321

Verkäufer: _____

Kundenname: _____
Kundenadresse: _____
Kundentelefon: _____

Beratung des o. a. Verkäufers durchgeführt am: _____

Probefahrt vereinbart: ja ◯ nein ◯

Falls ja:

- Mechaniker informieren zwecks Vorbereitung
- Temporäre Versicherung abschließen
  - ▸ Versicherungsberater informieren
    - ▸ Kunde an weiterer Versicherung interessiert: ja ◯ nein ◯

Kaufvertrag anvisiert: ja ◯ nein ◯

Falls ja:

- Filialleiter informieren zwecks Rabattverhandlung mit Kunden
- Mechaniker informieren zwecks Vorbereitung möglicher Serviceleistungen

3 Aufgaben zum Klassendiagramm

Aufgabe 3.1
Vervollständigen Sie die folgenden Klassendiagramme bzgl. Multiplizitäten, Leserichtung, Navigierbarkeit usw. Entscheiden Sie sich für Assoziation, Aggregation oder Komposition.

a)



b)



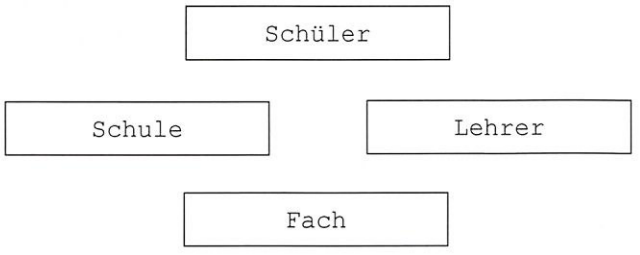
c)



d)



e)



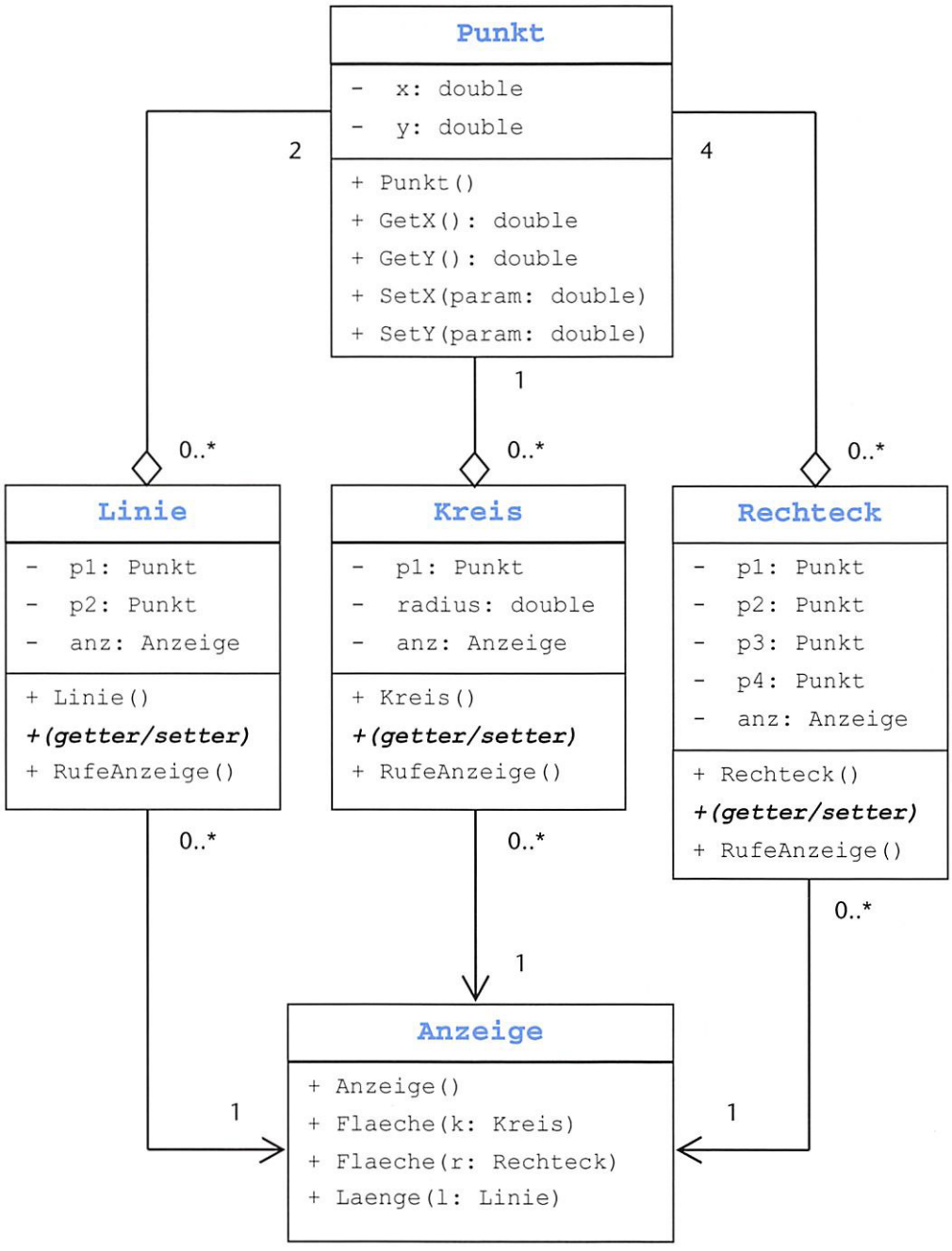
Aufgabe 3.2
Entwickeln Sie ein Klassendiagramm zu der folgenden Problemstellung:

In einem Softwaresystem sollen Autoren, Bücher und Verlage verwaltet werden. Ein Autor kann verschiedene Bücher schreiben, wobei ein Buch auch durchaus von mehreren Autoren geschrieben werden kann. Ein Verlag hat viele Bücher im Programm. Ein Buch wird aber nur von einem Verlag herausgebracht. Innerhalb des Verlags werden die Bücher nach Themengebieten eingeteilt. Für jedes Themengebiet ist genau ein Mitarbeiter des Verlags zuständig. Kein Mitarbeiter soll allerdings für mehr als drei Themengebiete verantwortlich sein.

Finden Sie jeweils einige geeignete Attribute und Methoden zu den Klassen.

Aufgabe 3.3
Ausgangssituation:

Eine Softwarefirma plant die Entwicklung einer neuen Software für den Einsatz im Mathematikunterricht. Ein Aspekt der Software soll die Darstellung von geometrischen Grundformen sein. Mithilfe dieser Funktionalität können geometrische Sachverhalte dynamisch dargestellt werden, beispielsweise die Darstellung einer Tangente an einem Kreis. In einer ersten Umsetzung soll die Erfassung von geometrischen Grundformen implementiert werden. Dazu liegt ein Klassendiagramm aus der objektorientierten Analyse OOA vor. Setzen Sie das Klassendiagramm in C# um.

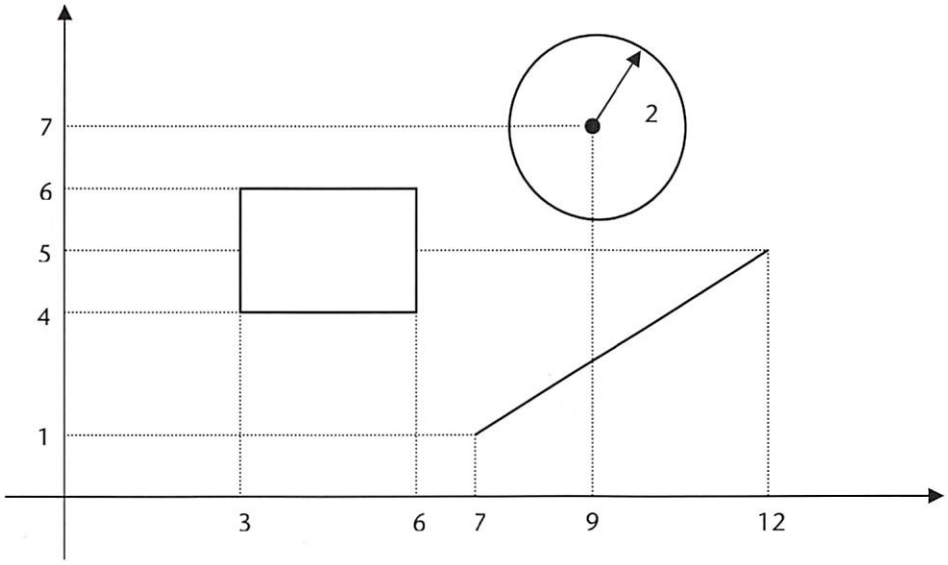


Hinweis:

Die Objekte der Klassen `Linie`, `Rechteck` und `Kreis` aggregieren Objekte der Klasse Punkt. Weiterhin kennen die Objekte der drei Geometrie-Klassen das Objekt der Klasse `Anzeige`. Das Objekt der Klasse `Anzeige` erhält eine entsprechende Botschaft und zeigt dann die Länge einer Linie, die Fläche eines Rechtecks oder eines Kreises an.

Schreiben Sie ein Test-Hauptprogramm, in dem alle Funktionalitäten der Klassen und deren Beziehungen getestet werden können.

Beispielsweise sollen die folgenden geometrischen Grundfiguren erfasst (durch die Punkte im Koordinatensystem) sowie die korrekte Länge und die korrekten Flächen berechnet und angezeigt werden:



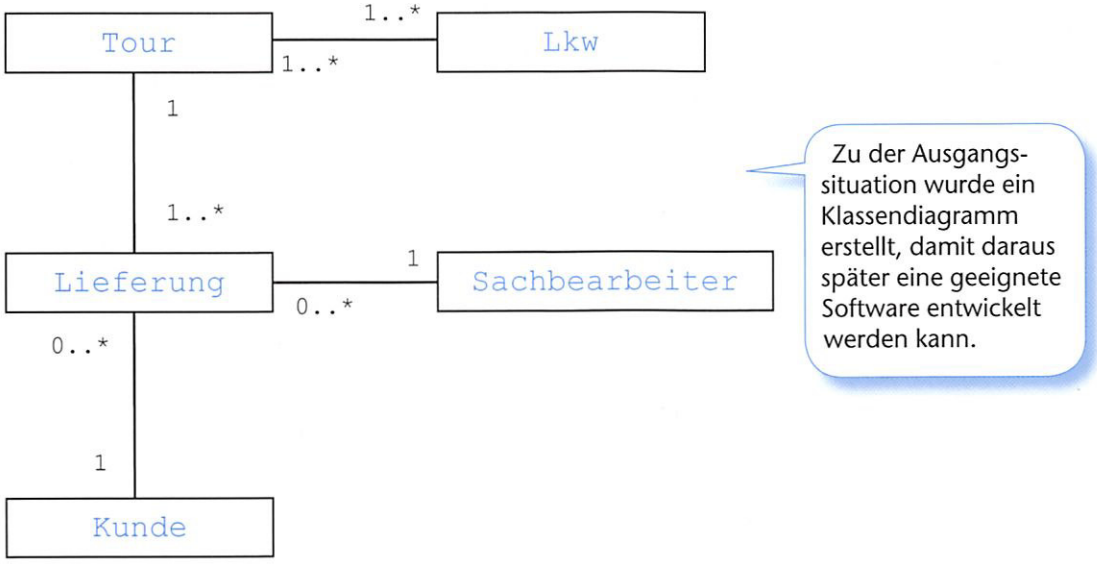
- Fläche Rechteck: $3 * 2 = 6$ (LE²) — LE....Längeneinheiten
- Fläche Kreis: $\pi * 2^2 \approx 12{,}57$ (LE²)
- Länge Linie: $\sqrt{5^2 + 4^2} \approx 6{,}40$ (LE²) — (Pythagoras benutzen)

4 Aufgaben zum Objektdiagramm

Aufgabe 4.1

Ausgangssituation:

In einer Speditionsfirma werden die Lieferungen der Kunden von verschiedenen Sachbearbeitern betreut. Jede Lieferung gehört zu einer sogenannten Tour. Eine Tour ist eine festgelegte Strecke, die von einem oder mehreren Lkws befahren wird.



