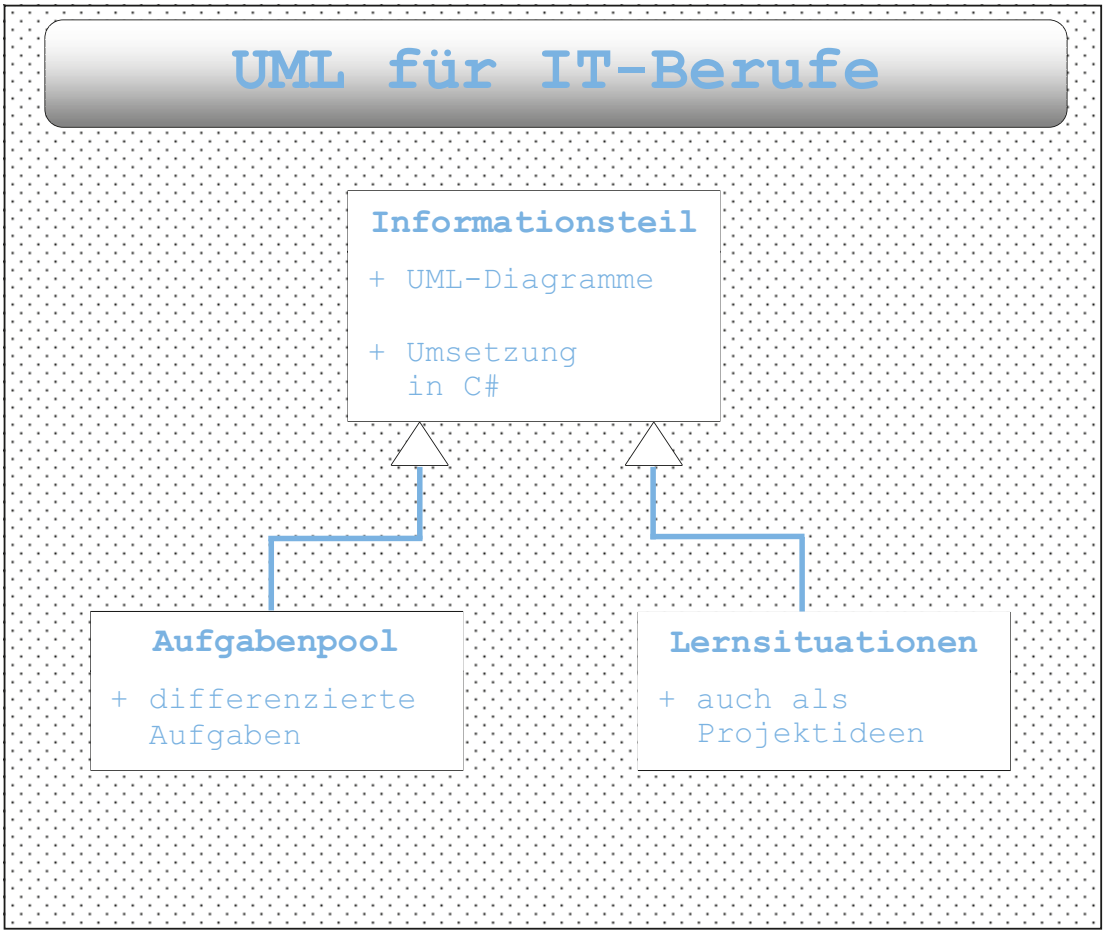
EUROPA-FACHBUCHREIHE
Programmierung für die
IT-Ausbildung



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

Alle Drucke derselben Auflage sind parallel einsetzbar, da sie bis auf die Korrektur von Druckfehlern identisch sind.

ISBN 978-3-8085-8588-7

Satz: Reemers Publishing Services GmbH, Krefeld
Umschlag: braunwerbeagentur, 42477 Radevormwald
Umschlagfotos: envfx-fotolia.com; Gina Sanders-fotolia.com; Gordon Bussiek-fotolia.com
Druck: Plump Druck & Medien 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.

