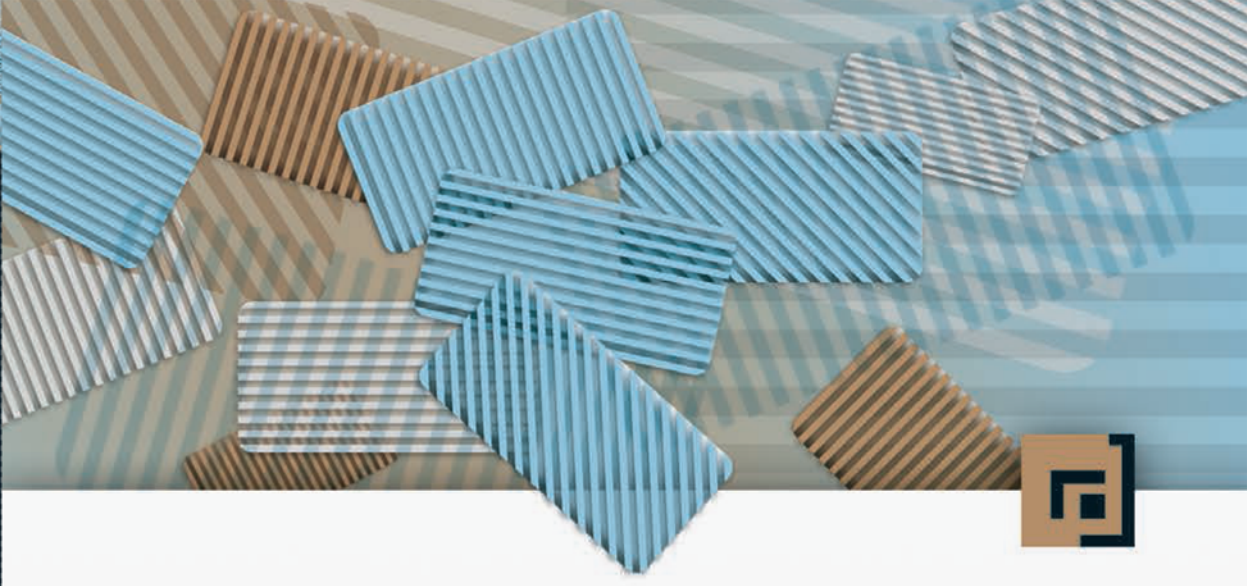

Martina Seidl · Marion Brandsteidl ·
Christian Huemer · Gerti Kappel

UML @ Classroom

Eine Einführung in die objektorientierte Modellierung

dpunkt.verlag

Martina Seidl ist Assistenzprofessorin am Institut für formale Modelle und Verifikation der Johannes Kepler Universität Linz und Research Associate in der Business Informatics Group der TU Wien.

Marion Brandsteidl unterrichtet an der TU Wien seit 2007 objektorientierte Modellierung. Als Senior Lecturer forscht sie nach neuen Lehrmethoden in diesem Bereich mit einem starken Fokus auf E-Learning.

Christian Huemer ist außerordentlicher Universitätsprofessor in der Business Informatics Group der TU Wien und wissenschaftlicher Leiter des Research Studio Interorganisational Systems der Research Studios Austria Forschungsgesellschaft.

Gerti Kappel ist seit 2001 Professorin für Wirtschaftsinformatik an der Technischen Universität Wien, wo sie die Business Informatics Group der Fakultät für Informatik leitet.

Martina Seidl · Marion Brandsteidl · Christian Huemer · Gerti Kappel

UML @ Classroom

Eine Einführung in die objektorientierte Modellierung

dpunkt.verlag

Martina Seidl
seidl@big.tuwien.ac.at

Marion Brandsteidl
brandsteidl@ifs.tuwien.ac.at

Christian Huemer
huemer@big.tuwien.ac.at

Gerti Kappel
gerti@big.tuwien.ac.at

Lektorat: Christa Preisendanz
Copy Editing: Ursula Zimpfer, Herrenberg
Satz: Autoren und Da-TeX, Leipzig
Herstellung: Nadine Thiele
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Buch 978-3-89864-776-2
PDF 978-3-86491-174-3
ePub 978-3-86491-175-0

1. Auflage 2012

Ringstraße 19B
69115 Heidelberg

5 4 3 2 1 0

Vorwort

Um hochwertige, moderne Software zu erstellen, die heutigen Anforderungen entspricht, sind Ad-hoc-Ansätze, bei denen »einfach darauf los programmiert« wird, überholt und nicht mehr zielführend. Vielmehr bedarf es einer klar strukturierten Vorgehensweise, die durch die Methoden des Software Engineering definiert und unterstützt wird.

Die Herausforderungen im Software Engineering sind vielfältig und gehen weit über Implementierungsaufgaben hinaus. Sie reichen von der Erfassung der Anforderungen über das Systemdesign bis hin zu Wartung und Weiterentwicklung der Software – um nur einige Stationen im Lebenszyklus von Software zu nennen. In einem Softwareentwicklungsprozess sind im Allgemeinen viele Personen involviert, die unterschiedliche Ausbildungen und Erfahrungen aufweisen. Sie benötigen eine gemeinsame Sprache, die alle verstehen und sprechen können, sodass ein Austausch möglich ist. Gleichzeitig soll diese Sprache möglichst präzise sein und nicht die Mehrdeutigkeiten einer natürlichen Sprache aufweisen. Zu diesem Zweck sind Modellierungssprachen entstanden. Sie dienen der Erstellung von Skizzen und Bauplänen für Softwaresysteme, die wiederum als Grundlage für die Erstellung bzw. für die automatische Generierung von ausführbarem Code dienen. Im Bereich der objektorientierten Softwareentwicklung konnte sich die Modellierungssprache *Unified Modeling Language (UML)* durchsetzen. Doch um diese Sprache richtig und effizient einzusetzen, muss man sie erst (kennen)lernen.

Seit 2006 bieten wir zweimal im Jahr die Lehrveranstaltung »Objektorientierte Modellierung« an der Technischen Universität Wien an. Diese Lehrveranstaltung ist für alle Studenten der Informatik und Wirtschaftsinformatik im 1. Studienjahr verpflichtend. So haben wir bis zu 1000 Studierende pro Jahr, die an unserer Lehrveranstaltung teilnehmen. Die Art und Weise, wie wir unsere Lehrveranstaltung organisieren, präsentierten wir in den Jahren 2008, 2009 und 2010 auf dem Educators' Symposium der MODELS-Konferenz [BSW+08, BSK09, BWH11].