Entwickeln Sie ein Objektdiagramm zu dem folgenden Szenario:

Der Kunde Herr Kaiser hat eine Lieferung nach Frankreich. Frau Schuster von der Spedition *Fast-Way* betreut diese Lieferung. Sie bucht die Lieferung auf die Tour *Hamburg-Paris*. Die Lkws mit den Kennzeichen *XX-11-12* und *XX-11-14* sind an der Tour beteiligt.

Hinweis:

Finden Sie geeignete Attribute, um den Sachverhalt und die Beziehungen zwischen den Objekten (entsprechend dem Klassendiagramm) auszudrücken.

Aufgabe 4.2

Ausgangssituation:

Die Schreinerei Holzmann GmbH wird die Rechnungserstellung in Zukunft mit einem angepassten **ERP**-System (Enterprise Resource Planning-System) erledigen. Bislang wurden die Rechnungen tatsächlich noch mit einer elektrischen Schreibmaschine auf Vordrucken ausgefüllt.

Im Rahmen der objektorientierten Analyse soll die Rechnungserstellung mithilfe eines Objektdiagramms erfasst werden. Das Objektdiagramm soll dann eine Ausgangsbasis für die Entwicklung der entsprechenden Klassen sein.

Analysieren Sie deshalb die Rechnung und erstellen Sie ein Objektdiagramm.

SCHREINEREI HOLZMANN GmbH

Frau Marlene Knudsen
Am Holzweg 12
12345 Holzhausen

Rechnung: 1251
Datum: 22.01.2019

Ihr Auftrag vom: 15.01.2019

Bezeichnung	Anzahl	Einzelpreis	Gesamtpreis
Holzstuhl Kiefer lasiert	4	239,00 €	956,00 €
Tisch Kiefer lasiert 160 × 85	1	458,00 €	458,00 €
Kommode Mahagoni furniert 80 × 60 × 35 (H × B × T) mit zwei Schubladen	1	625,00 €	625,00 €
Gesamt:			2039,00 €

In dem Rechnungsbetrag sind 19% Mehrwertsteuer enthalten (= 387,41 €)

Zahlbar bis: 22.03.2019

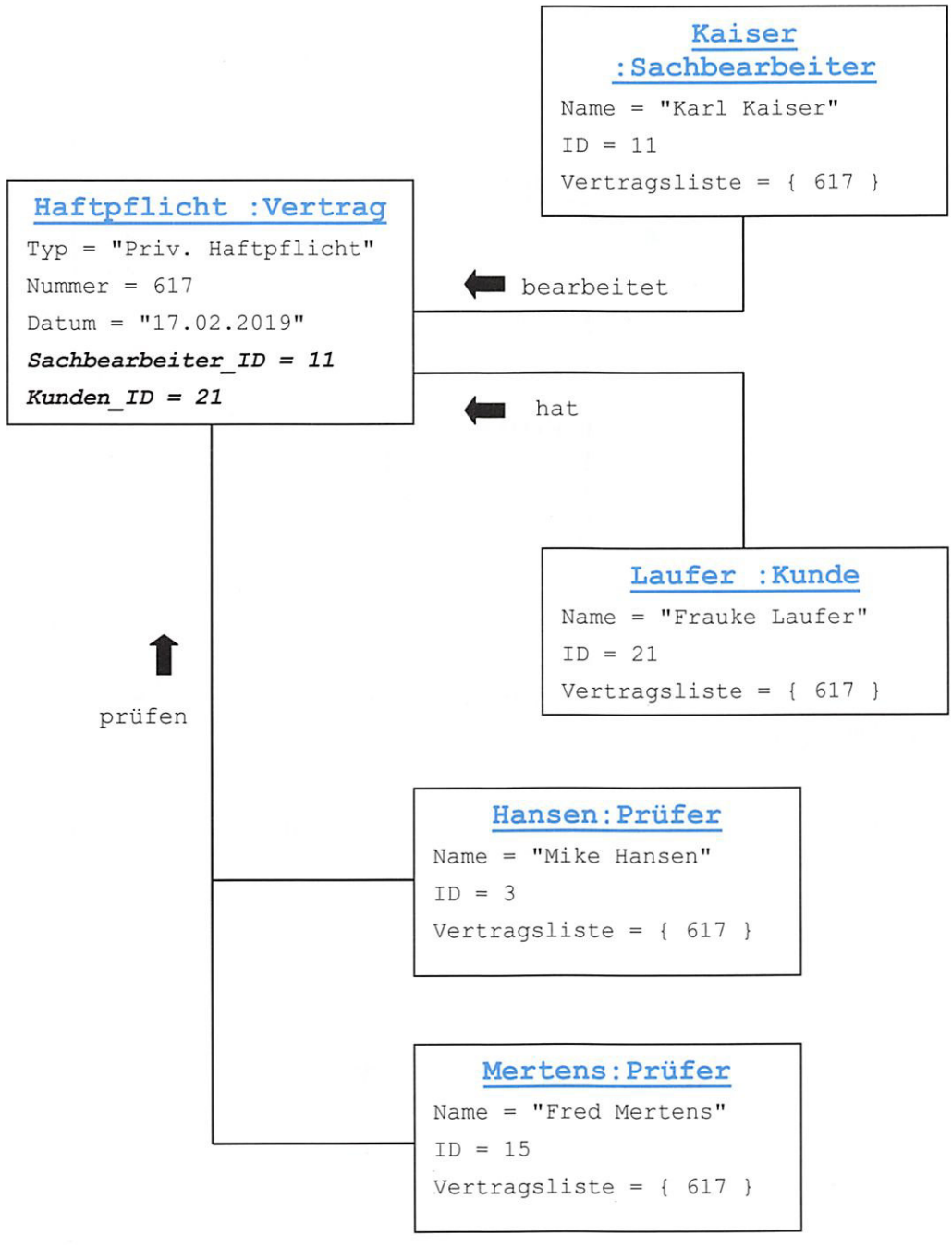
Vielen Dank für den Auftrag

Schreinerei Holzmann

Bankverbindung: KasseXY, IBAN: DEXXX BIC: ABEDXXX

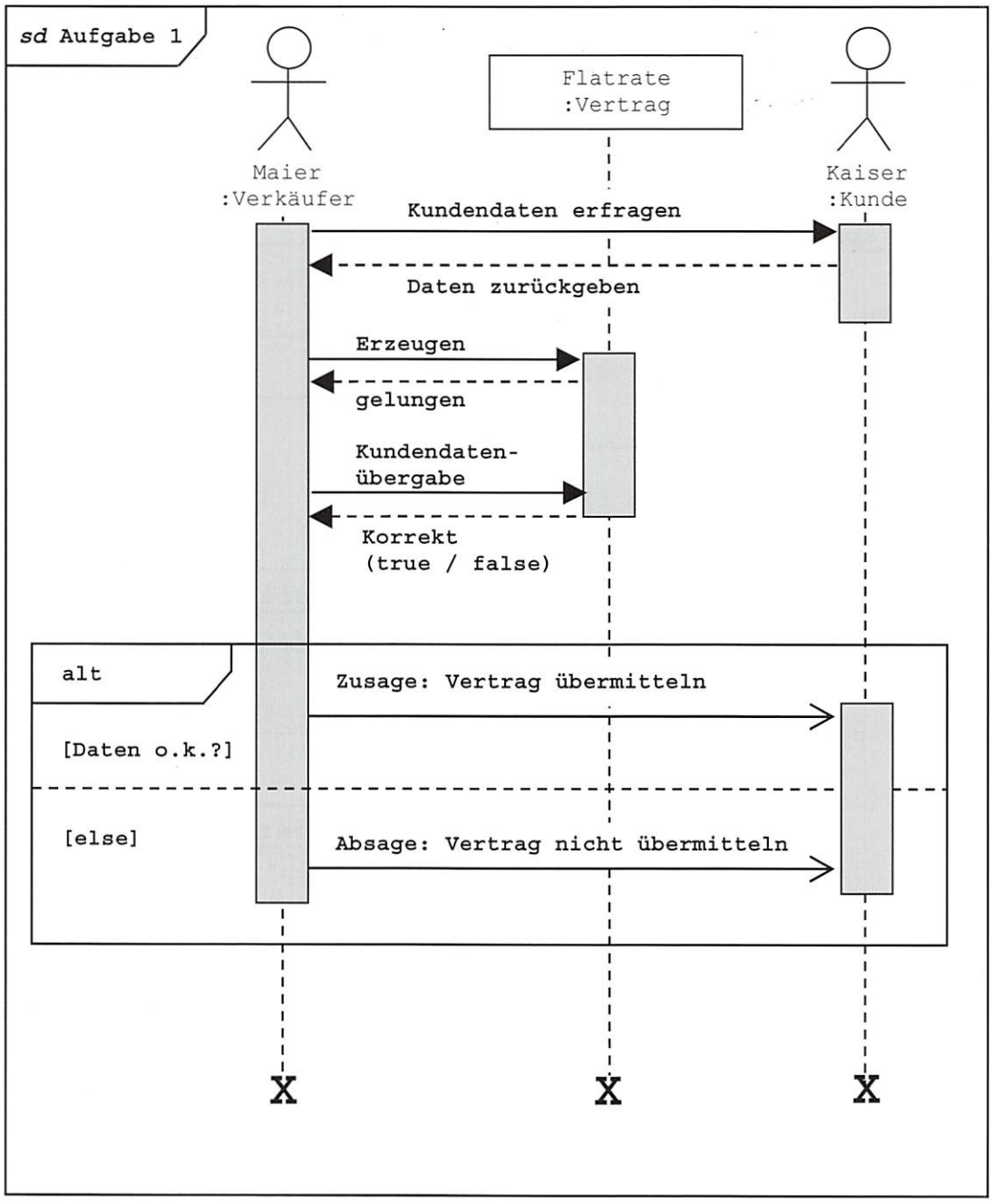
Aufgabe 4.3

Gegeben ist das folgende Objektdiagramm, welches eine Momentaufnahme in einer Versicherungsagentur darstellt. Die Mitarbeiter betreuen ihre Kunden und Verträge. Die Verträge werden zusätzlich von unabhängigen Prüfern geprüft. Analysieren Sie das Objektdiagramm und erstellen Sie anschließend ein Klassendiagramm, welches die Objekte und deren Beziehungen in den entsprechenden Klassen widerspiegelt.



