Erschienen in: Das Wirtschaftsstudium (wisu), 29. Jg, Heft 5, 2000, S. 709-718

Die Unified Modeling Language (UML) - ein bedeutsamer Standard für die konzeptionelle Modellierung

Prof. Dr. Ulrich Frank
Institut für Wirtschaftsinformatik
Universität Koblenz-Landau

Motivation

Für die Entwicklung betrieblicher Anwendungssysteme sind konzeptionelle Modelle von wesentlicher Bedeutung. Sie zielen darauf, die Lücke zwischen der Sichtweise der Anwender und der der Software-Entwickler zu schließen. Objektorientierte Modellierungssprachen sind dazu besonders gut geeignet. Seit einiger Zeit gibt es mit der Unified Modeling Language (UML) einen Standard, der für die konzeptionelle Modellierung eine Reihe vor allem wirtschaftlicher Vorteile verspricht. Da konzeptionelle Modelle nicht zuletzt darauf gerichtet, aktuelle und geplante Abbilder von Unternehmen in den Fachabteilungen zur Diskussion zu stellen, ist der Umgang mit solchen Modellen auch für Wirtschaftswissenschaftler von zunehmender Bedeutung. Der Beitrag gibt nach einer kurzen Einführung in die konzeptionelle Modellierung einen Überblick über die UML, der durch ein anschließendes Beispiel illustriert wird.

1 Bedeutung der konzeptionellen Modellierung

Die Entwicklung von Anwendungsprogrammen, etwa zur Unterstützung der Personalverwaltung, der Finanzbuchhaltung oder des elektronischen Geschäftsverkehrs erfordert eine sorgfältige Analyse der verwendeten Begriffe und der benötigten Funktionen und Prozesse. Eine große Herausforderung dabei ist das Erkennen aller relevanten Aspekte. Das gilt in zweifacher Hinsicht. In existierenden Geschäftsprozessen ist hier vor allem an Sonderfälle bzw. Ausnahmen zu denken. Sie werden aus naheliegenden Gründen bei der Erhebung der Anforderungen leicht vergessen. Andererseits ist zu berücksichtigen, dass sich die Anforderungen an Anwendungssysteme im Zeitverlauf ändern mögen: Geschäftsprozesse werden reorganisiert, neue Arten von Dokumenten sind zu integrieren, neuartige Funktionen sind abzubilden. Die Systemanalyse sollte solche zukünftigen Entwicklungen antizipieren, um das zu entwerfende System darauf frühzeitig vorzubereiten. Denn Modifikationen zu einem späteren Zeitpunkt sind i.d.R. mit hohen Kosten und erheblichen Risiken verbunden. Um zu leistungsfähigen Generalisierungen zu gelangen, die sowohl Sonderfälle abdecken als auch für zukünftige Entwicklungen offen sind, empfiehlt sich eine intensive Auseinandersetzung mit dem Analysegegenstand. Das wiederum erfordert eine geeignete Darstellung des Analysegegenstands. Eine solche Darstellung sollte sich auf die für den Analysezweck wesentlichen Aspekte beschränken und für alle Beteiligten hinreichend anschaulich sein. Die Analyse von Anwendungsdomänen empfiehlt also die Verwendung geeigneter Abstraktionen oder Modelle. <RN> Die Analyse und der Entwurf von Systemen erfordern Abstraktionen, um Komplexität zu reduzieren. </RN>

Die Systemanalyse ist kein Selbstzweck. Sie dient der Vorbereitung des Entwurfs von Software. Der Entwurf selbst zielt auf eine gehaltvolle Vorlage für die Implementierung. Er muss deshalb ein präzises Modell des zu realisierenden Systems liefern und gleichzeitig gewissen Randbedingungen der Implementierung Rechnung tragen – etwa den Sprachkonzepten, die von der zu verwendenden Programmier- oder Datenbanksprache zur Verfügung gestellt werden. Dessen ungeachtet entsteht ein Entwurfsmodell i.d.R. nicht allein im Kreis von Software-Entwicklern. Vielmehr sind auch Anwender und Domänenexperten zu beteiligen – es geht ja wesentlich um eine Präzisierung von Begriffen aus der Anwendungswelt. Programmiersprachen oder auch formale Spezifikationssprachen sind für