Wir lehren die Grundlagen der objektorientierten Modellierung anhand von UML. Konkret betrachten wir das Klassendiagramm, das Sequenzdiagramm, das Zustandsdiagramm, das Aktivitätsdiagramm sowie das Anwendungsfalldiagramm und deren Zusammenhänge im Detail. Hierfür führen wir die *Syntax* (die Notation der Sprachelemente), die *Semantik* (die Bedeutung der Sprachelemente) und die *Pragmatik* (die Verwendungsweise der Sprachelemente) von UML ein. Diese fünf Diagramme decken die wesentlichsten Konzepte objektorientierter Modellierung ab und werden in vielen verschiedenen Teilen des Softwareentwicklungsprozesses eingesetzt. Die Lehrveranstaltung ist inhaltlich für Studierende ausgelegt, die die Grundkonzepte der objektorientierten Programmiersprachen wie Java oder C# bereits kennen, aber noch keine praktische Erfahrung im Software Engineering haben.

In diesem Buch fassen wir die Erfahrungen zusammen, die wir über die Jahre hinweg in unserer Lehrveranstaltung gesammelt haben. Dabei richten wir uns sowohl an jene Leser, die UML in kompakt dargestellter Form erlernen wollen, als auch an Lehrende, denen wir mit unserem umfangreichen Beispielrepertoire Inspiration für ihre eigenen Übungsaufgaben bieten wollen. Wir unterrichten UML möglichst nahe am Standard und illustrieren sämtliche Konzepte anhand von anschaulichen Beispielen. Ergänzt wird das Buch durch eine eigene Website, die u. a. den kompletten Foliensatz zum Buch in Wort und Schrift beinhaltet (www.uml.ac.at). Für Feedback, Anregungen etc. zum Buch sind wir unter info@uml.ac.at jederzeit erreichbar.

Materialien zum Buch: www.uml.ac.at

Kontakt: info@uml.ac.at

Die Modellierung ist ein sehr junges Gebiet der Informatik, das im vergangenen Jahrzehnt einen unglaublichen Wachstumsschub erfahren durfte. Der Einsatzbereich von Modellen geht weit über die Verwendung als reines Mittel zur Dokumentation hinaus. So lösen Techniken aus der Modellierung herkömmliche Programmierung mehr und mehr ab. Modelle sind bei Weitem mehr als nur Bilder, und Modellieren ist bei Weitem mehr als nur Zeichnen. In diesem Buch möchten wir eine solide Grundlage der wichtigsten objektorientierten Modellierungskonzepte und ein tief greifendes Verständnis vermitteln. Wir hoffen, Begeisterung und Interesse für dieses spannende und äußerst wichtige Gebiet der Informatik wecken zu können.

UML@Classroom ist ein Lehrbuch, das als kleines Geschwister zu UML@Work [HKKR05] zu sehen ist. Im Gegensatz zu UML@Work richtet es sich explizit an Einsteiger und Personen mit geringer oder keiner Modellierungserfahrung und führt grundlegende Konzepte auf sehr genaue Art und Weise ein, während auf

die Interpretation von selten auftretenden Sonderfällen und die Diskussion des UML-Standards weitgehend verzichtet wird. Bezüglich letzteren sei auf UML@Work verwiesen, das Personen mit Modellierungserfahrung als Zielgruppe anspricht.

Dem aufmerksamen Leser wird nicht entgangen sein, dass wir das Vorwort nicht geschlechtsneutral formuliert haben. Dies ist in keinem Fall als irgendeine Form der Diskriminierung aufzufassen. Vielmehr haben wir uns hier sowie im Rest dieses Buchs aus Gründen der Lesbarkeit dafür entschieden, die jeweils kürzere Form eines Worts zu verwenden. So reden wir im Folgenden von einem »Sekretär«, auch wenn »SekretärIn« gemeint ist, genauso wie wir »Vortragende« anstelle von »VortragendeR« verwenden.

Abschließend möchten wir uns bei jenen Personen bedanken, die in besonderem Maße zum Gelingen des Buchs beigetragen haben. Zuallererst gilt unser Dank unseren Familien, dass sie so lange das Buchprojekt mitgetragen haben. Auf das Konto von Katja Hildebrandt gehen alle Grafiken und tagelanges Korrekturlesen – herzlichen Dank dafür! Christa Preisendanz und der dpunkt.verlag haben wieder einmal Ausdauer und Hartnäckigkeit mit nicht eingehaltenen Deadlines bewiesen. Dafür gebührt ihnen unser besonderer Dank. Den Studierenden der Lehrveranstaltung »Objektorientierte Modellierung«, sowohl den vergangenen als auch den zukünftigen, danken wir für ihr zähes Hinterfragen. Damit findet Lernen immer wieder auf beiden Seiten des »Katheders« statt.

Linz und Wien, Mai 2012

Martina Seidl
Marion Brandsteidl
Christian Huemer
Gerti Kappel

Inhaltsverzeichnis

1 Einleitung

Die *Unified Modeling Language* (UML) bildet eine Vereinigung der Best Practices von Modellierungstechniken, die sich über Jahre hinweg etablieren konnten. UML erlaubt es, die unterschiedlichsten Aspekte eines Softwaresystems (z. B. Anforderungen, Datenstrukturen, Daten- und Informationsflüsse) innerhalb eines Rahmenwerks auf einheitliche Weise mittels objektorientierter Konzepte darzustellen. Bevor wir uns in die Tiefen von UML vorwagen, erläutern wir in diesem Kapitel zunächst kurz, warum Modellieren aus der Softwareentwicklung nicht mehr wegzudenken ist. Dafür gehen wir der Frage nach, was ein Modell ist und wofür Modelle gebraucht werden. Wir wiederholen kurz grundlegende Konzepte der Objektorientierung, bevor wir einen Überblick über den weiteren Aufbau des Buchs geben.