5 Aufgaben zum Sequenzdiagramm

Aufgabe 5.1
Setzen Sie das folgende Sequenzdiagramm in C# um:



Hinweise:

- Erstellen Sie die entsprechenden Klassen und finden Sie einige geeignete Attribute und Methoden für die Klassen `Vertrag` und `Verkäufer`. Die Kunden-Klasse soll allerdings die folgenden Attribute haben:

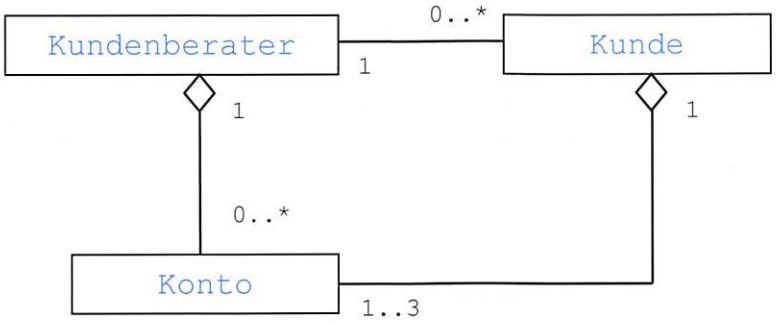
```
string name;
int kunden_ID
string geburtsdatum;
```

- Die Kundendaten müssen von der Vertrags-Klasse geprüft werden, bevor ein Vertrag zustande kommen kann. Dazu muss Folgendes geprüft werden:
  - Liegt die `kunden_ID` zwischen 1 und 99999 (inkl.)?
  - Ist das Geburtsdatum ein gültiges Datum (auch Schaltjahre einbeziehen)?
  - Ist der Kunde mindestens 18 Jahre alt (auf das Vertragsdatum bezogen)?

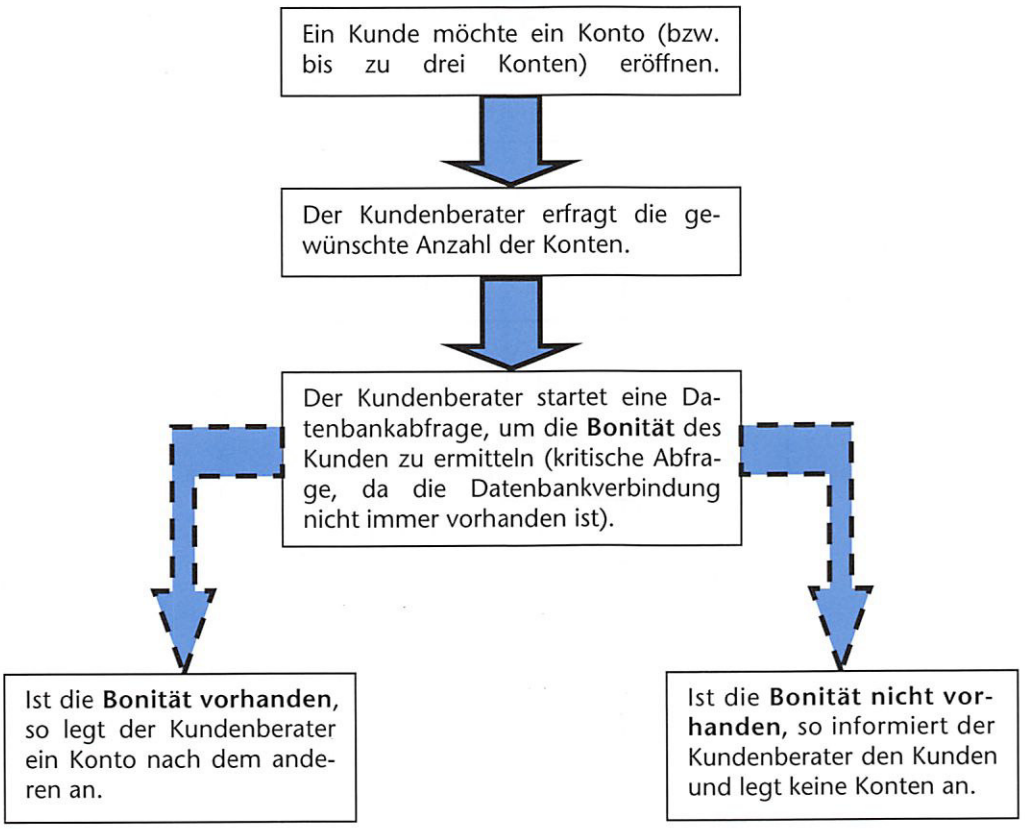
Aufgabe 5.2

Ausgangssituation:

Für eine Bank soll ein Softwaresystem entwickelt werden. Der Zusammenhang zwischen Kunden, Beratern und Konten ist bereits in einem Klassendiagramm erfasst worden:



Entwickeln Sie ein Sequenzdiagramm zu dem folgenden Szenario:



6 Aufgaben zum Aktivitätsdiagramm

Aufgabe 6.1
Ausgangssituation:

Für die Speicherung von Messwerten ist die Klasse `CAuswertung` vorgegeben. Die Klasse hat den folgenden Aufbau in C#:

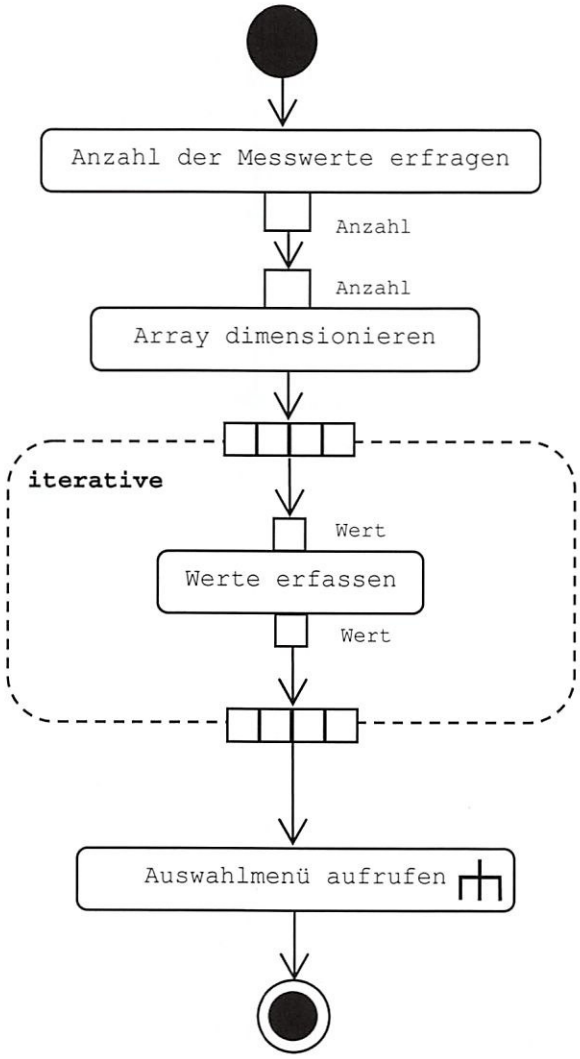
```
class CAuswertung
{
    private double [] werte;
    :
    :
}
```

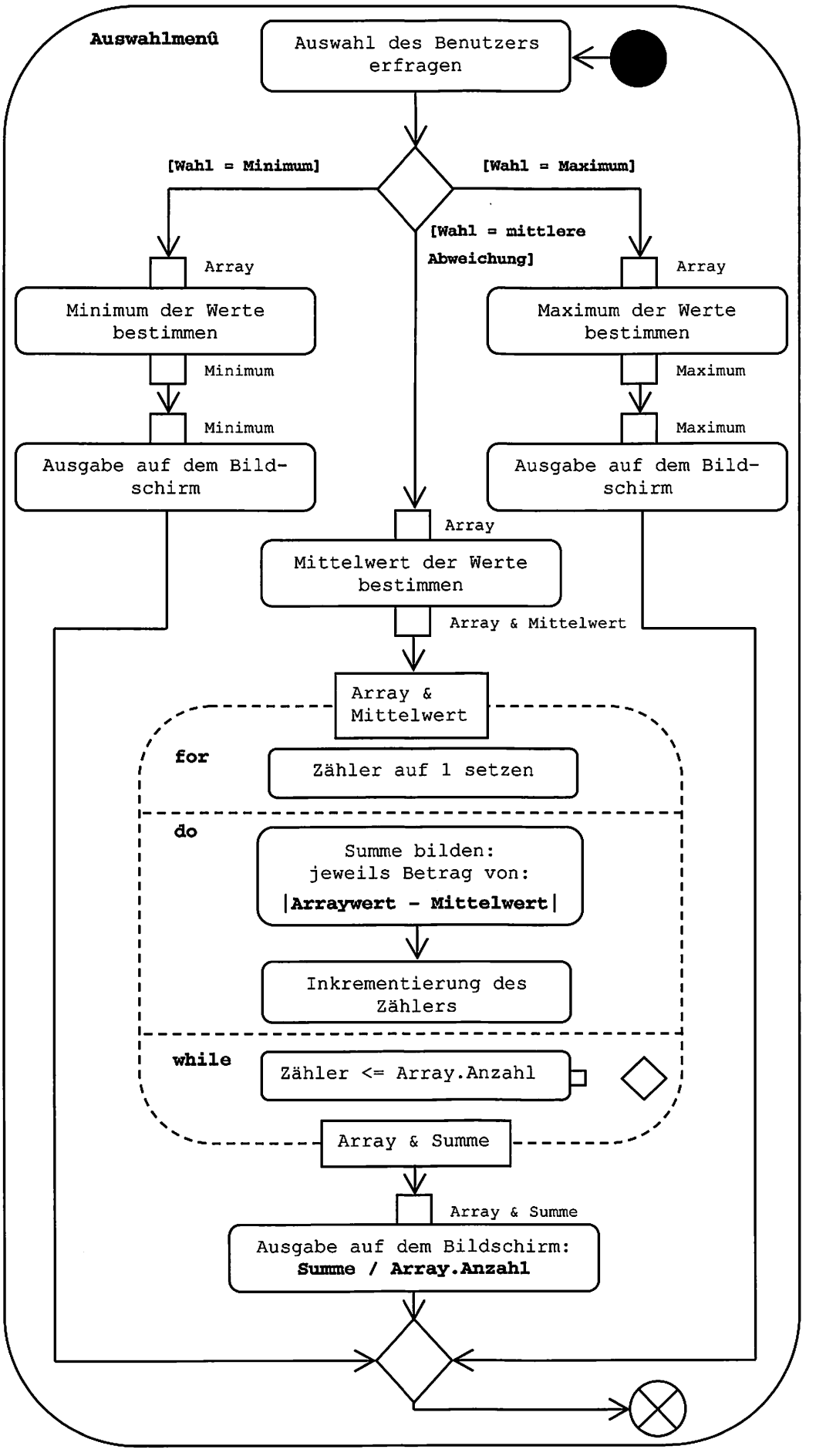
C#-Verweis auf ein `double`-Array

Die Klassen sind in der Lage, beliebig viele Messwerte vom Typ `double` aufzunehmen (dynamische Reservierung), um damit statistische Auswertungen vorzunehmen.