die Entwurfsphase kaum geeignet. Sie sind entweder auf abstrakte Maschinenmodelle oder ein hohes Maß mathematischer Präzision ausgerichtet und sperren sich deshalb gegen Darstellungen, die von allen Beteiligten als verständlich und authentisch empfunden werden. Auf der anderen Seite ist auch die Beschränkung auf eine natürliche Sprache, sei es in Form der Umgangssprache und/oder einer dedizierten Fachsprache, unzureichend: Schließlich sollen Beschreibungen von Anwendungsdomänen Vorgaben für die Implementierung liefern, so dass daraus erwachsende Besonderheiten auch berücksichtigt werden müssen. Gleichzeitig sollten die für den Entwurf verwendeten Konzepte einen engen Bezug zu den Konzepten der Analyse aufweisen, um unnötige Friktionen bei der Verwendung von Analyseergebnissen zu vermeiden.

Vor dem Hintergrund dieses Spannungsfelds ist die konzeptionelle Modellierung entstanden. Die konzeptionelle Modellierung dient einer allen Beteiligten an einem Software-Entwicklungsprozess verständlichen Darstellung einer Anwendungsdomäne. Die Darstellung beschränkt sich dabei auf die für die Erstellung des beabsichtigten Systems wesentlichen Aspekte dieser Domäne. Zusätzlich zu dieser Abstraktion muss ein konzeptionelles Modell auch den Randbedingungen Rechnung tragen, die durch die in einer späteren Phase zu verwendenden Implementierungssprachen entstehen. Dementsprechend sollten konzeptionelle Modelle – in unterschiedlicher Detaillierung und Formalisierung – sowohl für die Analyse als auch für den Entwurf eingesetzt werden. <RN> Konzeptionelle Modelle sollen so anschaulich sein, dass sie zweckgerichtete Diskussionen zwischen Projektbeteiligten mit unterschiedlichem beruflichem Hintergrund unterstützen. </RN>

Die Erstellung konzeptioneller Modelle erfordert geeignete Modellierungssprachen. Zu den nach wie vor bedeutendsten Sprachen der konzeptionellen Modellierung gehört das Entity Relationship Model (ERM) mit zahlreichen Derivaten. Auch Datenflussdiagramme (DFD) werden noch häufig verwendet. Seit einigen Jahren werden allerdings zunehmend objektorientierte Modellierungssprachen eingesetzt.

2 Objektorientierte Software-Entwicklung

In den achtziger Jahren wurden objektorientierte Technologien vor allem in Forschungsprojekten und von wenigen innovativen Unternehmen eingesetzt. Mittlerweile haben objektorientierte Programmiersprachen wie C++ und Java eine weite Verbreitung gefunden. Die Verwendung einer objektorientierten Programmiersprache allein ist allerdings nicht hinreichend dafür, die Potentiale objektorientierter Konzepte in sinnvoller Weise zu nutzen. Das erfordert ein angemessenes Verständnis dieser Konzepte sowie den Einsatz einer objektorientierten Modellierungssprache. <RN> Objektorientierte Software-Entwicklung erfordert neben geeigneten Programmiersprachen auch spezielle Sprachen für die konzeptionelle Modellierung. </RN>

2.1 Zentrale Konzepte und Vorteile gegenüber traditionellen Ansätzen

Traditionelle Anwendungssysteme bestehen aus Daten und darauf operierenden Funktionen. In der Kontrollstruktur eines Programms ist festgelegt, in welcher Reihenfolge bzw. bei welchen Ereignissen bestimmte Funktionen aufzurufen sind. Im Unterschied dazu besteht ein objektorientiertes System aus Objekten, die über Nachrichten kommunizieren. <RN> Objektorientierte Software-Systeme bestehen aus Objekten, die Nachrichten austauschen. </RN> Ein Objekt ist ein Artefakt, das einen Gegenstand oder ein Konzept aus der Anwendungsdomäne repräsentiert, also z.B. einen Kunden oder ein Verrechnungskonto. <RN> Ein Objekt ist ein software-technisches Konzept, das häufig mit einem realweltlichen Objekt oder Konzept korrespondiert. </RN> Ein Objekt kann von seiner Umwelt über seine Schnittstelle genutzt werden. Die Schnittstelle besteht aus einer Menge von Diensten, die von