Unified Modeling Language (UML)

1.1 Motivation

Stellen Sie sich vor, Sie möchten ein Softwaresystem entwickeln, das ein Kunde bei Ihnen in Auftrag gegeben hat. Eine der ersten Herausforderungen, mit der Sie konfrontiert werden, ist das Klären der Fragen, was der Kunde eigentlich genau will und ob Sie die genauen Anforderungen des Kunden an das zukünftige System korrekt erfasst haben. Schon von diesem ersten Schritt hängt das Gelingen oder das Scheitern Ihres Projekts ab. Sofort stellt sich die Frage, wie Sie mit Ihrem Kunden kommunizieren. Natürliche Sprache stellt hierbei nicht unbedingt die erste Wahl dar, da diese unpräzise und mehrdeutig ist. Missverständnisse können sehr leicht auftreten, und die Gefahr ist sehr groß, dass Leute mit unterschiedlichen Hintergründen (z. B. Informatiker und Betriebswirt) aneinander vorbeireden, was fatale Folgen haben kann.

Was Sie jetzt benötigen, ist die Möglichkeit, ein Modell für Ihre Software zu erstellen, das die wesentlichen Aspekte in einer möglichst einfachen und eindeutigen Notation hervorhebt, aber von irrelevanten Details abstrahiert, genauso wie es bei Modellen in

der Architektur, wie z. B. den Bauplänen, passiert. Zum Beispiel enthält ein Bauplan für ein Gebäude zwar Informationen, wie der Grundriss des zu bauenden Hauses auszusehen hat, die zu verwendenden Baustoffe werden hier aber nicht angegeben, da sie im Moment keinerlei Bedeutung haben und den Plan unnötig aufblasen würden. Genauso finden wir hier keinerlei Informationen, wie die elektrischen Leitungen zu verlegen sind. Hierfür wird ein eigener Plan erstellt, um nicht zu viele Informationen auf einmal darzustellen. Auch in der Informatik ist es wie in der Architektur wichtig, dass Leute mit unterschiedlichen Hintergründen (z. B. Architekt und Bauherr) das Modell lesen, interpretieren und umsetzen können.

Modellierungssprache

Genau für solche Szenarien wurden *Modellierungssprachen* entwickelt, die klar definierte Regeln zur strukturierten Beschreibung eines Systems aufweisen. Solche Sprachen können *textuell* (z. B. eine Programmiersprache wie Java) oder *visuell* sein (z. B. eine Sprache, die miteinander verbindbare Symbole für Transistoren, Dioden etc. zur Verfügung stellt). Modellierungssprachen können speziell für eine bestimmte Domäne, z. B. für die Beschreibung von Webanwendungen, bestimmt sein. Diese *domänenspezifischen Modellierungssprachen* geben Möglichkeiten und Richtlinien zur Problemlösung in einem bestimmten Bereich vor, können dafür aber auch stark einschränkend sein. Oder Modellierungssprachen sind universell einsetzbar. Die Sprache UML, die wir in diesem Buch betrachten, gehört der Kategorie der universell einsetzbaren Sprachen an. Anhand von UML lernen wir die wichtigsten Konzepte der objektorientierten Modellierung kennen.

Domänenspezifische Modellierungssprache

Universelle Modellierungssprache

Objektorientierte Modellierung

Objektorientierte Modellierung ist eine Form der Modellierung, die dem objektorientierten Paradigma gehorcht. In den nachfolgenden zwei Unterkapiteln gehen wir auf den Modellbegriff und die wesentlichen objektorientierten Konzepte kurz ein, um uns dann der objektorientierten Modellierung mit UML widmen zu können.

1.2 Modell

System

Modelle erlauben eine effiziente und elegante Beschreibung von Systemen. Ein *System* ist eine Gesamtheit von Elementen, die so aufeinander bezogen sind und in einer Weise wechselseitig wirken, dass sie als eine aufgaben-, sinn- oder zweckgebundene Einheit angesehen werden können und sich in dieser Hinsicht gegenüber der sie umgebenden Umwelt abgrenzen [Wik11]. Beispiele für Sys-

teme sind materielle Dinge wie Autos oder Flugzeuge, ökologische Umgebungen wie Seen und Wälder, aber auch Organisationseinheiten wie eine Universität. Im Bereich der Informatik sind wir speziell an Softwaresystemen und damit einhergehend an Modellen interessiert, die Softwaresysteme beschreiben.

Softwaresystem

Softwaresysteme selbst basieren auf *Abstraktionen*, die eine maschinenverarbeitbare Form von Sachverhalten der Realität darstellen. Unter Abstraktion versteht man in diesem Zusammenhang eine Verallgemeinerung, das Absehen vom Besonderen und Einzelnen, das Loslösen vom Dinglichen. Abstraktion ist das Gegenteil von Konkretisierung. Unter Abstrahieren versteht man dementsprechend das Abgehen vom Konkreten, das Herausheben des Wesentlichen aus dem Zufälligen, das Erkennen gleicher Merkmale [Bal09].

Abstraktion

Bei der Erstellung von Softwaresystemen ist die Wahl geeigneter Abstraktionsmittel einerseits für die Umsetzung, andererseits aber auch für deren spätere Benutzung extrem wichtig. Wählt man die richtigen Abstraktionsmittel, so geht das Programmieren leicht von der Hand. Die einzelnen Teile haben dann einfache und kleine Schnittstellen. Neue Funktionalität kann ohne größere Reorganisation eingeführt werden. Wählt man die falschen Abstraktionsmittel, so ergeben sich bei der Umsetzung eine Vielzahl von bösen Überraschungen: Die Schnittstellen werden schwerfällig und Änderungen sind nur mühsam einzupflegen. Nur durch geeignete Abstraktionsmittel ist es möglich, mit der ständig steigenden Komplexität von modernen Softwaresystemen umzugehen [Jac11]. Gerade die Modellierung kann hier wertvolle Dienste leisten.

Wahl von Abstraktionsmittel

Um ein besseres Verständnis für den Modellbegriff zu entwickeln, werden nachfolgend weitverbreitete und allgemein anerkannte Definitionen des Modellbegriffs sowie die Eigenschaften, die ein gutes Modell aufweisen soll, vorgestellt.