Aufgabenstellung:

Analysieren Sie das folgende Aktivitätsdiagramm und setzen Sie das Diagramm in C# um. Ergänzen Sie die nötigen Methoden und entwerfen Sie eine geeignete Benutzeroberfläche.




Auswahlmenü
Auswahl des Benutzers erfragen
[Wahl = Minimum]
[Wahl = Maximum]
[Wahl = mittlere Abweichung]
Array
Minimum der Werte bestimmen
Minimum
Minimum
Ausgabe auf dem Bild-schirm
Array
Maximum der Werte bestimmen
Maximum
Maximum
Ausgabe auf dem Bild-schirm
Array
Mittelwert der Werte bestimmen
Array & Mittelwert
Array & Mittelwert
for
Zähler auf 1 setzen
do
Summe bilden: jeweils Betrag von: |Arraywert - Mittelwert|
Inkrementierung des Zählers
while
Zähler <= Array.Anzahl
Array & Summe
Array & Summe
Ausgabe auf dem Bildschirm: Summe / Array.Anzahl

Aufgabe 6.2

Entwickeln Sie ein Aktivitätsdiagramm zu der folgenden Ausgangssituation:

Die Kunden einer Kfz-Versicherungsgesellschaft können ihre Schäden online über eine entsprechende Maske eintragen. Die Daten werden anschließend von einem Sachbearbeiter geprüft. In einem ersten Schritt muss die Schuldfrage geklärt werden. Hat der Versicherte Schuld (oder Teilschuld) an dem Schadensfall, so fordert der Sachbearbeiter ein Gutachten an, in dem die Schadenssumme genau berechnet wird. Liegt diese Schadenssumme unter 12.500 Euro, so muss der Sachbearbeiter die Vertragsdaten des Versicherten anpassen (Anpassung des Schadensfreiheitrabatts). Danach kann eine Reparaturfreigabe erfolgen, allerdings unter einschränkenden Bedingungen, die dem Versicherten mitgeteilt werden.

Liegt die Schadenssumme über 12.500 Euro, so muss der Sachbearbeiter neben der Anpassung des Vertrags auch den Abteilungsleiter informieren. Erst danach kann dann die Reparaturfreigabe unter Einschränkung erfolgen.

Falls der Versicherte keine (Teil-)Schuld trägt, so kann eine uneingeschränkte Reparaturfreigabe erfolgen.

7 Aufgaben zur Softwareentwicklung

Hinweis:

Die Aufgaben zur Softwareentwicklung sind in der Regel sehr komplex, da sie mehrere Diagrammtypen und die entsprechenden Implementierungen enthalten. Deshalb werden an dieser Stelle keine einzelnen Aufgaben gestellt, sondern auf die Lernsituationen verwiesen. Die Lernsituationen bieten den Rahmen für solche komplexen Aufgaben.

8 Aufgaben zu den weiteren Diagrammen

Aufgabe 8.1 Kompositionsstrukturdiagramm

Stellen Sie die folgende Problematik mithilfe eines Kompositionsstrukturdiagramms dar:

Eine Schule hat Lehrer, die wiederum Klassen (maximal sieben) unterrichten. Die Lehrer haben einen Zugang zu der Vertretungsplansoftware, über welche sie die aktuellen Vertretungspläne abrufen können.

Weiterhin kommunizieren Lehrer und Schüler, wenn es um die Beurteilung von Leistungen geht. Dabei kann die Kommunikation zwischen Sportlehrern und deren Schülern als ein Spezialfall betrachtet werden, denn die Mitarbeitsnote wird anders ermittelt als im Unterricht, der in einem Klassenraum stattfindet.

Aufgabe 8.2 Komponentendiagramm

Ausgangssituation:

Ein Online-Auktionshaus möchte seine Software komplett neu gestalten. In einem ersten Schritt soll eine Übersicht der zugrunde liegenden Komponenten entwickelt werden. Das neue System soll ein Drei-Schichten-System sein (*three tier*). Auf einem Server soll ein Webserver installiert werden, der über einen Perl-Interpreter verfügt. In der Sprache Perl sollen dann auch die nötigen Klassen für die Abwicklung der Auktionen geschrieben werden (Kunden-Klasse, Auktions-Klasse etc.). Die Daten sollen in einer Datenbank gesichert werden, die auch auf einem anderen separaten Server installiert werden kann. Der Zugriff der Kunden erfolgt über einen beliebigen Browser, der allerdings für JavaScript geeignet sein muss, da einige JavaScripts lokal auf dem Browser laufen müssen.

Entwickeln Sie ein Komponentendiagramm zu der obigen Ausgangssituation.

Aufgabe 8.3 Verteilungsdiagramm

Ergänzen Sie das Komponentendiagramm zu der geplanten Online-Auktions-Software (aus der Aufgabe 8.2) durch ein Verteilungsdiagramm. Dabei werden die folgenden Daten zur benötigten Hardware und Software zugrunde gelegt:

Anforderungen an den Server:

Beschreibung	Anzahl
Prozessor: Intel® Xeon® Silver 4110	1
RAM: 64 GB ECC DDR4	1
RAID: RAID 6	1
Festplatte: 960 GB SSD	4
USV-System	1
Betriebssystem: Linux	1
Webserver: Apache	1

Anforderungen an den Client:

Beschreibung	Anzahl
Prozessor: ab Intel Core i5	1
RAM: ab 4 GB DDR4	1
Festplatte: ab 500 GB	1
Grafikkarte: ab Auflösung 2560x1600	1
Betriebssystem: Linux / Windows XX	1
Browser: Microsoft Edge ab 44.XX / Firefox ab 62.XX	1

Anforderungen an den Datenbankserver:

Beschreibung	Anzahl
Prozessor: ab Intel Core i7	1
RAM: ab 16 GB	1
Festplatte: ab 1,5 TB	1
Betriebssystem: Linux / Windows XX	1
DBMD: Oracle / MySQL	1

Anforderungen an die zu installierende Software:

Beschreibung	Anzahl
Perl-Interpreter ab Version 5.2	1
Oracle DBMS ab 11g/ MySQL ab 8	1
JavaScript: ab ECMA-Script 2017	1
Apache: ab 2.4	1

Aufgabe 8.4 Paketdiagramm

Ausgangssituation:

Die Firma *ProMath* entwickelt mathematische Computersoftware in einer objektorientierten Sprache. Die einzelnen Funktionalitäten der Software sind in Module (Pakete) unterteilt. In einem Grundlagenmodul sind Klassen für die Bruchrechnung und die Dezimalrechnung implementiert mit Funktionalitäten wie „*Kürzen*" und „*Wurzel ziehen*". Die Module zur Differenzialrechnung und Integralrechnung beinhalten dieses Grundlagenmodul. Zusätzlich haben die beiden Module auch Zugriff auf das Modul der komplexen Zahlen, welches wiederum Zugriff auf das Trigonometrie-Modul hat, um Funktionen wie *Kosinus* und *Sinus* zu benutzen, die in einer entsprechenden Klasse angeboten werden. Allerdings soll der Zugriff auf die trigonometrischen Funktionen nur dem Modul der komplexen Zahlen möglich sein.

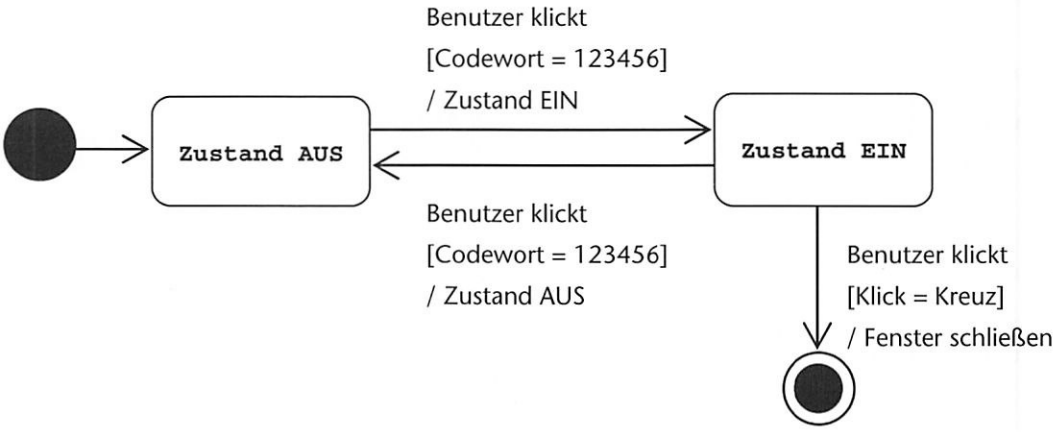
Entwickeln Sie ein Paketdiagramm zu der obigen Ausgangssituation.

Aufgabe 8.5 Profildiagramm

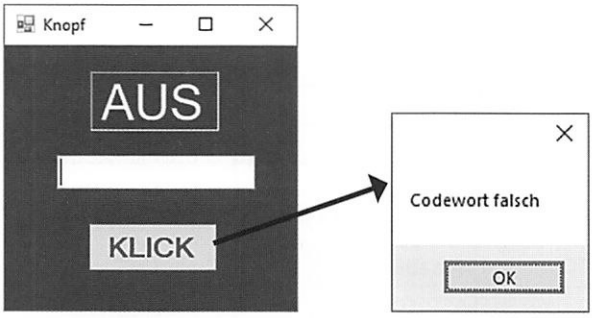
Dieses Diagramm arbeitet auf einer Meta-Ebene und wird von erfahrenen Entwicklern eingesetzt, um die Sprache UML an ihre Bedürfnisse anzupassen. Eine Aufgabenstellung zu diesem Diagramm würde deshalb an dieser Stelle nicht angemessen sein. Dem fortgeschrittenen Leser sei beispielsweise die Literatur zur modellgetriebenen Softwareentwicklung (Model-Driven Architecture, *MDA*) empfohlen.