den Operationen des Objekts bereitgestellt werden. Auf diese Weise können Eigenschaften eines Objekts (z.B. der Name eines Kunden) abgefragt werden oder Funktionen, die in den Operationen implementiert sind, genutzt werden (z.B. die Berechnung des Alters eines Kunden aus dem in einem Objekt gespeicherten Geburtsdatum). Die besondere Eignung von Objekten für die Software-Entwicklung ist darin zu sehen, dass sie ein vielen Menschen auch außerhalb der Informatik vertrautes Konzept darstellen und zudem ein hohes Abstraktionsniveau unterstützen. Dadurch wird einerseits eine Darstellung von Systemen unterstützt, die nicht nur für Software-Entwickler anschaulich ist. Andererseits wird die Flexibilität von Anwendungen gegenüber zukünftigen Anforderungsänderungen gefördert bzw. die Wiederverwendbarkeit existierender Programmteile verbessert. Dies wird durch die folgenden Konzepte sichergestellt. <RN> Der wesentliche Vorteil objektorientierter Software-Entwicklung liegt darin, dass sie ein besonders hohes Abstraktionsniveau erlaubt. </RN> Klassen erlauben die Zusammenfassung gleichartiger Objekte. Dadurch wird es möglich, von den spezifischen Eigenschaften einzelner Objekte (auch Instanzen genannt) zu abstrahieren. So legt man z.B. die Eigenschaften der Klasse „Kunde“ mit Hilfe geeigneter Attribute (z.B. „Vorname“, „Nachname“, „Einkommensklasse“) und Operationen (z.B.: „alterInJahren“) fest. Wenn sich nun im Zeitverlauf neue Eigenschaften als sinnvoll erweisen (z.B. das Attribut „Email-Adresse“), wird lediglich die Klassenbeschreibung geändert. Damit ist die Erweiterung implizit auch für alle Objekte dieser Klasse wirksam (wobei wir hier von den Besonderheiten einzelner Programmiersprachen und korrespondierender Werkzeuge abstrahieren).

Verkapselung bedeutet, dass auf die von einem Objekt verwalteten Daten (die auch als Objekte vorliegen können), von anderen Objekten nicht direkt zugegriffen werden kann. Vielmehr kann nur indirekt über die Schnittstelle eines Objekts zugegriffen werden. In der Außensicht kann man also von der inneren Struktur eines Objekts abstrahieren („information hiding“). Wenn sich im Zeitverlauf diese Struktur ändert, zieht dies nicht zwangsläufig eine Änderung der Schnittstelle nach sich, was die Wartbarkeit eines Systems erheblich erleichtert. <RN> Verkapselung gestattet die Abstraktion von der internen, im Zeitverlauf u.U. zu ändernden Struktur eines Objekts. </RN>

Durch Generalisierung wird es möglich, gemeinsame Eigenschaften einer Menge von Klassen in einer Oberklasse zusammenzufassen. Auf diese Weise können Klassenhierarchien gebildet werden. Durch die Bildung von Unterklassen, auch Spezialisierung genannt, können vorhandene Klassen an neue Anforderungen angepasst werden: Eine Unterklasse erbt alle Eigenschaften ihrer Oberklasse und kann mit zusätzlichen Eigenschaften versehen werden, weshalb man in diesem Zusammenhang auch von Vererbung spricht. Änderungen eines Systems setzen immer auf dem höchsten sinnvollen Generalisierungsniveau an. Von den jeweiligen Unterklassen kann also abstrahiert werden. Generalisierung kann zu Oberklassen führen, denen in der Anwendungsdomäne keine konkreten Klassen entsprechen. Beispielsweise könnte eine Generalisierung über die Klassen „Rechnung“, „Gutschrift“ und „Lieferschein“ zur Oberklasse „Auftragsabwicklungsdokument“ führen. <RN> Generalisierung unterstützt eine effiziente Systempflege. </RN> Um sicherzustellen, dass in dem zu erstellenden System keine unsinnigen Instanzen gebildet werden, können solche Klassen als abstrakte Klassen gekennzeichnet werden. Eine abstrakte Klasse dient lediglich der Nutzung von Generalisierungsmöglichkeiten. Sie hat keine Instanzen.