Der Begriff *Modell* spielt nicht nur in der Informatik, sondern auch in vielen anderen wissenschaftlichen Disziplinen (Mathematik, Philosophie, Psychologie, Wirtschaftswissenschaften etc.) eine bedeutende Rolle. Abgeleitet von dem lateinischen Wort »modulus«, das einen Maßstab in der Architektur bezeichnet, wurde in Italien während der Renaissance das Wort »modello« für ein Anschauungsobjekt verwendet, anhand dessen Auftraggeber Form und Gestaltung eines geplanten Gebäudes vorgeführt bekamen sowie konstruktive und architektonische Fragestellungen geklärt wurden [FHR08]. Über die nachfolgenden Jahrhunderte fand der Begriff »Modell« Einzug in diverse Wissenschaftszweige, und zwar

Modellbegriff

zur vereinfachenden Beschreibung von komplexen Sachverhalten aus der Realität.

Definition von H. Stachowiak

1973 schlug Herbert Stachowiak einen Modellbegriff vor, der durch drei Merkmale gekennzeichnet ist [Sta73]:

1. *Abbildung.* Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
2. *Verkürzung.* Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellierer bzw. dem Benutzer des Modells relevant erscheinen.
3. *Pragmatismus.* Pragmatismus bedeutet so viel wie Orientierung am Nützlichen. Die Zuordnung eines Modells zum Original wird durch die Fragen *Für wen?*, *Warum?* und *Wozu?* getroffen. Ein Modell wird vom Modellierer bzw. von seinem Benutzer innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Zweck anstelle eines Originals eingesetzt. Das Modell wird somit interpretiert.

Modelle unterstützen eine auf das Wesentliche reduzierte Darstellung eines Systems, um seine Komplexität auf handhabbare Teilaspekte zu verringern. Ein System wird üblicherweise nicht durch eine einzige Sicht beschrieben, sondern durch eine Vielzahl von Sichten, die zusammen ein einheitliches Gesamtbild ergeben. So kann sich eine Sicht auf die Beschreibung der involvierten Objekte und ihre Beziehungen zueinander beziehen, eine andere Sicht hingegen kann die Beschreibung des Verhaltens einer Gruppe von Objekten beinhalten oder die Interaktionen zwischen unterschiedlichen Objekten darstellen.

Eigenschaften von Modellen

Modelle müssen sorgfältig und wohlbedacht erstellt werden. Nach Bran Selic [Sel03] bestimmen folgende fünf Charakteristika die Qualität eines Modells:

- *Abstraktion.* Ein Modell ist immer eine reduzierte Darstellung des Systems, das es abbildet. Indem für eine Sicht irrelevante Details versteckt oder entfernt werden, ist es für den Benutzer leichter, die Essenz des Ganzen zu erfassen.
- *Verständlichkeit.* Es ist unzureichend, sich zur Erhöhung der Verständlichkeit auf das Weglassen irrelevanter Details zu beschränken. Vielmehr noch ist es wichtig, dass die verbleibenden Elemente in einer möglichst intuitiven Form, z. B. in einer grafischen Notation, dargestellt sind. Die Verständlichkeit ergibt sich direkt aus der Ausdrucksstärke der Modellierungssprache. Ausdrucksstärke kann als die Fähigkeit de-

finiert werden, einen komplexen Sachverhalt mit so wenig Information wie möglich darzustellen. Auf diese Weise reduziert ein gutes Modell den intellektuellen Aufwand, der notwendig ist, um den abgebildeten Sachverhalt zu verstehen. Zum Beispiel sind typische Programmiersprachen nicht besonders ausdrucksstark für den menschlichen Leser, da sehr viel Aufwand notwendig ist, um den Inhalt des Programms zu verstehen.

- *Genauigkeit.* Ein Modell muss eine möglichst realitätsgetreue Abbildung der relevanten, d. h. der nicht wegabstrahierten Eigenschaften des modellierten Systems ermöglichen.
- *Vorhersagewert.* Durch ein Modell muss es möglich sein, interessante, aber nicht offensichtliche Eigenschaften des modellierten Systems vorherzusagen. Dies kann entweder durch Simulation oder durch Analyse formaler Eigenschaften passieren.
- *Verhältnismäßigkeit.* Die Erstellung des Modells muss billiger sein als die Erstellung des modellierten Systems.

Modelle können für unterschiedliche Zwecke eingesetzt werden. So unterscheiden wir zwischen *deskriptiven* und *präskriptiven* Modellen [FHR08]. *Deskriptive Modelle* beschreiben das Element der Wirklichkeit, um einen gewissen Aspekt leichter verständlich zu machen. Zum Beispiel beschreibt ein Stadtplan eine Stadt derart, dass eine ortsunkundige Person Routen innerhalb dieser Stadt leichter finden kann. *Präskriptive Modelle* hingegen dienen dazu, eine Bauanleitung zur Erstellung eines zu entwickelnden Systems zu bieten.

Deskriptives Modell

Präskriptives Modell

In diesem Buch betrachten wir, wie die unterschiedlichen Aspekte eines Softwaresystems mithilfe einer Modellierungssprache, der Unified Modeling Language, modelliert werden können, sodass daraus ausführbarer Code manuell oder (teil-)automatisch abgeleitet bzw. eine leicht verständliche Dokumentation erstellt werden kann. Übrigens stellt auch der ausführbare Code, der in einer beliebigen Programmiersprache wie z. B. Java entwickelt wurde, wiederum ein Modell dar. Dieses Modell repräsentiert die zu lösende Problemstellung und ist für die Ausführung auf Computer optimiert. Zusammengefasst gibt es für Modelle folgende drei Verwendungsarten [Fow03b]:

Ausführbarer Code als Modell

- Modelle als Skizze
- Modelle als Blaupause
- Modelle als ausführbare Programme

Modelle als Skizze

Modelle werden als *Skizze* verwendet, um gewisse Aspekte eines Systems auf einfache Weise zu kommunizieren. Hierbei geht es nicht um eine vollständige Abbildung des Systems. Vielmehr zeichnen sich Skizzen durch ihre Selektivität aus, indem sie auf das für die Lösung eines Problems Wesentliche reduziert sind. Oft werden durch die Skizze alternative Lösungsansätze sichtbar, die dann im Entwicklerteam diskutiert werden. Modelle dienen also auch als Diskussionsgrundlage für Mitglieder eines Teams.