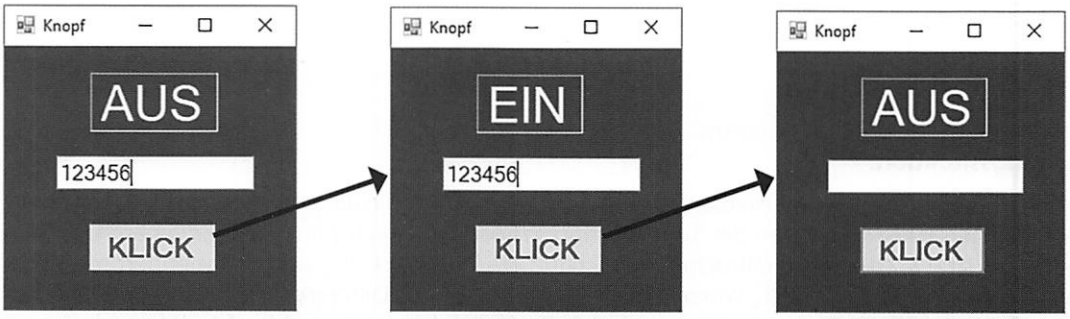
Aufgabe 8.6 Zustandsdiagramm

Für die Steuerung einer Maschine soll eine Art Knopf simuliert werden, auf den ein Benutzer klicken kann. Zusätzlich muss eine Art Codewort eingegeben werden. Ist das Codewort korrekt, so ändert der Knopf seinen Zustand. Das folgende Zustandsdiagramm soll diesen Vorgang beschreiben:

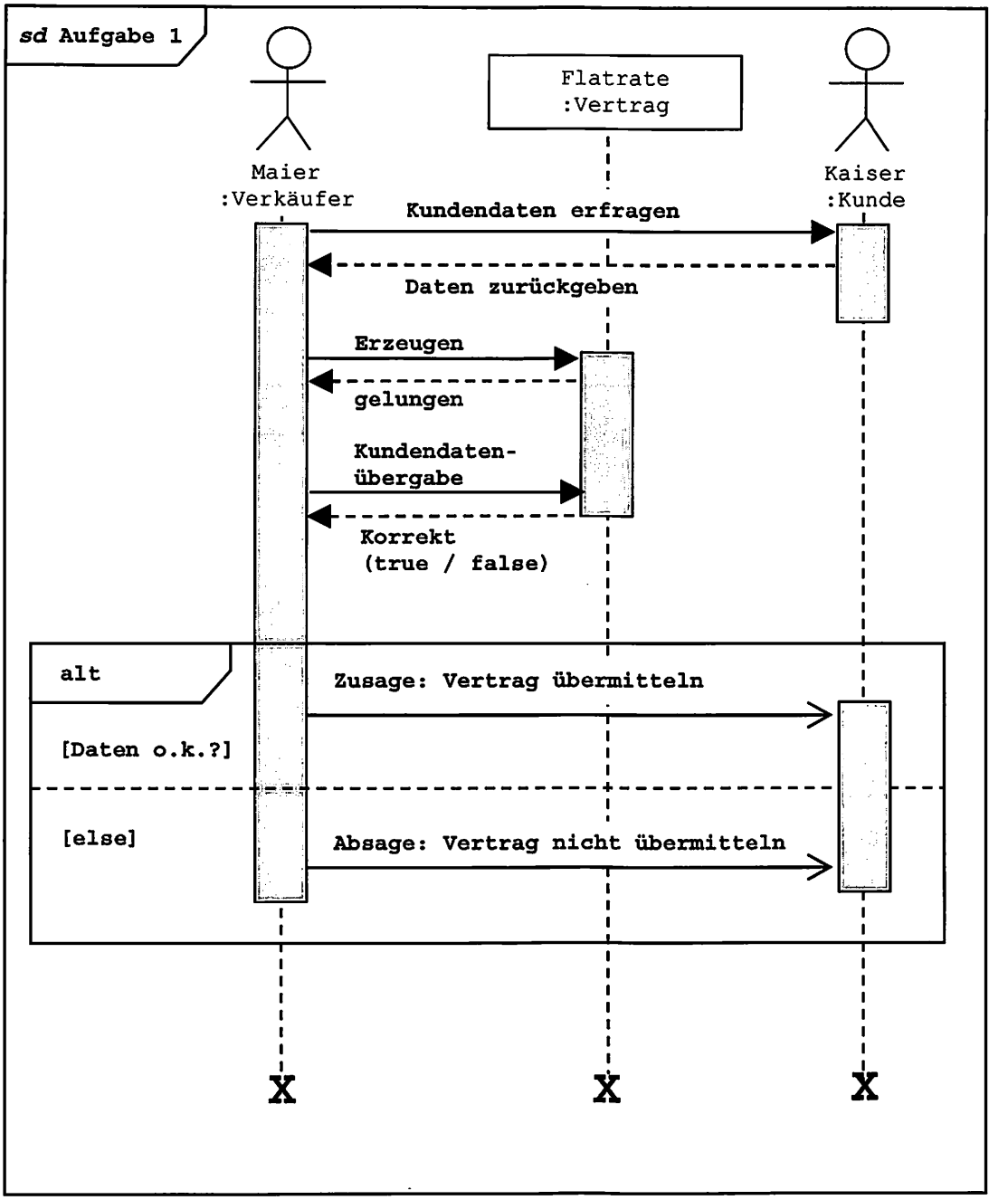


Entwickeln Sie eine Windows-Anwendung (beispielsweise eine Windows-Forms-Anwendung), die dieses Zustandsdiagramm in Form eines kleinen Fensters modelliert. Die folgenden Screenshots zeigen, wie das Fenster aussehen könnte:



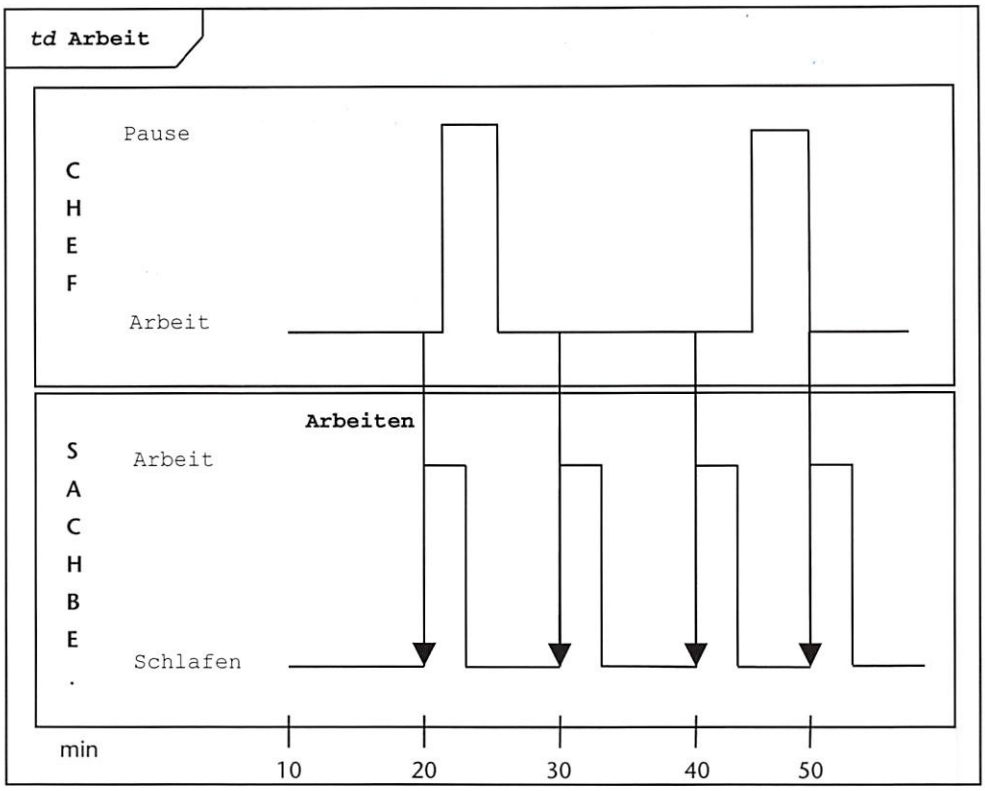


Aufgabe 8.7 Kommunikationsdiagramm
Übersetzen Sie das bereits bekannte Sequenzdiagramm aus Aufgabe 5.1 in ein Kommunikationsdiagramm.



Aufgabe 8.8 Zeitverlaufsdiagramm

Das folgende Zeitverlaufsdiagramm soll den Arbeitsprozess eines Sachbearbeiters und seines Chefs widerspiegeln. Der Sachbearbeiter schläft hauptsächlich und wird allerdings in festgelegten Intervallen von seinem Chef zur Arbeit ermahnt. Der Sachbearbeiter arbeitet dann für einen kurzen Moment und fällt anschließend wieder in den Schlaf. Der Chef hingegen arbeitet hauptsächlich und macht in regelmäßigen Abständen eine Pause.



Hinweise:

Der Chef sendet alle 10 Minuten ein Signal zum Sachbearbeiter. Der Sachbearbeiter arbeitet dann für 3 Minuten und fällt anschließend wieder in den Schlaf. Alle 20 Minuten macht der Chef eine 5-minütige Pause.

Setzen Sie das Zeitverlaufsdiagramm in einer Windows-Anwendung (beispielsweise Windows-Forms) um. Benutzen Sie dazu einige Timer, die die Zustände steuern können.

Beispiel: Einsatz eines Timers in C#

```
private Timer timer = new Timer();
timer.Interval = 3000;  // in Millisekunden

timer.Tick += new EventHandler(TimerMethode);

timer.Start(); //Der Timer wird gestartet

void TimerMethode(object obj, EventArgs ea)
{

   //TODO: die Funktionalität programmieren
}
```

Die Methode wird immer nach 3000 Millisekunden aufgerufen.

Der Delegate `Tick`

Die folgenden Screenshots zeigen eine mögliche Umsetzung in C# (die Minuten werden durch Sekunden ersetzt): Das Programm startet um **09:11:38** Uhr.





- Es ist **09:11:38** Uhr – der Sachbearbeiter schläft und der Chef arbeitet.
- Um **09:12:14** Uhr macht der Chef Pause und der Sachbearbeiter arbeitet.





- Um **09:12:33** Uhr arbeiten Chef und Sachbearbeiter.
- Um **09:13:05** Uhr macht der Chef wieder Pause und der Sachbearbeiter schläft auch schon wieder.

Hinweis:

Die Bilder sind mithilfe einer `PictureBox` eingebunden worden. Das Ändern eines Bildes geschieht einfach durch die Zuweisung eines anderen Bildes an die Eigenschaft `Image`.