Neben den bereits genannten Vorteilen verspricht die objektorientierte Software-Entwicklung eine besonders enge Integration der verschiedenen Phasen der Software-Entwicklung: Von der Analyse bis zur Implementierung stellen Objekte das zentrale Konzept dar – wenn auch in unterschiedlicher Detaillierung.

2.2 Vorläufer der UML

Eine objektorientierte Modellierungssprache wird durch eine abstrakte Syntax, eine Semantik und eine konkrete Syntax beschrieben. Die abstrakte Syntax legt fest, wie die Konzepte (wie „Klasse“, „Objekt“, „Attribut“, etc.) einer Sprache verknüpft werden dürfen. Die Semantik legt fest, wie die Konzepte zu interpretieren sind, also z.B. die Bedeutung von Vererbung. Die konkrete Syntax schließlich beschreibt die grafische und ggfs. textuelle Notation. Abstrakte Syntax und Semantik werden häufig mit Hilfe sog. Metamodelle dargestellt (s. 3.2). Objektorientierte Modellierungssprachen sind seit längerem Gegenstand intensiver Forschungsbemühungen. Entsprechend groß ist die Zahl einschlägiger Ansätze. <RN> In den frühen neunziger Jahren gab es eine Fülle objektorientierter Modellierungssprachen. </RN> Bis Mitte der neunziger Jahre hatten drei Ansätze besondere Popularität erlangt. Die „Object Modeling Technique“ (OMT, Rumbaugh et al. 1991), in einer Arbeitsgruppe von General Electric entwickelt, ist deutlich an traditionellen Sprachen wie dem ERM und Datenflussdiagrammen orientiert. Die Methode von Booch (Booch 1994) ist im Unterschied dazu durch eine ausgeprägtere Berücksichtigung objektorientierter Konzepte gekennzeichnet. Das in Schweden bei Ericsson entstandene „Object-Oriented Software Engineering“ (OOSE, Jacobson 1993) bietet mit sog. Anwendungsszenarien („use cases“) ein Konzept, das eine durchgängige Beschreibung eines Systems aus der Sicht der Nutzer unterstützt. Während die drei Methoden spezifische Stärken und Schwächen aufweisen, sind sie durch einen gemeinsamen Mangel eingeschränkt: Es fehlt ihnen eine präzise Sprachbeschreibung. Die Sprachkonzepte werden statt dessen durch unvollständige und mehrdeutige Darstellungen sowie ergänzende Beispiele illustriert.

2.3 Gründe für die Standardisierung von Modellierungssprachen

Aus der Sicht von Unternehmen, die Software entwickeln, war die damalige Situation unbefriedigend. Auf der einen Seite hatte sich die Erkenntnis durchgesetzt, dass objektorientierte Ansätze eine wirtschaftlichere Entwicklung und Wartung von Software erlauben, auf der anderen Seite gab es keine einheitliche Modellierungssprache. Vor allem das Bemühen um Investitionsschutz spricht für eine Standardisierung von Modellierungssprachen. Im einzelnen ist an folgende Aspekte zu denken:

- Der Einsatz einer Modellierungssprache erfordert i.d.R. Modellierungswerkzeuge, die diese Sprache unterstützen. Bei solchen Werkzeugen handelt es sich um grafische Editoren, die die Erstellung, Verwaltung und Pflege von Modellen unterstützen und zudem die Generierung von Code erlauben. Die Einführung solcher Werkzeuge ist mit hohen Kosten verbunden. Das gilt vordergründig für den Kauf und die Wartung, vor allem aber für die Schulung der Mitarbeiter. Wenn die gewählte Modellierungssprache im Zeitverlauf durch andere verdrängt wird, sind diese Investitionen gefährdet: Der Anbieter des entsprechenden Modellierungswerkzeugs sieht keine hinreichenden Absatzchancen mehr und stellt das Produkt deshalb ein. Ein Standard hingegen verspricht größere Unabhängigkeit von einem Anbieter. <RN> Standardisierung verspricht vor allem einen besseren Investitionsschutz. </RN>
- Die Erstellung konzeptioneller Modelle erfordert einen hohen Aufwand. Ein Standard hilft nicht nur, die in die Modelle getätigten Investitionen zu schützen, sondern vereinfacht auch die Wiederverwendung von Modellen in anderen Kontexten. Auf diese Weise wird auch ein Marktplatz für konzeptionelle Modelle möglich. In letzter Zeit hat diese Vorstellung unter dem Etikett „Referenzmodell“ erhebliche Resonanz gefunden.
- Wenn eine standardisierte Modellierungssprache verwendet wird, vergrößert sich die Chance, am Arbeitsmarkt einschlägig qualifizierte Mitarbeiter zu finden. In Zeiten eines ohnehin gravierenden Mangels an DV-Fachkräften ist dies ein gewichtiges Argument.

3 Unified Modeling Language

Die Entstehung der UML geht auf zwei parallele Entwicklungen zurück. Zunächst einigten sich Rumbaugh und Booch darauf, ihre rudimentären Sprachentwürfe zu einer neuen, gemeinsamen Modellierungssprache weiter zu entwickeln und zusammen mir zugehörigen Werkzeugen zu vermarkten. Die neue Sprache wurde Unified Modeling Language genannt. Kurze Zeit später schloss sich auch Jacobson der Firma Rational an. Im Juni 1996 veröffentlichte die Object Management Group (OMG), ein Konsortium aus Anbietern und Anwendung von Informationstechnologie, einen "Request for Proposals", in dem Vorschläge für standardisierungsfähige Sprachen eingefordert wurden. <RN> Die UML entstand durch das gemeinsame Bemühen dreier Fachautoren und eine Initiative der OMG. </RN> Bis Januar 1997 sind sechs Vorschläge bei der OMG eingegangen. Kurze Zeit später einigten sich die Einsender unter Mitwirkung der OMG darauf, gemeinsam einen Vorschlag weiter zu verfolgen, der wesentlich auf der UML der amerikanischen Firma Rational basierte. Nach einer Reihe von Überarbeitungen wurde die UML 1998 von der OMG als Standard verabschiedet. Mittlerweile liegt die Version 1.3 vor. <RN> Gegenwärtig liegt die UML in der Version 1.3. vor. </RN> Es ist davon auszugehen, dass die UML in der kommerziellen Software-Entwicklung heute die wichtigste objektorientierte Modellierungssprache ist.

3.1 Diagrammarten und Erweiterungsmöglichkeiten

Konzeptionelle Modelle sind Abstraktionen. Im Mittelpunkt der objektorientierten Modellierung stehen sog. Klassendiagramme. Sie dienen der Beschreibung statischer Systemaspekte. Ein Klassendiagramm besteht aus Klassen und Beziehungen zwischen Klassen. Eine Klasse wird als Rechteck dargestellt. Neben dem Klassennamen kann das Rechteck auch die Namen von Attributen und Operationen enthalten. <RN> Klassendiagramme stellen die statische Struktur eines Systems dar. </RN> Generalisierungsbeziehungen werden als Pfeil dargestellt, wobei die Pfeilspitze auf die Oberklasse zeigt. In Abb. 2 ist „Konto“ auf diese Weise als Oberklasse von „Girokonto“ und „Sparkonto“ gekennzeichnet. Zudem ist „Konto“ als abstrakte Klasse ausgewiesen. Assoziationen zwischen Klassen werden durch Linien zwischen jeweils zwei Klassensymbolen repräsentiert. An die Linien werden Kardinalitäten angetragen, die die minimale und maximale Anzahl von an der Assoziation beteiligten Objekte einer Klasse beschreiben. So drückt etwa die Kardinalität 0..* in der Assoziation zwischen Bank und Kontoinhaber aus, dass eine Bank minimal kein Konto führt, maximal beliebig viele. Außerdem kann eine Assoziation mit einem Namen gekennzeichnet werden, für den mittels eines Pfeils eine Leserichtung angegeben wird. Attribute werden durch einen Namen und eine Klasse spezifiziert. Die Klasse eines Attributs beschreibt seine Bedeutung. So wird im Beispiel das Attribut „Geburtsdatum“ durch die Klasse „Date“ charakterisiert. Operationen können durch die jeweils benötigen Parameter näher beschrieben werden.