Modelle als Blaupause

Im Gegensatz zum Einsatz von Modellen als Skizze spielt Vollständigkeit eine große Rolle, wenn Modelle als *Blaupause* eingesetzt werden. Diese müssen genügend Details enthalten, sodass Entwickler daraus lauffähige Systeme erstellen können, ohne Entwurfsentscheidungen treffen zu müssen. Modelle als Blaupausen spezifizieren oft nicht das gesamte System, sondern nur bestimmte Teile. Zum Beispiel werden die Schnittstellendefinitionen zwischen Subsystemen festgelegt, wobei die internen Implementierungsdetails den Entwicklern überlassen werden. Handelt es sich bei den Modellen um Verhaltensbeschreibungen, so können diese auch simuliert und getestet und damit Fehler im Vorfeld erkannt werden.

Forward Engineering
Backward Engineering

Modelle als Skizzen und als Blaupausen können sowohl für *Forward Engineering* als auch für *Backward Engineering* eingesetzt werden. Beim Forward Engineering dient das Modell als Grundlage für die Erstellung von Code, während beim Backward Engineering das Modell aus dem Code generiert wird, um diesen auf leicht verständliche und übersichtliche Weise zu dokumentieren.

Modelle als ausführbare Programme

Schließlich können Modelle als *ausführbare Programme* genutzt werden. Dies bedeutet, dass Modelle so genau spezifiziert werden, dass daraus automatisch Code generiert werden kann. Im Kontext von UML wurde die modellgetriebene Softwareentwicklung in den letzten Jahren extrem populär, die ein Verfahren bietet, UML als Programmiersprache zu verwenden. Diese werden wir kurz in Kapitel 9 dieses Buchs kennenlernen, nachdem wir uns die Grundlagen von UML angeeignet haben. In manchen Praxisbereichen wie z. B. bei der Entwicklung von eingebetteten Systemen werden Modelle bereits anstelle von traditionellen Programmiersprachen eingesetzt. In anderen Bereichen findet rege Forschung statt, um die Entwicklung von Softwaresystemen auf eine neue und damit leichter wartbare und weniger fehleranfällige Abstraktionsebene zu heben.

1.3 Objektorientierung

Wenn wir objektorientiert modellieren wollen, müssen wir zuerst klären, was unter *Objektorientierung* zu verstehen ist. Die Motivation für die erstmalige Einführung der Objektorientierung in den 60er-Jahren in Form der Simulationssprache SIMULA [Hol94] war ein für den Menschen möglichst natürliches Paradigma zu verwenden, um die Welt zu beschreiben. Die objektorientierte Betrachtungsweise entspricht unserem Denken über die reale Welt. Es handelt sich dabei um ein Denken über eine Gesellschaft von autonomen Individuen, die einen festen Platz in dieser Gesellschaft einnehmen und dabei vordefinierte Pflichten erfüllen müssen.

Objektorientierung

Es gibt nicht eine festgelegte Definition für den Begriff Objektorientierung. Es gibt allerdings einen allgemein anerkannten Konsens, welche Eigenschaften Objektorientierung charakterisieren. Natürlich spielen bei objektorientierten Ansätzen Objekte eine zentrale Rolle. Vereinfacht gesehen sind Objekte Elemente im System, deren Daten und Operationen beschrieben werden und die miteinander interagieren und kommunizieren können. Im Allgemeinen erwartet man von einem objektorientierten Ansatz folgende Konzepte.

Klasse. In objektorientierten Ansätzen muss es möglich sein, *Klassen* zu definieren, die die Attribute und das Verhalten einer Menge von Objekten, den Instanzen einer Klasse, abstrakt beschreiben und so Gemeinsamkeiten von Objekten zusammenfassen. Zum Beispiel haben Personen einen Namen, eine Adresse und eine Sozialversicherungsnummer. Lehrveranstaltungen haben eine eindeutige Nummer, einen Titel und eine Beschreibung. Hörsäle haben eine Bezeichnung sowie einen Standort usw. Zusätzlich definiert eine Klasse eine Menge von zulässigen Operationen, die auf ihre Instanzen angewendet werden dürfen. Zum Beispiel kann man einen Hörsaal für einen gewissen Termin reservieren, ein Student kann sich für eine Prüfung anmelden etc. Auf diese Weise wird das Verhalten von Objekten beschrieben.

Klasse

Objekt. Die Instanzen einer Klasse werden als ihre *Objekte* bezeichnet. Beispielsweise ist »Hörsaal 1 der TU Wien« eine konkrete Ausprägung der Klasse Hörsaal. Ein Objekt zeichnet sich speziell dadurch aus, dass es eine eigene Identität aufweist, d. h., unterschiedliche Instanzen einer Klasse sind eindeutig identifizierbar. Zum Beispiel handelt es sich bei dem Beamer in Hörsaal 1 um ein anderes Objekt als bei dem Beamer in Hörsaal 2, auch wenn es

Objekt

baugleiche Geräte sind. Wir sprechen hier von *gleichen* Geräten, nicht aber vom *selben* Gerät. Anders verhält es sich bei konkreten Werten von Datentypen: Die Zahl »1«, die eine konkrete Ausprägung des Datentyps Integer darstellt, besitzt keine eigene Identität.

Ein Objekt befindet sich immer in einem bestimmten Zustand. Ein Zustand wird ausgedrückt durch die Werte seiner Attribute. Ein Hörsaal kann sich z. B. in dem Zustand »belegt« oder »frei« befinden. Außerdem weist ein Objekt Verhalten auf. Das Verhalten eines Objekts wird durch die Menge seiner Operationen beschrieben. Operationen werden durch das Senden einer Nachricht aktiviert.

Kapselung

Kapselung. Die *Kapselung* bezeichnet den Schutz vor unerlaubtem Zugriff auf den internen Zustand eines Objekts durch eine eindeutig definierte Schnittstelle. Verschiedene Ebenen der Sichtbarkeit der Schnittstellen helfen, verschiedene Zugriffsberechtigungen zu definieren. So gibt es in Java z. B. die Sichtbarkeiten `public`, `private` und `protected`, die den Zugriff für alle, nur innerhalb des Objekts und nur für Angehörige derselben Klasse bzw. aller Unterklassen zulassen.