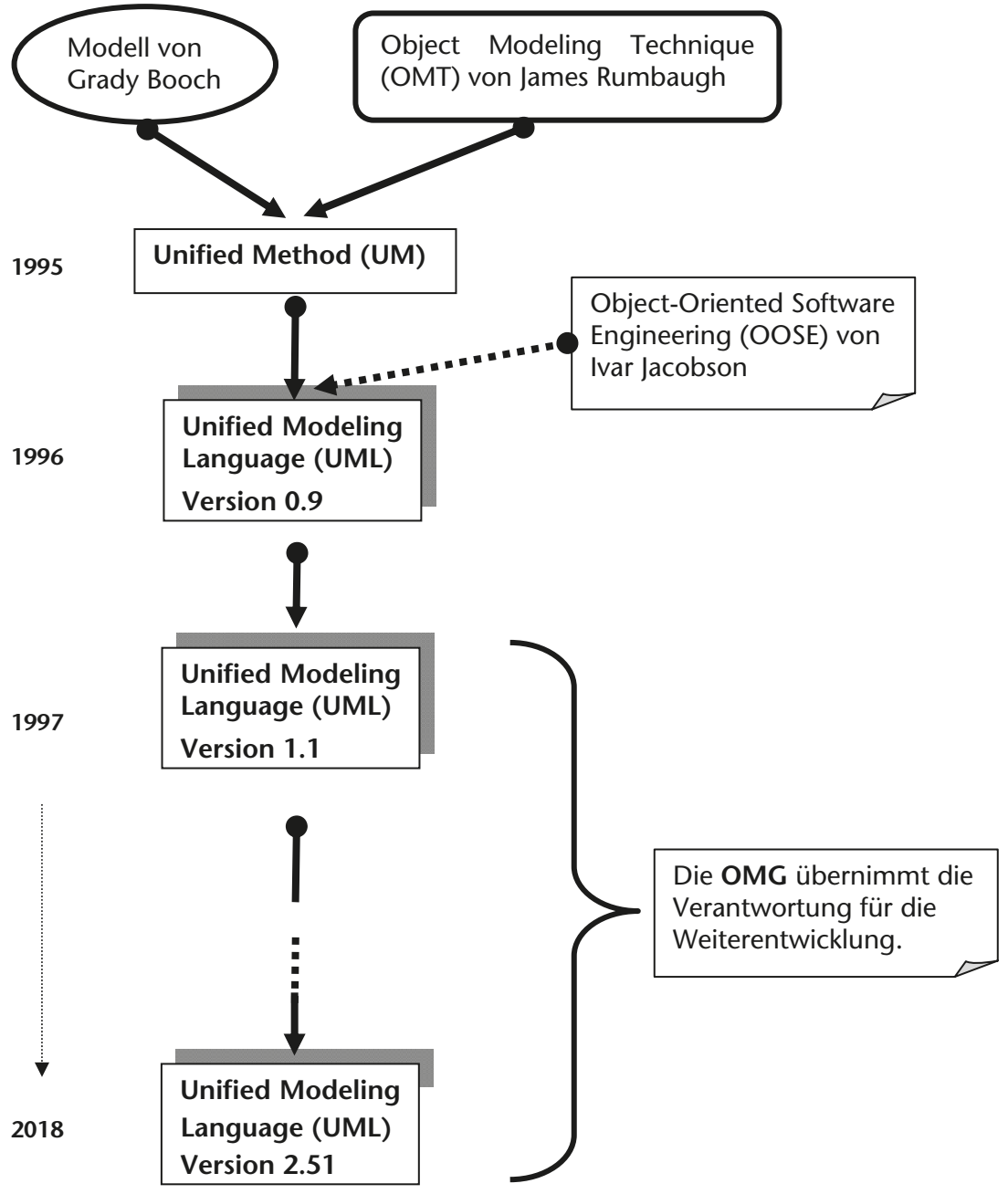
Die UML stellt dazu verschiedene Diagramme zur Verfügung, die wiederum verschiedene grafische Elemente enthalten. Innerhalb der UML gibt es allerdings für ein und denselben Sachverhalt manchmal mehrere Darstellungsarten.

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.

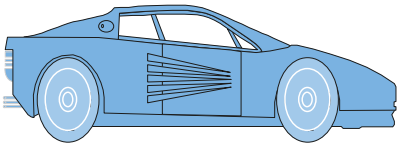
Was ist eine Klasse?