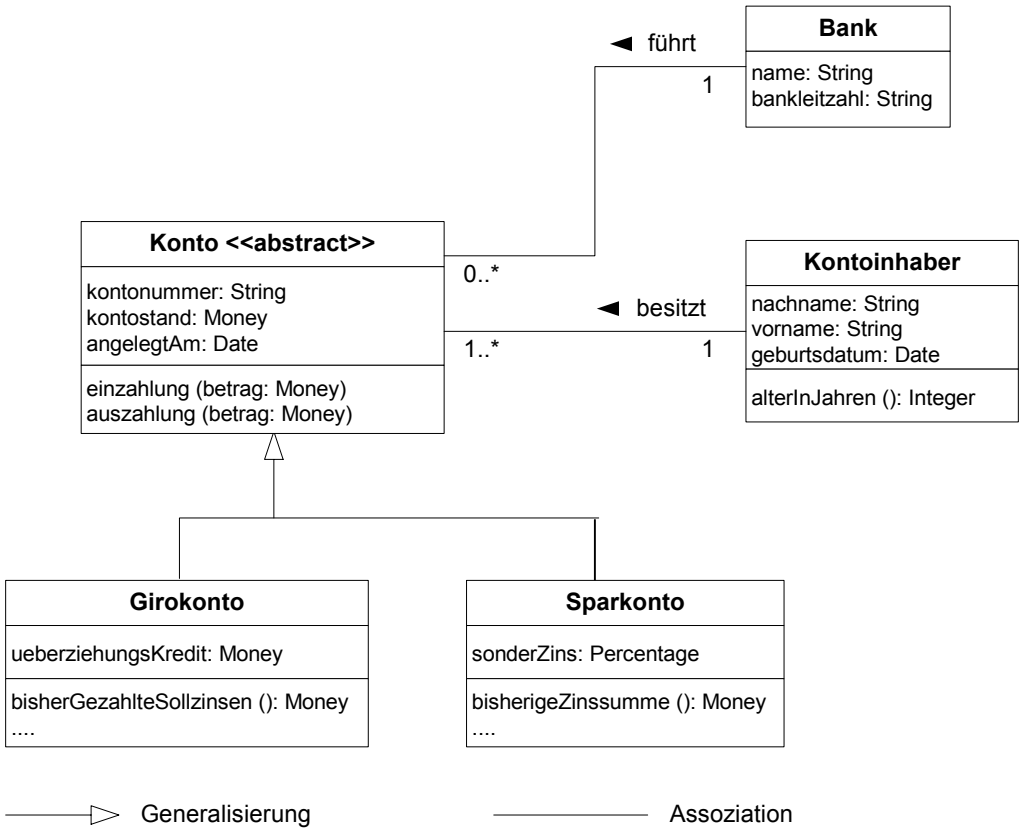

Abb. 1: Klassendiagramm in UML

In Ergänzung zu Klassendiagrammen können in Abhängigkeit vom jeweils vorherrschenden Blickwinkel weitere Systemaspekte von Bedeutung sein, z.B. der Nachrichtenaustausch zwischen Objekten oder zulässige Zustandsänderungen von Objekten. <RN> Eine Diagrammart in UML unterstützt eine bestimmte Sicht auf ein System. </RN> Die UML bietet denn auch eine Reihe unterschiedlicher Diagrammarten, die jeweils für eine bestimmte Abstraktion gedacht sind. Insgesamt erlaubt die UML z.Z. acht verschiedene Diagrammarten, die in der folgenden Übersicht dargestellt sind. <RN> Die UML bietet acht verschiedene Diagrammarten. </RN>