Nachricht

Nachricht. Objekte kommunizieren miteinander durch *Nachrichten*. Eine Nachricht an ein Objekt stellt eine Aufforderung dar, eine Operation auszuführen. Das Objekt selbst entscheidet, ob und wie es diese Operation ausführt. Die Operation wird nur dann ausgeführt, wenn der Sender zum Aufruf der Operation berechtigt ist, was in Form von Sichtbarkeiten (siehe vorhergehender Absatz) geregelt werden kann, und wenn eine geeignete Implementierung vorhanden ist. Meist wird das Konzept des *Überladens* unterstützt, das es ermöglicht, eine Operation für verschiedene Typen von Parametern unterschiedlich zu definieren. Zum Beispiel verhält sich der Operator »+« für die Addition von ganzen Zahlen anders (z. B. `1 + 1 = 2`) als für die Konkatenation von Zeichenketten (z. B. `'a' + 'b' = 'ab'`).

Überladen

Vererbung

Vererbung. Das Konzept der *Vererbung* ist ein Mechanismus, neue Klassen aus existierenden Klassen abzuleiten. Eine aus einer bestehenden Klasse (= Oberklasse) abgeleitete Unterklasse erbt alle Attribute und Operationen (Spezifikation und Implementierung) der Oberklasse. Eine Unterklasse kann

- neue Attribute und/oder Operationen definieren,
- die Implementierung geerbter Operationen überschreiben und
- geerbte Operationen durch eigenen Code ergänzen.

Die Vererbung ermöglicht eweiterbare Klassen und in weiterer Folge den Aufbau von *Klassenhierarchien* als Basis objektorientierter Systementwicklung. Eine Hierarchie besteht aus Klassen mit ähnlichen Eigenschaften (z. B. Person ← Mitarbeiter ← Professor ← ...). *Klassenhierarchie*

Bei richtigem Einsatz bietet Vererbung eine Vielzahl von Vorteilen: Wiederverwendung von Programm- bzw. Modellteilen, wodurch Redundanzen und Fehler vermieden werden, konsistente Definition von Schnittstellen, Modellierungshilfe durch eine natürliche Kategorisierung der auftretenden Elemente und Unterstützung von inkrementeller Entwicklung, d. h. schrittweise Verfeinerung von allgemeinen Konzepten zu speziellen Konzepten.

Polymorphismus. Im Allgemeinen versteht man unter *Polymorphismus* die Fähigkeit, verschiedene Gestalt anzunehmen. Ein polymorphes Attribut kann im Laufe der Ausführung eines Programms Referenzen auf Objekte von verschiedenen Klassen haben. Bei der Deklaration dieses Attributs wird statisch zur Compilezeit eine Klasse (z. B. Person) zugeordnet. Zur Laufzeit kann dieses Attribut dynamisch auch an eine Unterklasse gebunden sein (z. B. Mitarbeiter oder Student). *Polymorphismus*

Eine polymorphe Operation kann auf Objekte verschiedener Klassen ausgeführt werden und jeweils eine andere Semantik haben. Dies kann auf mehrere Arten realisiert werden: (i) Bei *parametrisiertem Polymorphismus*, besser bekannt als Generizität, werden den Operationen die konkreten Klassen als Argumente übergeben, (ii) beim *Inklusionspolymorphismus* sind Operationen auf Klassen und auf ihre direkten und indirekten Unterklassen anwendbar, (iii) durch Umsetzung mittels *Überladen von Operationen* und (iv) mittels *Coercion*, also der Umwandlung von Typen. Bei den ersten zwei Realisierungsarten spricht man vom *universellen Polymorphismus*, die anderen beiden Arten werden als *Adhoc-Polymorphismus* bezeichnet [CW85].

UML basiert durchgängig auf objektorientierten Konzepten. Dies fällt besonders beim Klassendiagramm auf, das sehr einfach in eine objektorientierte Programmiersprache übersetzt werden kann. Das Klassendiagramm und mögliche Übersetzungen nach Java lernen wir in Kapitel 4 kennen.

1.4 Aufbau des Buchs

Nachdem wir uns in Kapitel 2 einen kurzen Überblick über UML verschafft haben, indem wir dessen Entstehungsgeschichte näher beleuchten und die 14 unterschiedlichen Diagramme kurz betrachtet haben, werden in Kapitel 3 die Konzepte des Anwendungsfalldiagramms eingeführt. Damit sind wir in der Lage, die Anforderungen, die an ein zu entwickelndes System gestellt werden, zu beschreiben. In Kapitel 4 wird das Klassendiagramm vorgestellt, das uns erlaubt, die Struktur eines Systems zu beschreiben. Um nun zusätzlich auch das Verhalten des Systems abzubilden, werden in Kapitel 5 das Zustandsdiagramm, in Kapitel 6 das Sequenzdiagramm und in Kapitel 7 das Aktivitätsdiagramm eingeführt. Das Zusammenspiel der unterschiedlichen Diagrammarten wird in Kapitel 8 anhand mehrerer Beispiele erläutert. Weiterführende Konzepte, die für den praktischen Einsatz von UML von maßgeblicher Bedeutung sind, werden abschließend in Kapitel 9 kurz betrachtet.

Da der UML-Standard auf Englisch verfasst wurde und auch die Modellierung – genauso wie viele andere Gebiete der Informatik – Englisch als Arbeitssprache einsetzt, werden die Bezeichnungen der Sprachelemente sowohl in Deutsch als auch in Englisch angegeben. Anhang A fasst alle englischen Originalbegriffe und die verwendeten deutschen Begriffe zusammen.

Die Konzepte werden durchgängig anhand von Beispielen erklärt. Alle Beispiele stammen aus dem universitären Bereich. Sie stellen meistens stark vereinfachte Szenarien dar. Wir wollen in diesem Buch kein durchgängiges System modellieren, da die Gefahr sehr hoch ist, dass wir uns in einer Vielzahl von technischen Details verlieren. Wir haben die Beispiele vielmehr entsprechend ihrem didaktischen Nutzen und entsprechend ihrer illustrativen Ausdrucksstärke gewählt. In vielen Fällen wurden daher Annahmen getroffen, die aus didaktischen Gründen auf vereinfachten Darstellungen der Wirklichkeit beruhen.

2 Eine kurze Tour durch UML

Bevor wir in den nächsten Kapiteln die wichtigsten Konzepte von UML kennenlernen, beschäftigen wir uns in diesem Kapitel mit den Hintergründen dieser Modellierungssprache. Dazu gehen wir darauf ein, wie UML entstanden ist und was es mit dem »U« für »unified« eigentlich auf sich hat. Dann gehen wir der Frage nach, wie UML selbst definiert ist, d. h., woher kommen die Regeln, die uns sagen, wie ein gültiges Modell in UML auszusehen hat? Und wofür wird UML überhaupt verwendet? Schließlich geben wir noch einen kurzen Überblick über die gesamten 14 Diagramme von UML in der aktuellen Version 2.3, die sich sowohl für Struktur- als auch für Verhaltensmodellierung einsetzen lassen.