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:

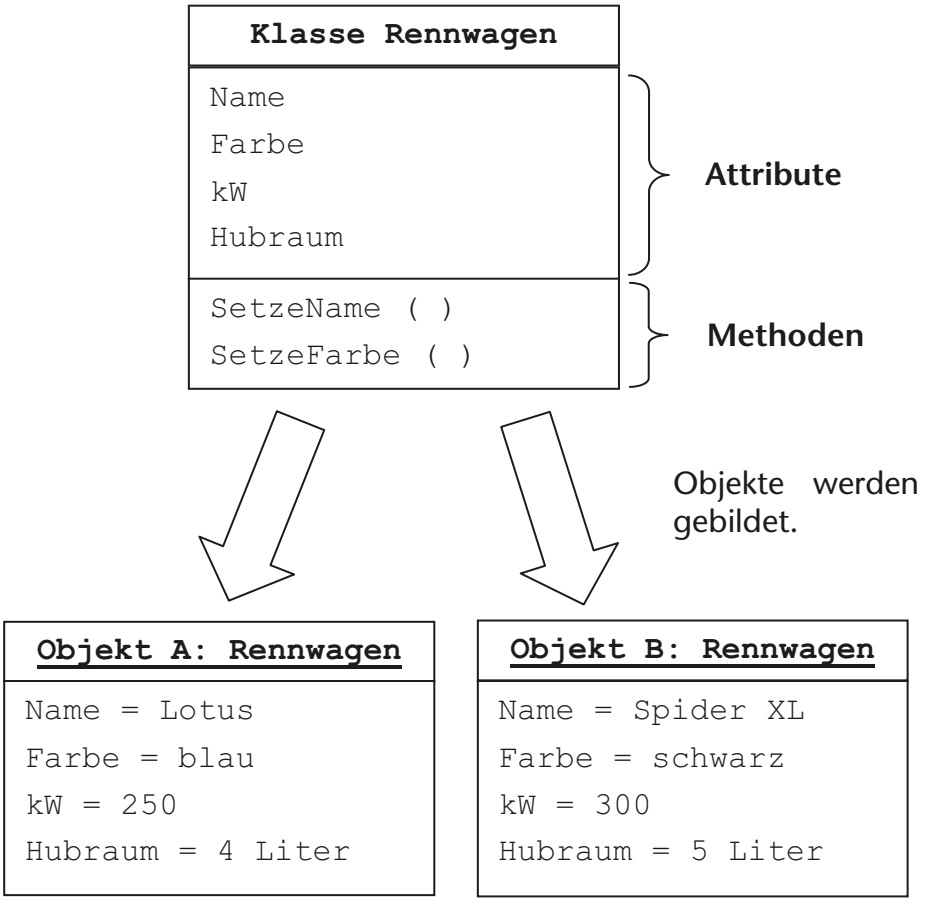
Diese Rennwagen sind konkrete Objekte. Sie haben Attribute wie Farbe, Leistung in kW und den Hubraum.



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.

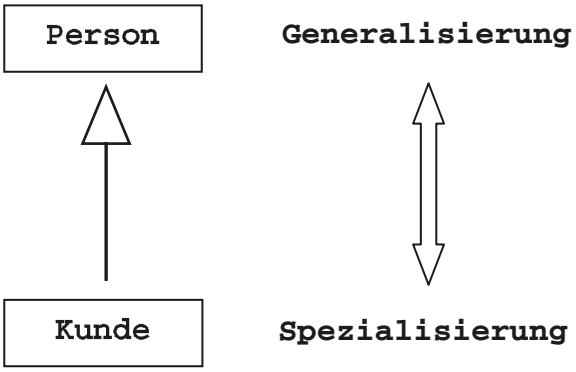
1.2.2 Generalisierung und Spezialisierung

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:

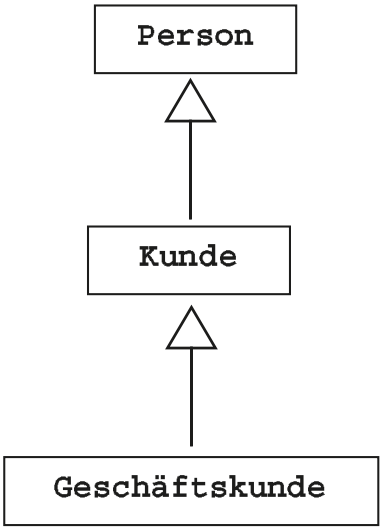
```
Definiere Klasse Person
   Geschützter Bereich:
   Attribut Name:    Typ Zeichenkette
   Öffentlicher Bereich:
   Methode SetzeName (Übergabeparameter N: Typ Zeichenkette)
      :
   Ende Methode
Ende Klasse

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.

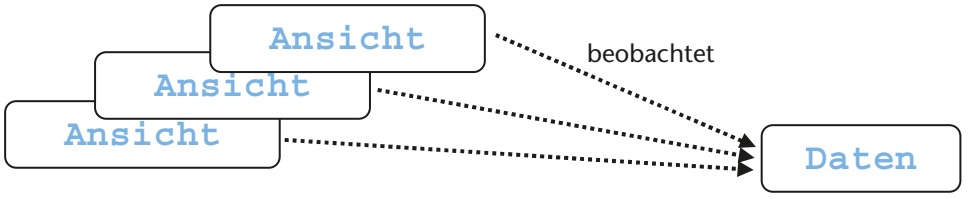
`Z(3)` ist zwar vom Typ `Person`, verweist aber auf ein Objekt vom Typ `Geschäftskunde`. Dank der Polymorphie wird trotzdem die korrekte Ausgabemethode der Klasse Geschäftskunde aufgerufen.

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.



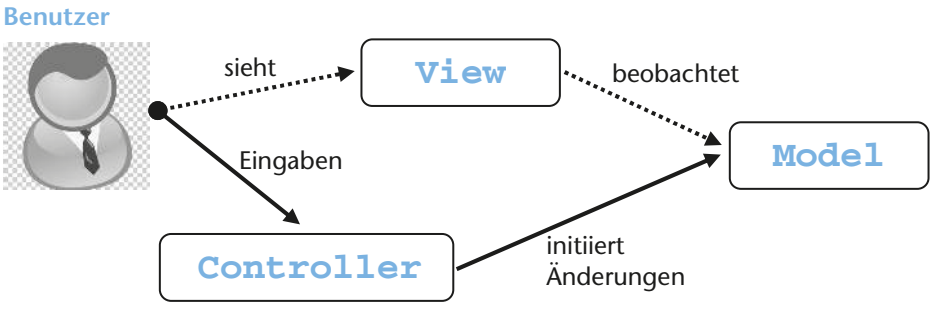
Das automatische Informieren der Ansichten ist ein großer Vorteil dieses Musters. Allerdings könnte es nachteilig sein, wenn sehr viele Ansichten das Datenobjekt beobachten. Da bei der kleinsten Änderung alle Ansichten informiert werden (auch die Ansichten, die diese Änderung möglicherweise nicht betreffen),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

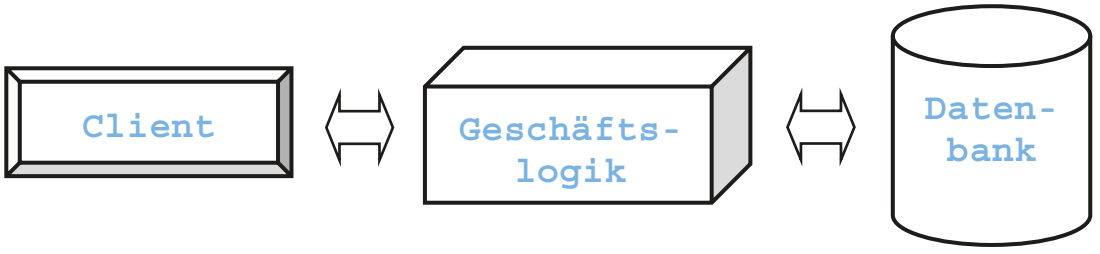
Dieses Konzept trennt ein System in eine Modell-Klasse, eine Ansichts-Klasse (oder mehrere Ansichts-Klassen) und eine Steuerungs-Klasse. Die Modell-Klasse repräsentiert die Daten des Systems, die mithilfe der Ansichts-Klassen in verschiedenen Sichtweisen dargestellt werden können. Angenommen, es handelt 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:



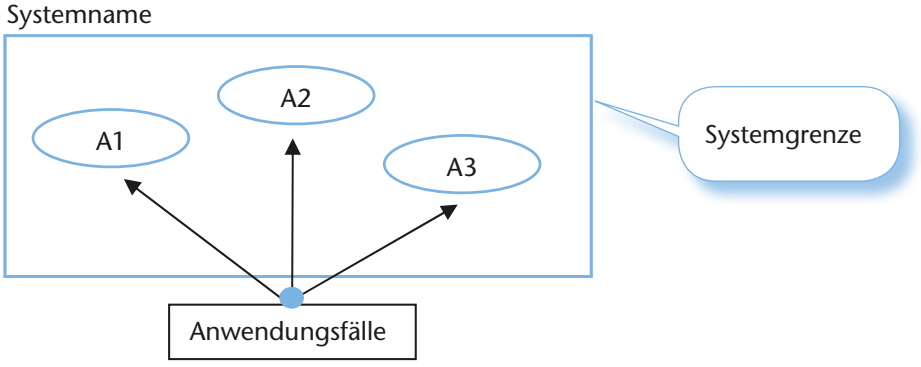
2 Das Anwendungsfall-diagramm

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*.