Aufgabe 8.9 Interaktionsübersichtsdiagramm

Zu diesem Diagrammtyp wird keine separate Aufgabe gestellt, da es im Prinzip nur um die Integration aller besprochenen Diagramme in ein Übersichtsdiagramm geht.

Teil 3
Lernsituationen

Lernsituation 1:
Erstellen einer Präsentation mit Hintergrundinformationen zur Sprache UML (in Deutsch oder Englisch) 146

Lernsituation 2:
Anfertigen einer Dokumentation für den Einsatz eines CASE-Tools (in Deutsch oder Englisch) 147

Lernsituation 3:
Entwicklung einer Software zur Darstellung von Wetterdaten mit dem Model-View-Controller-Konzept 148

Lernsituation 4:
Durchführung einer objektorientierten Analyse und eines objektorientierten Designs zur Entwicklung eines Softwaresystems zur Verwaltung der Schulbibliothek eines Berufskollegs 151

Lernsituation 5:
Entwicklung einer Software zur Verwaltung eines Schulungsunternehmens 152

Lernsituation 1:

Erstellen einer Präsentation mit Hintergrundinformationen zur Sprache UML (in Deutsch oder Englisch)

Ausgangssituation:

Sie haben die Ausbildung zum Fachinformatiker bei der mittelständischen Softwareentwicklungsfirma **ProSource** begonnen. Unter anderem führt die Firma Inhouse-Schulungen in verschiedenen IT-Bereichen durch.

Es ist eine Schulung in der formalen Sprache UML geplant. Da die Entwickler der Firma unter Zeitdruck stehen, ist die Vorbereitung der Schulung problematisch. Sie erhalten deshalb den Auftrag, den einführenden Informationsteil der Schulung zu gestalten. Dieser Teil soll ungefähr 15 Minuten in Anspruch nehmen. Neben historischen Daten sollen die interessanten Aspekte der Sprache ansprechend vorgestellt werden. Die Präsentation soll auch in den ausländischen Niederlassungen der Firma genutzt und deshalb auch parallel in englischer Sprache vorbereitet werden.

Arbeitsschritte in Einzel- oder Partnerarbeit:

Planung:

Legen Sie die Präsentationsmittel fest (PowerPoint-Präsentation, Handouts usw.).

Denken Sie über den Umfang der Präsentation nach (Zeitrahmen: 15 Minuten).

Informieren Sie sich über die Hintergründe von UML mithilfe des Informationsteils dieses Buches und weiteren Quellen wie dem Internet.

Wenn es möglich ist, dann arbeiten Sie fächerübergreifend im Deutsch- und/oder Englischunterricht an der Präsentation weiter.

Durchführung:

Gestalten Sie die Folien ansprechend, ohne sie zu überfrachten. Formulieren Sie die Folientexte kurz und aussagekräftig. Halten Sie die Präsentation entweder in Deutsch oder in Englisch, wenn der fächerübergreifende Unterricht stattfinden konnte.

Kontrolle:

Führen Sie die Präsentation vor Ihrem Partner oder einer anderen Lerngruppe vor. Der Partner bzw. die Zuhörer beobachten die Präsentation unter Einbeziehung des unten angegebenen Kriterienkatalogs, der im Anschluss die Grundlage für das kritische Auseinandersetzen bietet.

Kriterienkatalog für die Beurteilung einer Präsentation

- ✓ **Fachliche Seite:**
- ✓ Gliederung/Strukturierung des Vortrages
- ✓ Logischer Aufbau des Vortrages
- ✓ Angemessener Einsatz von Fachsprache
- ✓ Zusammenhänge deutlich machen
- ✓ Zusammenfassungen bieten (Resümee ziehen)
- ✓ Abschnitte in dem Vortrag deutlich machen
- ✓ **Persönliche Seite:**
- ✓ Ruhige und präzise Sprache/Aussprache
- ✓ Pausen machen
- ✓ Dynamik und Gestik
- ✓ Blickkontakt zu den Zuhörern

Lernziele:

- Sie lernen wichtige Aspekte der formalen Sprache UML kennen.
- Sie sammeln Erfahrungen im Erarbeiten und Durchführen einer fachlichen Präsentation im Bereich IT.
- Sie verbessern Ihre Kenntnisse in der englischen Sprache und lernen grundlegende Fachbegriffe in Englisch kennen.
- Sie verbessern das gezielte Beobachten von anderen Vorträgen.

Lernsituation 2:

Anfertigen einer Dokumentation für den Einsatz eines CASE-Tools (in Deutsch oder Englisch)

Ausgangssituation:

Die Softwareentwicklungsfirma **ProSource** hat sich entschieden, in künftigen Projekten mit einem CASE-Tool (**C**omputer **A**ided **S**oftware **E**ngineering) zu arbeiten. Viele Entwickler der Firma haben keine Erfahrungen mit solchen Tools. Aus Gründen der Motivation hat sich die Firmenleitung entschlossen, diesen Entwicklern den Einstieg zu erleichtern.

Deshalb soll eine kurze und verständliche Dokumentation zu dem CASE-Tool geschrieben werden. Als erfahrener Auszubildender der Firma erhalten Sie den Auftrag, diese Dokumentation zu schreiben. Der Umfang sollte sechs Seiten nicht überschreiten. Die Dokumentation könnte auch in den ausländischen Niederlassungen der Firma genutzt werden und sollte deshalb auch parallel in englischer Sprache vorbereitet werden.

Arbeitsschritte in Einzel- oder Partnerarbeit:

Planung:
Überlegen Sie sich ein Konzept für diese Dokumentation. Orientieren Sie sich an den folgenden Fragen:

- Wie kann die Dokumentation gegliedert werden (allgemeine Beschreibung, Anlegen eines Projektes in dem CASE-Tool)?
- Mit welchem Programm kann die Dokumentation adäquat umgesetzt werden (Editor, Word o.Ä.)?

Entscheiden Sie sich für ein CASE-Tool. Vorgeschlagen wird hier die kostenfreie Evaluationsversion von StarUML. Wenn Sie in Ihrem Lernumfeld mit einem anderen Tool arbeiten, so können Sie auch dieses verwenden.

Wenn es möglich ist, dann arbeiten Sie fächerübergreifend im Deutsch- und/oder Englischunterricht an der Dokumentation weiter.

Benutzen Sie den Informationsteil dieses Buches und weitere Quellen wie das Internet, um die nötigen Kenntnisse zu erarbeiten.

Durchführung:
Gestalten Sie die Kundendokumentation in einem ansprechenden Wechsel von Text und Grafik (Screenshots des CASE-Tools).

Kontrolle:
Nutzen Sie die Rechtschreibkontrolle Ihrer Textverarbeitung. Falls es möglich ist, lassen Sie Ihre Dokumentation von jemandem lesen, der das CASE-Tool nicht kennt – es wird sich dann zeigen, ob Ihre Erläuterungen zum Ziel führen.

Lernziele:

- Sie lernen ein CASE-Tool kennen und können Projekte damit anlegen.
- Sie erarbeiten die Grundstruktur des Tools.
- Sie verbessern Ihre Kenntnisse in der englischen Sprache und lernen grundlegende Fachbegriffe in Englisch kennen.

Lernsituation 3:

Entwicklung einer Software zur Darstellung von Wetterdaten mit dem **Model-View-Controller-Konzept**

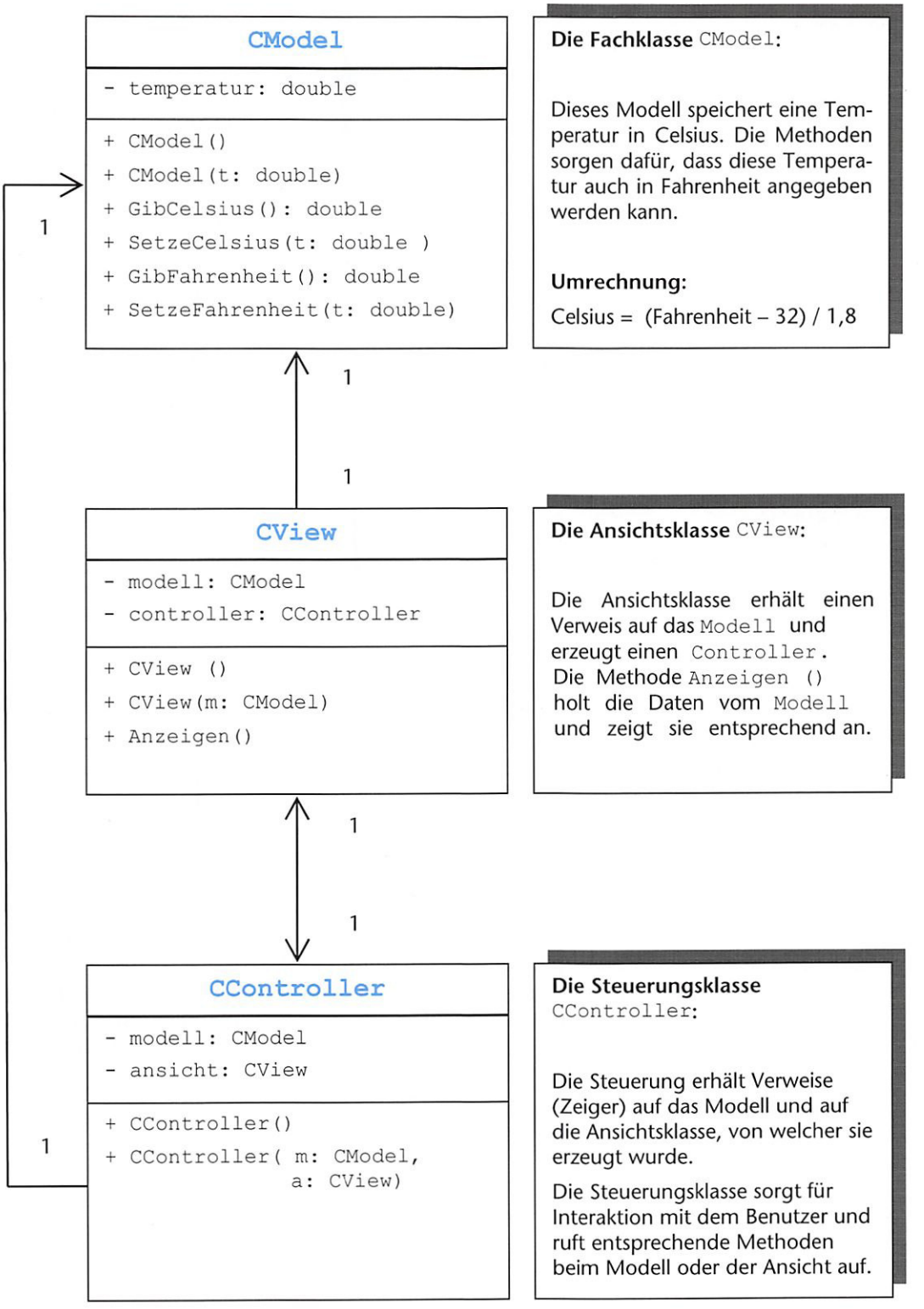
Ausgangssituation:

Die Firma ***WetterCom*** liefert Messdaten von ihren Messstationen an ihre Kunden. Bislang wurden die Daten als Dateianhänge im Textformat versendet. Diese nicht zeitgemäße Versendung von Daten soll durch ein neues Softwaresystem abgelöst werden. Dieses System soll mit der **MVC**-Architektur realisiert werden (Model-View-Controller-Architektur).