2.1 Historische Entwicklung

Ursprünge der Objektorientierung

Die Einführung von objektorientierten Konzepten in der Informatik geht auf Arbeiten der frühen 60er-Jahre des vergangenen Jahrhunderts zurück [Cap03]. Die ersten Ideen fanden in Systemen wie z. B. Sketchpad ihre Umsetzung, das einen neuen, grafischen Kommunikationsansatz zwischen Mensch und Computer darstellte [Kay93, Sut64].

SIMULA

Die Programmiersprache SIMULA [Hol94] wird heutzutage als erste objektorientierte Programmiersprache angesehen. SIMULA wurde in erster Linie zur Entwicklung von Simulationssoftware eingesetzt und erfuhr noch einen verhältnismäßig geringen Verbreitungsgrad. Konzepte wie Klassen, Objekte, Vererbung und dynamisches Binden waren in SIMULA bereits realisiert [BDMN79].

Objektorientierte Programmiersprachen

Dies war der Anfang einer Revolution in der Softwareentwicklung. Basierend auf dem objektorientierten Paradigma entstand in den darauffolgenden Jahrzehnten eine Vielzahl von Programmiersprachen [GJ97]. Darunter befanden sich Sprachen wie C++ [Str09], Eiffel [Mey88] und Smalltalk [Kay93], die bereits viele wichtige Konzepte moderner Programmiersprachen enthielten und die zum Teil heute noch eingesetzt werden.

Die Entstehung und Einführung der Objektorientierung als Methode im Software Engineering ist eng mit dem Aufkommen von objektorientierten Programmiersprachen verbunden. Objektorientierung gilt inzwischen als gut etablierter und erprobter Ansatz, um mit der Komplexität von Softwaresystemen umzugehen, und hat nicht nur in Programmiersprachen, sondern auch in anderen Bereichen wie z. B. bei Datenbanken oder der Beschreibung von Benutzerschnittstellen ihre Daseinsberechtigung gefunden.

Wie wir bereits bei der Definition des Modellbegriffs in Abschnitt 1.2 festgestellt haben, sind Softwaresysteme Abstraktionen, also stark vereinfachte Darstellungen, die es zum Ziel haben, Probleme der realen Welt maschinenunterstützt zu lösen. Um die reale Welt adäquat zu beschreiben, sind herkömmliche prozedurale Programmiersprachen nicht unbedingt das einfachste Mittel, da die konzeptionellen Unterschiede zwischen einer natürlichen Problembeschreibung und der praktischen Umsetzung als Programm groß sind. Mit der objektorientierten Programmierung, deren Programmierkonzepte sich an der realen Welt orientieren, versuchte man, bessere und vor allem besser wartbare Programme zu entwickeln [Cap03].

Die Objektorientierung ist über die Jahre zum wichtigsten Softwareentwicklungsparadigma geworden, was sich heute in objektorientierten Programmiersprachen wie Java [Mös11] oder C# [Mös09] und objektorientierten Modellierungssprachen wie UML widerspiegelt. Doch bis zum heutigen State-of-the-Art der Softwareentwicklung war es ein langer, kurvenreicher Weg.

Ada

In den 80er-Jahren erfuhr die Programmiersprache Ada, die vom amerikanischen Verteidigungsministerium entwickelt und propagiert wurde, aufgrund ihrer mächtigen Konzepte und effizienten Compiler starke Popularität [IBFW86]. Ada bot schon damals Unterstützung für abstrakte Datentypen in Form von *Packages* und Unterstützung von Nebenläufigkeit in Form von *Tasks*. Packages erlaubten die Trennung von Spezifikation und Implementierung und die Erweiterung der Sprache um Objekte und um Klassen von Objekten. Ada unterschied sich damit grundlegend von anderen strukturierten Sprachen wie Fortran und Cobol. Um nun Ada-Programme leichter entwickeln zu können, wurde der Ruf nach objektorientierten Analyse- und Entwurfsmethoden laut. Diese Modellierungsmethoden wurden aufgrund der weiten Verbreitung von Ada und durch den Druck des amerikanischen Verteidigungsministeriums speziell auf die Charakteristika von Ada ausgerichtet. Grady Booch war einer der ersten Forscher, der

Methode von Booch

Arbeiten zu dem *objektorientierten Entwurf von Ada-Programmen* veröffentlichte [Boo86].

Mit der Zeit entstand eine Vielzahl von weiteren objektorientierten Modellierungsmethoden (siehe [Cap03, KS96] für einen detaillierten Überblick). Die Modellierungsmethoden hatten meist entweder einen starken Bezug zu Programmiersprachen wie die Methode von Booch oder einen starken Bezug zur Datenmodellierung wie OMT (Object Modeling Technique) von James Rumbaugh [RBP+91]. OMT unterstützte die Entwicklung komplexer Objekte im Sinne einer objektorientierten Erweiterung des Entity-Relationship-Modells [Che76], das für die Beschreibung von Datenbanken eingeführt worden war. Davon unabhängig führte Ivar Jacobson seinen Ansatz *Object-Oriented Software Engineering* (OOSE) ein, der ursprünglich zur Beschreibung von Telekommunikationssystemen entwickelt worden war [JCJO92]. OOSE stellte die Modellierung und die Simulation von Vorgängen der realen Welt in den Mittelpunkt.

OMT-Ansatz von Rumbaugh

OOSE-Ansatz von Jacobson

In den 80er- und frühen 90er-Jahren des vergangenen Jahrhunderts wurde die Modellierungswelt von einer Vielzahl verschiedener Modellierungssprachen überflutet. Um die auftretenden Kompatibilitätsprobleme in den Griff zu bekommen, war erheblicher Aufwand notwendig. Die Modelle von verschiedenen Projektpartnern waren oft nicht kompatibel und die Wiederverwendbarkeit in anderen Projekten war nicht gegeben. Erschöpfende Diskussionen über unterschiedliche Notationen waren die Folge, die von den eigentlichen Modellierungsproblemen ablenkten. Da immer wieder neue Modellierungssprachen auftauchten, war nicht klar, in welche investiert werden sollte und welche nur einen kurzlebigen Trend darstellten. Setzte sich eine Sprache nicht durch, waren sämtliche Investitionen, die für die Etablierung einer Sprache innerhalb eines Projekts oder eines Unternehmens getätigt worden waren, weitgehend verloren. Diese Zeit der zahlreichen, oft nur in Details unterschiedlichen Ansätze wird rückblickend auch als *Methodenkrieg* bezeichnet.