Abstraktion	Diagrammart	Zentrale Konzepte
Statische Sicht	Class Diagram	Class, Association, Generalisation, Attribute, …
Systemnutzung	Use Case Diagram	Use Case, Actor, Association
Implementierungssicht	Component Diagram	Component, Interface, Dependency
Einsatzsicht	Deployment Diagram	Node, Component, Dependency
Dynamische Sicht	State Chart Diagram	State, Event, Transition, Action
Prozesssicht	Activity Diagram	State, Activity, Completion Transition, Fork, Join
Interaktionssicht	Sequence Diagram	Interaction, Message, Activation
	Collaboration Diagram	Collaboration, Interaction, Message

Jede Diagrammart ist mit einer bestimmten Sicht bzw. Abstraktion verknüpft. An diese Stelle ist eine ausführliche Beschreibung aller Diagrammarten (eine solche findet sich in Rumbaugh et al. 1999) nicht möglich. Statt dessen skizzieren wir die mit den Diagrammarten verbundenen Abstraktionen durch die Fragen, mit denen sich der Modellierer bzw. Betrachter jeweils dem Modellierungsgegenstand nähert: <RN> Die Auswahl einer Diagrammart richtet sich nach dem Zweck der Analyse bzw. des Entwurfs. </RN>

- Welche Klassen sind für die Abbildung des Gegenstandsbereichs geeignet? Welche Attribute und Operationen weisen die Klassen auf? Welche Beziehungen gibt es zwischen den Klassen bzw. Objekten? (Class Diagram)
- Welche Fälle der Systemnutzung gibt es? Wie stellen sich in jedem Fall die Anforderungen an die Systemfunktionalität aus der Sicht des Akteurs dar? Welche Dienste erwartet der Akteur von den verfügbaren Objekten? (Use Case Diagram)
- Welche Komponenten (vorhandene Anwendungen, implementierte Klassen und Klassengruppierungen) sind für die Systemarchitektur wesentlich? Welche Beziehungen/Abhängigkeiten gibt es zwischen den Komponenten? Welche Schnittstellen bieten die Komponenten und das System? (Component Diagram)
- Welche Systemressourcen (Computer, Massenspeicher, periphere Geräte) werden von den Komponenten benötigt? (Deployment Diagram)
- Welche wesentlichen Ereignisse sind für Objekte einer Klasse zu beachten? Wie sollen die Objekte dieser Klasse auf die Ereignisse reagieren? In welcher Weise ändert sich beim Auftreten eines Ereignisses ggfs. der Zustand eines Objekts? (State Chart Diagram)
- Welche Prozesse sind für das zu erstellende System von Bedeutung? Wie lassen sich diese Prozesse als zeitliche Folge von Aktivitäten beschreiben? (Activity Diagram)
- Welche Nachrichten schicken sich Objekte verschiedener Klassen zu? In welcher zeitlichen Reihenfolge erfolgt der Nachrichtenaustausch? (Sequence Diagram)
- Wie können Objekte, die untereinander Nachrichten austauschen (die also „zusammenarbeiten“), sich gegenseitig identifizieren? (Collaboration Diagram)

Für die Systementwicklung ist es von zentraler Bedeutung, dass die in verschiedenen Diagrammen dargestellten Sichten auf ein System in konsistenter Weise integriert werden. Integration erfordert gemeinsame Konzepte. In Kapitel 4 werden ausgewählte Diagrammarten sowie deren Integration anhand eines Beispiels näher erläutert.