In einem ersten Prototyp soll eine lokale Anwendung erstellt werden, die ein einfaches MVC-Modell umsetzt. Die Firma **ProSource** erhält den Auftrag, diese Prototyp-Anwendung in der Sprache C# als Konsolenanwendung zu entwickeln. Als Auszubildender der Firma **ProSource** haben Sie bereits einige Konsolenanwendungen entwickelt. Die Aufgabe wird deshalb an Sie delegiert.

Arbeitsschritte in Einzel- oder Partnerarbeit:

Planung:
Aus der objektorientierten Analyse liegt das folgende Klassendiagramm vor. Es beschreibt den Aufbau der Klassen und deren Beziehung untereinander.



Die Fachklasse `CModel`:

Dieses Modell speichert eine Temperatur in Celsius. Die Methoden sorgen dafür, dass diese Temperatur auch in Fahrenheit angegeben werden kann.

Umrechnung:

Celsius = (Fahrenheit – 32) / 1,8

Die Ansichtsklasse `CView`:

Die Ansichtsklasse erhält einen Verweis auf das `Modell` und erzeugt einen `Controller`. Die Methode `Anzeigen ()` holt die Daten vom `Modell` und zeigt sie entsprechend an.

Die Steuerungsklasse `CController`:

Die Steuerung erhält Verweise (Zeiger) auf das Modell und auf die Ansichtsklasse, von welcher sie erzeugt wurde.

Die Steuerungsklasse sorgt für Interaktion mit dem Benutzer und ruft entsprechende Methoden beim Modell oder der Ansicht auf.

Hinweis:

Das obige Modell ist eine einfache Variante der MVC-Architektur, da nur ein Modell, eine Ansicht und eine Steuerung instanziert werden sollen. In der Regel gibt es mehrere Ansichten oder auch Steuerungen, die auf das Modell zugreifen.

Eine Bildschirmausgabe der Konsolenanwendung könnte so aussehen:

```
Auswahl:
<1> Neue Temperatur in Celsius eingeben
<2> Neue Temperatur in Fahrenheit eingeben
<3> ENDE
1
Bitte die Temperatur in Celsius: 10
```

Controller

```
T E M P E R A T U R - U M R E C H N U N G <Version 1.0>
Temperatur in Celsius:    10
Temperatur in Fahrenheit: 50
```

View

```
Auswahl:
<1> Neue Temperatur in Celsius eingeben
<2> Neue Temperatur in Fahrenheit eingeben
<3> ENDE
1
Bitte die Temperatur in Celsius: 20
```

Controller

```
T E M P E R A T U R - U M R E C H N U N G <Version 1.0>
Temperatur in Celsius:    20
Temperatur in Fahrenheit: 68
```

View

Durchführung:

Implementieren Sie die drei Klassen entsprechend des Klassendiagramms. Die Eingabelogik (Menü und Benutzereingabe) soll dabei als Methode der Steuerungs-Klasse umgesetzt werden.

Die Ausgabe auf dem Bildschirm wird als Methode der Ansichts-Klasse umgesetzt, die von der Steuerungs-Klasse aufgerufen wird, sobald der Benutzer eine neue Wahl getroffen hat. Die Modell-Klasse wird sowohl von der Steuerungs-Klasse (zur Speicherung neuer Werte) als auch von der Ansichts-Klasse genutzt, um die Daten für die Bildschirmausgabe zu erhalten.

Das Hauptprogramm in C# ist vorgegeben:

```
static void Main(string[] args)
{
   CModel modell = new CModel(10);
   CView ansicht = new CView(modell);
}
```

Kontrolle:
Kontrollieren Sie die korrekte Programmausführung durch die Eingabe von einigen Werten, die umgerechnet und dargestellt werden sollen.

Lernziele:

- Sie lernen die Grundzüge einer wichtigen Architektur kennen – der **Model-View-Controller-Architektur**.
- Sie erarbeiten sich weitere Kenntnisse in der Umsetzung von UML-Diagrammen.

Lernsituation 4:

Durchführung einer objektorientierten Analyse und eines objektorientierten Designs zur Entwicklung eines Softwaresystems zur Verwaltung der Schulbibliothek eines Berufskollegs

Ausgangssituation:

Als Auszubildender der Firma **ProSource** haben Sie zusätzlich zu Ihrer Ausbildung in der Firma Unterricht in der Berufsschule. Diese Berufsschule ist ein Berufskolleg mit vielen verschiedenen Bildungsgängen.

Im Rahmen des Unterrichts für das Lernfeld „Entwickeln und Bereitstellen von Anwendungssystemen" erhalten Sie den Auftrag, ein Softwaresystem für die Verwaltung der Schulbibliothek zu entwickeln.

Die Schulleitung ist sich nicht einig, ob das System mit Open-Source-Software oder einer lizenzierten Entwicklungsumgebung umgesetzt werden soll. Deshalb beschränkt sich der Auftrag auf die Definition der Anforderungen, die objektorientierte Analyse sowie das objektorientierte Design.

Arbeitsschritte in Einzel- oder Partnerarbeit:

Planung:
In einem Gespräch mit dem zuständigen Lehrer für die Schulbibliothek wurden die folgenden Kriterien für das System festgehalten:

- Die Anzahl der verschiedenen Bücher liegt momentan bei circa 175. Es können aber durchaus noch mehr Bücher werden.
- Von jedem Buch sind mindestens 20 Exemplare (Klassensatz) vorhanden.
- Jeder Schüler darf bis zu drei Bücher ausleihen. Die Ausleihfrist beträgt 21 Tage.
- Das System sollte alle Bücher bzw. Exemplare erfassen. Ebenso werden alle Schüler erfasst, die ein Buch ausleihen.
- Wenn ein Schüler die Leihfrist überschreitet, so soll das System eine automatische Erinnerungs-E-Mail an die hinterlegte Adresse des Schülers senden.
- Bei Überschreitung einer Leihfrist um vier Wochen wird ein automatisches Mahnschreiben generiert. Wenn der Schüler bereits mehr als dreimal die Leihfrist um vier Wochen überschritten hat, wird ihm das Nutzungsrecht für sechs Monate entzogen.
- Gibt der Schüler trotz Mahnungen die Bücher nicht zurück, so wird ihm eine Rechnung über die Bücher gestellt, die entsprechend dem Wiederbeschaffungswert festzusetzen ist.

Durchführung:

- Definieren Sie die Anforderungen an das System mithilfe eines Anwendungsfalldiagramms.
- Entwickeln Sie in der objektorientierten Analyse ein Klassendiagramm und ein Sequenzdiagramm und verfeinern Sie die Diagramme in der Designphase.
- Stellen Sie die Problematik der Leihfristüberschreitung zusätzlich mit einem Aktivitätsdiagramm dar.

Kontrolle:

Tauschen Sie die Ergebnisse Ihrer Arbeit mit den Mitschülerinnen und Mitschülern Ihrer Klasse aus. Kontrollieren Sie die Ergebnisse und diskutieren Sie über Abweichungen in der Analyse und dem Design.

Lernziele:

- Sie erarbeiten sich die wichtigen Schritte einer objektorientierten Softwareentwicklung.
- Sie lernen die Diagrammtypen Anwendungsfalldiagramm, Klassendiagramm, Sequenzdiagramm und Aktivitätsdiagramm in den Phasen der objektorientierten Analyse und des objektorientierten Designs einzusetzen.

Lernsituation 5:

Entwicklung einer Software zur Verwaltung eines Schulungsunternehmens

Ausgangssituation:

Das Schulungsunternehmen **CompuDacta** ist spezialisiert auf Schulungen im IT-Bereich. Für die Verwaltung der angebotenen Kurse sowie der Dozenten soll eine neue Software eingesetzt werden. Das Unternehmen konnte im Bereich der Standardsoftware keine angemessene Lösung finden und möchte die Software jetzt neu entwickeln lassen.

Die Firma **ProSource** erhält dazu den Auftrag, eine Software zur Verwaltung der Kurse, Dozenten und Teilnehmer zu entwickeln. Als erfahrener Auszubildender der Firma **ProSource** haben Sie bereits Erfahrungen mit der objektorientierten Softwareentwicklung gemacht. Die Aufgabe wird deshalb an Sie delegiert.

Arbeitsschritte in Einzel- oder Partnerarbeit:

Planung:

Das Schulungsunternehmen hat ein Lastenheft eingereicht, in dem die Anforderungen an die Software beschrieben wurden. Die folgenden Ausführungen zeigen die wichtigsten Anforderungen der Software.

LASTENHEFT

CompuDacta

IT-Schulungsunternehmen

Beschreibung:

Die Firma ProSource soll damit beauftragt werden, eine Individualsoftware zu entwickeln, mit der die Schulungskurse, Mitarbeiter, Dozenten und Teilnehmer verwaltet werden können. Die Software soll benutzerfreundlich sein und über automatische Datensicherungsmechanismen verfügen (Speichern der Daten in einer Datei oder in einer Datenbank).