Zeit des Methodenkriegs

Um dieser unzulänglichen Situation ein Ende zu setzen, rief 1996 die *Object Management Group* (OMG) [Obj], das wichtigste Standardisierungsgremium für objektorientierte Softwareentwicklung, zur Spezifikation eines einheitlichen Modellierungsstandards auf.

Object Management Group (OMG)

Bereits im Jahr 1995 vereinten die Wissenschaftler Grady Booch, Ivar Jacobson und James Rumbaugh auf der OOPSLA-Konferenz (das Akronym OOPSLA steht für Object-Oriented Programming, Systems, Languages, and Applications) ihre Ideen und

Drei Amigos

Ansätze. Seit damals werden Booch, Jacobson und Rumbaugh oft auch als die »drei Amigos« bezeichnet. Dabei steckten sie sich folgende Ziele [BM99]:

- Verwendung von objektorientierten Konzepten zur Repräsentation von kompletten Systemen und nicht nur einem Teil der Software;
- Herstellung einer expliziten Bindung zwischen Modellierungskonzepten und ausführbarem Programmcode;
- Berücksichtigung von Skalierungsfaktoren, die in komplexen und kritischen Systemen inhärent sind;
- Erstellung einer Modellierungssprache, die sowohl maschinenverarbeitbar als auch für Menschen lesbar ist.

Unified Modeling Language (UML)

Das Resultat ihrer Anstrengungen war die *Unified Modeling Language* (UML), die 1997 in der Version 1.0 auf den OMG-Aufruf eingereicht wurde. Eine Vielzahl von ehemaligen Konkurrenten beteiligte sich an der Erstellung von Version 1.1, die schließlich 1998 erschien. Hierbei bestand eines der Hauptziele in einer konsistenten Formulierung des Sprachkerns von UML, der in dem sogenannten *Metamodell* (siehe Kap. 9) festgehalten ist. Das Metamodell legt fest, welche Elemente die Sprache UML zur Verfügung stellt und wie diese korrekt zu verwenden sind. Zu diesem Zweck wurde zusätzlich die auf Prädikatenlogik basierende *Object Constraint Language* (OCL) [Obj10a] eingesetzt. In darauffolgenden Versionen wurden neben der Überarbeitung gewisser Sprachkonzepte Mechanismen zur Austauschbarkeit von Modellen in der Form des *XML Metadata Interchange Format* (XMI) [Obj07] und Mechanismen für die Generierung von Programmcode eingeführt. Neben diesen eher kleinen Änderungen wurde bereits im Jahr 2000 von der OMG ein Erneuerungsprozess von UML initiiert, der schließlich zur Verabschiedung des Sprachstandards UML 2.0 im Jahr 2005 führte. Bis auf kleine Änderungen, die über Zwischenversionen in der aktuellen Version 2.3 [Obj10d] resultierten und die in diesem Buch auch berücksichtigt werden, ist dies die Sprachbeschreibung von UML, wie wir sie in diesem Buch kennen- und verwenden lernen.

Metamodell

Object Constraint Language (OCL)

XML Metadata Interchange Format (XMI)

UML ist heutzutage eine der am weitesten verbreiteten grafischen objektorientierten Modellierungssprachen. Trotz der zahlreichen Überarbeitungen sind ihre Wurzeln (OMT, OOSE, Methode von Booch) immer noch deutlich erkennbar. UML eignet sich sowohl für die Modellierung von komplexen Objektbeziehungen als auch für die Modellierung von Abläufen mit Nebenläufigkeit.

UML ist eine universelle Modellierungssprache, d. h., ihre Verwendung ist nicht auf einen speziellen Anwendungsbereich beschränkt. UML stellt für die Modellierung beliebiger Anwendungsbereiche Sprach- und Modellierungskonzepte und eine intuitive grafische Notation zur Verfügung und ermöglicht so das

- Spezifizieren,
- Konstruieren,
- Visualisieren und
- Dokumentieren

eines Softwaresystems [RJB04]. Das Resultat einer Modellierung mit UML stellt ein grafisches Modell dar, das in Form von verschiedenen Diagrammen unterschiedliche Sichtweisen auf ein System bietet.

2.2 Verwendung

UML bindet sich weder an ein bestimmtes Entwicklungswerkzeug noch an eine bestimmte Programmiersprache oder an eine bestimmte Zielplattform, auf der das zu entwickelnde System eingesetzt werden soll. UML bietet auch kein Vorgehensmodell, wie Software entwickelt werden soll. UML entkoppelt vielmehr Modellierungssprache und Modellierungsmethode, die so projekt- bzw. unternehmensspezifisch festgelegt werden können. Die Sprachkonzepte von UML begünstigen allerdings eine iterative und inkrementelle Vorgehensweise [RJB04].

Einsatz im Softwareentwicklungsprozess

UML kann über den gesamten Softwareentwicklungsprozess hinweg in durchgängiger Weise eingesetzt werden. In allen Stufen der Entwicklung können die gleichen Sprachkonzepte in der gleichen Notation verwendet werden. So lässt sich ein Modell schrittweise verfeinern. Eine Übersetzung eines Modells in eine andere Modellierungssprache entfällt. Dadurch wird eine iterative und inkrementelle Vorgehensweise bei der Softwareentwicklung ermöglicht.

UML kann für verschiedenste Anwendungsbereiche eingesetzt werden, die unterschiedliche Anforderungen in Bezug auf verschiedenste Aspekte wie Komplexität, Datenvolumen, Echtzeit, Verteilung etc. stellen.

Generische Sprachkonzepte

Die Modellelemente von UML und ihre erlaubte Verwendung sind im *Metamodell* von UML spezifiziert [Obj10d]. Die Sprachkonzepte sind derart generisch definiert, dass eine breite und flexible Anwendbarkeit geschaffen wird. Um den Einsatz von UML