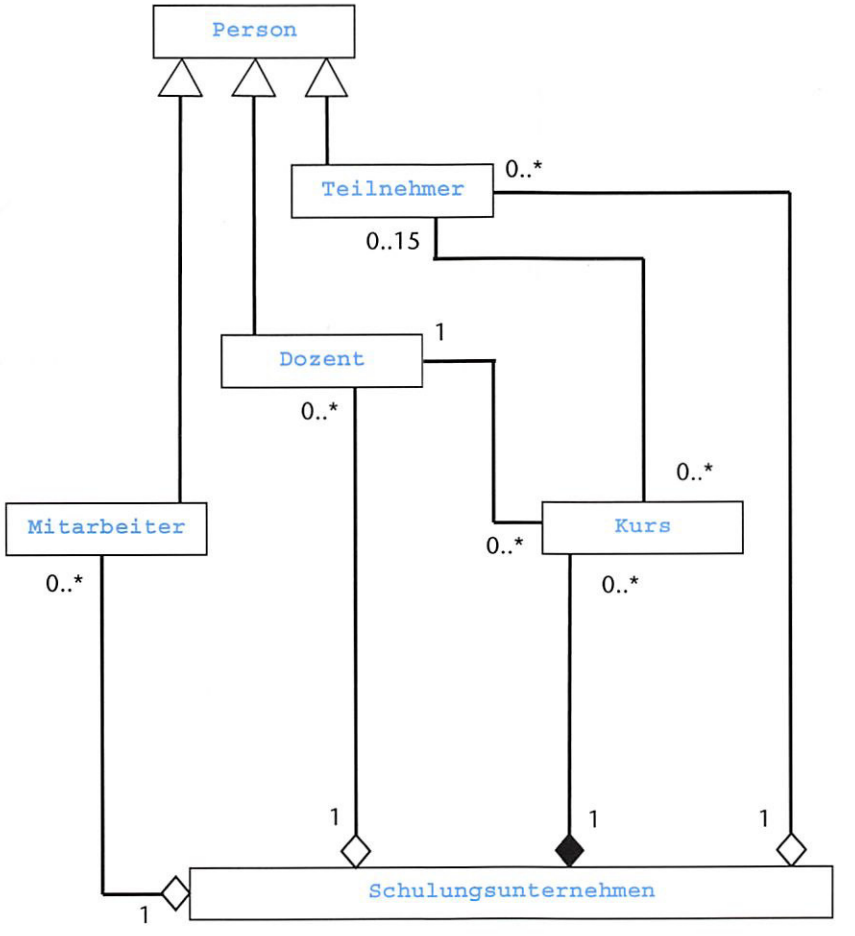
Wichtige Funktionalitäten:

- Anlegen von Mitarbeitern und Verwalten bestehender Mitarbeiter
- Anlegen von neuen Kursen und Verwalten bestehender Kurse
- Anlegen von neuen Dozenten und Verwalten bestehender Dozenten
- Anlegen von neuen Teilnehmern und Verwalten bestehender Teilnehmer
- Teilnehmer und Dozenten den Kursen hinzufügen
- Ausgabe einer Liste der Kurse (inkl. Dozenten und Teilnehmer)
- Ausgabe einer Liste der Dozenten (inkl. der Kurse)
- Ausgabe einer Liste der Mitarbeiter
- Ausgabe einer Liste der Teilnehmer (inkl. der Kurse)

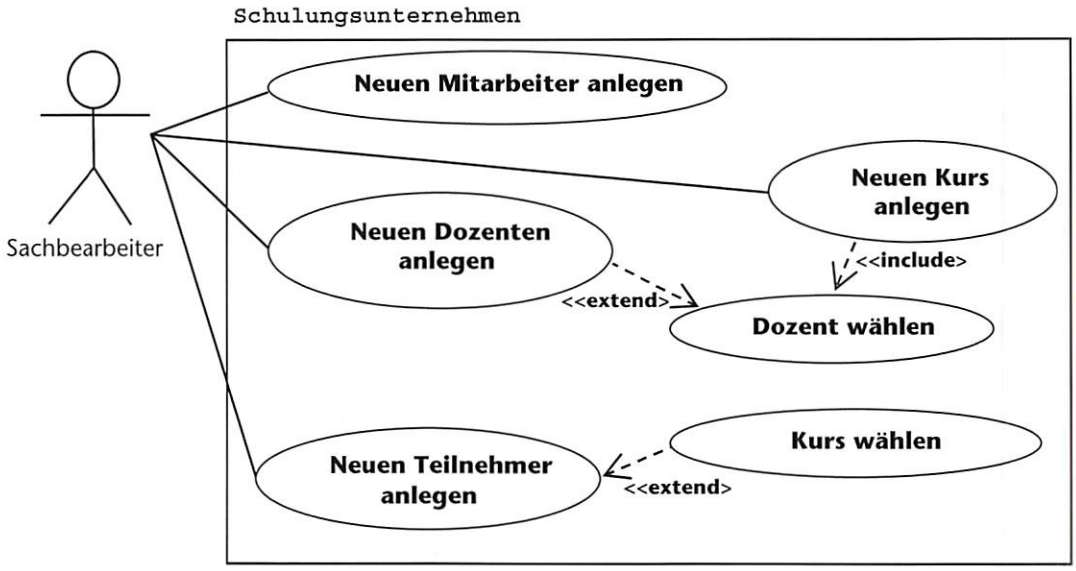
Durchführung:

In der objektorientierten Analyse sind die folgenden Diagramme entstanden:

Klassendiagramm:



Anwendungsfalldiagramm:



Implementieren Sie die Klassen in C# und achten Sie dabei auf die korrekte Umsetzung der Beziehungen und deren mögliche Implikationen, die im Klassendiagramm und Anwendungsfalldiagramm dargestellt sind.

Schreiben Sie eine Konsolenanwendung mit einem Hauptprogramm, welches ein Menü anbietet, das die Funktionalitäten aus dem Lastenheft widerspiegelt. Orientieren Sie sich dabei auch an dem Anwendungsfalldiagramm.

Alternativ zu einer Konsolenanwendung können Sie auch eine Windows-Anwendung implementieren, die die Eingabe und Verwaltung der Kurse, Mitarbeiter, Dozenten und Teilnehmer übernimmt.

Die Sicherung der Daten kann entweder in einer entsprechend aufgebauten Datei oder in einer Datenbank geschehen. Dazu wäre die Entwicklung eines ER-Modells sinnvoll, welches die Klassen in entsprechende Entitäten abbildet.

Kontrolle:

Testen Sie das Programm mit dem ***Black-Box***-Testverfahren. Die Testpersonen kennen den internen Aufbau des Programms nicht und testen nur die Funktionalität und das Ergebnis.

Simulieren Sie die Eingaben von möglichst vielen Mitarbeitern, Dozenten, Teilnehmern und Kursen.

Lernziele:

- Sie lernen eine komplexere Aufgabenstellung zu analysieren und zu realisieren.
- Sie lernen die Implementierung anhand eines Lastenheftes und geeigneter UML-Diagramme.
- Sie testen Ihr Programm mit einem allgemeinen Testverfahren, dem ***Black-Box***-Test.

Index

A

Abbruchfragment 68
abgeleitete Klasse 16, 50
abstrakte Basis-Klasse 52
access 115
Aggregation 41
Akteure 21
Aktion 71
Aktivität 72
Aktivitätsbalken 64
Alternative 66
Anwendungsfall 20
application server 112
apply 116
Architekturmuster 18
ArrayList 47
Artefakte 110
assert 68
Assoziation 34
asynchrone Nachricht 65
atomar 71
Attribute 13, 14, 28
Aufzählungen 54

B

bag 29
Ball- and Socket-Symbol 56
base 51
Basis-Klasse 16, 50
Beziehungsattributen 37
bidirektionalen Navigierbarkeit 36, 60
Black-Box-Test 154
break 68

C

call by reference 31
call by value 31
client workstation 112
critical 68

D

deploy 112
Deployment 112
device 112

E

Effekte 117
einfacher Datentyp 53
embedded device 112
Empfänger 78
endliche Automaten 116
Endpunkt 73
Entity-Relationship-Modell 28
Entwurfsmuster 18
Entwurfsmuster Observer 126
enumeration 54
Ereignisgesteuerte Prozessketten 71
Ereignisse 117
ER-Modell 154
Events 117
Exception-Handling 79
Exclusive-Oder-Verknüpfung 78
executable 111
execution environment 112
Expansionsbereich 83
extend-Beziehung 25
extension points 25

F

foreach 83
Fragmente 66

G

Gabel-Symbol 72
Gabelung 76
Ganzes-Teile-Beziehung 41
Generalisierung 15, 23, 26, 50
gerichtete Assoziation 22
Geschäftslogik 19
Geschäftsprozesse 20
Grady Booch 11
Guards 71, 117, 120

H

Hardwarekomponenten 111
Hat-Beziehung 41, 46

I

if-else-Logik 66
implements 110
import 115
in 31
include-Beziehung 24
inout 31
instantiate 58
Interaktionsdiagramme 118
Interaktionsrahmen 63, 118, 120
Interaktionsübersichtsdiagramm 123
Interfaces 55
Ist-Beziehung 16, 41, 49
Iteration 81
Ivar Jacobson 11

J

James Rumbaugh 11

K

Klasse 13, 28, 58
Klassenattribut 30
Knoten 112
Kollaborationen 107, 108
Kollaborationsausprägung 109
Kommunikationsdiagramm 118
Kommunikationspfad 113
Komponente 110
Komponentendiagramm 109
Komposition 46
Kompositionsstrukturdiagramm 107
Konnektoren 108
Kritischer Bereich 68

L

Lastenheft 152
Lebenslinie 63, 119, 120
Leserichtung 35
Link 59

M

Mehrfachgeneralisierung 50
merge 115
Methoden 28
Mikrocontroller 112
mobile device 112
Model-View-Controller-Konzept 18, 148
Multiplizitäten 23, 29, 35, 42, 47, 59

N

Nachrichten 64, 119, 121
Navigationsverbot 36, 60
Navigierbarkeit 36, 60

O

Ober-Klasse 16, 50
Object Management Group 11
Objekt 13
Objektdiagramm 58
Objekte 58
Objektfluss 76
Objektorientierte Analyse (OOA) 11

Objektorientierte Programmierung (OOP) 11
Objektorientiertes Design (OOD) 11
Observer-Muster 18
OOP 49
Operationen 28
Option 66
ordered 29
Oszilloskop 120
out 31

P

Paket 113
Paketdiagramm 113
Parallele Ausführung 67
Parametersatz 78
Parts 107
passive Zeit 64
PIN-Notation 77
Polymorphie 16, 18
Ports 108
primitiver Datentyp 53
private 29
profile 116
Programmablaufplan 71
protected 29, 50
Pseudocode 15
public 29
public-Vererbung 51

R

Rational Rose 12
Rational Software 12
readonly 29
Rolle 36, 59

S

Schichtenmodell 19
Schleifen 67
Schnappschuss 58
Schnittstellen 55
script 111
Selektion 81
Sequenzdiagramm 63
service 110
Signal-Empfangen 78
Signal-Senden 78
Softwarekomponenten 111
source 111
specification 110
Spezialisierung 15, 23, 26, 50
Startpunkt 73
Stereotyp 53, 110
Steuerungsfluss 71
Steuerungsfluss-Endpunkt 73
Strukturdiagramme 13, 107
subsystem 110
synchrone Nachricht 65
Systemgrenzen 20

T

Threads 76
three tier 19
Transition 117
two tier 19

U

Übergabeparameter 65
UML-Kommentar 21
ungerichtete Assoziation 22
unidirektionalen Navigierbarkeit 36, 60
Unified Method 11
Unified Modeling Language 11
Unterbrechungsbereich 79
Unterbrechungsfluss 79
Unter-Klasse 16, 50
Use-Case-Diagramme 20

V

Verantwortungsbereiche 75
Vereinigung 76
Vererbung 16
Verhaltensdiagramme 13, 116
Verlaufslinie 121
Verteilungsdiagramm 111
Verzweigungen 71

Z

Zeitverlaufsdiagramm 120
Zusicherung 68
Zustand 116
Zustände von Objekten 77
Zustandsdiagramm 116