Auch wenn die UML eine Fülle von Diagrammarten bietet, mag es Modellierungsaufgaben geben, für die keine zufriedenstellenden Konzepte verfügbar ist. Für solche Fälle erlaubt die UML individuelle Erweiterungen. Ein wichtiges Instrument für solche Erweiterungen stellen sog. „Stereotypes“ dar. Ein Stereotype wird eingeführt, indem man ein existierendes Sprachkonzept mit einer speziellen Interpretation (also einer individuellen Semantik) versieht. Zudem kann man dieses Konzept durch ein selbst definiertes grafisches Symbol kennzeichnen. Auf diese Weise könnte man etwa für die Organisationsmodellierung das Konzept „Organisationseinheit“ einführen, indem man das vorhandene Konzept „Klasse“ um eine entsprechende Semantik ergänzt. Dazu würde etwa gehören, dass eine Organisationseinheit anderen Organisationseinheiten übergeordnet sein kann, aber nicht sich selbst. Der zusätzlichen Flexibilität, die die UML durch Stereotypes gewinnt, steht ein gravierender Nachteil gegenüber: Die für Stereotypes festgelegte Semantik ist nicht Bestandteil des Standards, so dass deren Nutzung die Wiederverwendung von Modellen erheblich einschränkt.

3.2 Sprachspezifikation

Die UML ist eine komplexe Sprache. Dementsprechend umfangreich ist die Sprachbeschreibung. <RN> Die Beschreibung einer Sprache erfordert eine Metasprache. </RN> Sie besteht aus einer Sprachspezifikation und einem ergänzenden Handbuch, das die Anwendung der Sprachkonzepte beschreibt. Die Sprachspezifikation besteht aus einem Metamodell und ergänzenden Integritätsbedingungen. Ein Metamodell ist gleichsam das Modell einer Sprache. <RN> Die aktuelle Spezifikation der UML findet sich unter www.rational.com bzw. unter www.omg.org. </RN> Abb. 2 illustriert die verschiedenen Sprachebenen, die bei der Modellierung eines Realitätsausschnitts zu beachten sind.

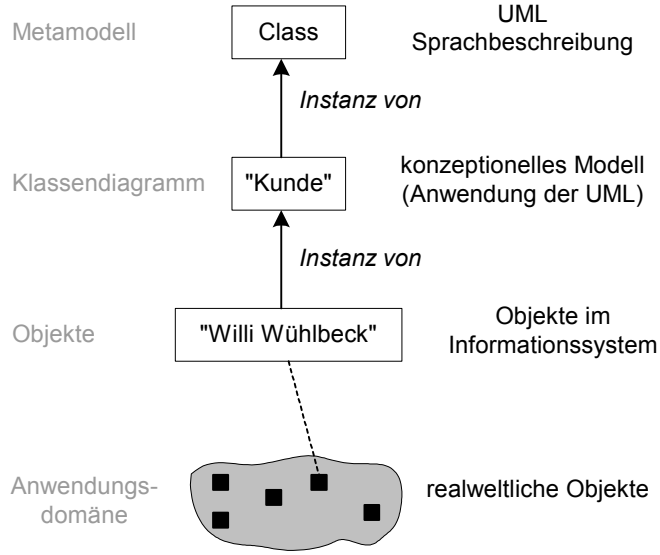

Abb. 2: Ebenen der Modellierung

Im Fall der UML wird das Metamodell mit Hilfe eines vereinfachten Klassendiagramms dargestellt. <RN> Ein konkretes UML-Diagramm ist eine Instanz des UML-Metamodells. </RN> Grundsätzlich könnte die Sprachspezifikation auch mittels einer Grammatik erfolgen. Die Verwendung von Metamodellen hat allerdings zwei Vorteile: Durch die Verwendung gleicher Darstellungskonzepte für die Sprachbeschreibung und die Sprachanwendung wird ein Paradigmenbruch vermieden, der den Sprachanwendern einen leichteren Zugang zur Sprachspezifikation eröffnen sollte. Die Anwendung einer Modellierungssprache empfiehlt geeignete Modellierungswerkzeuge. Für die Realisierung solcher Werkzeuge ist ein objektorientierter Ansatz besonders gut geeignet. Metamodelle bieten deshalb eine gute Grundlage für den Entwurf von Werkzeugen, da sie leicht in Klassendiagramme überführt werden können. Abb. 3 zeigt einen Ausschnitt aus dem Metamodell der UML. Er stellt die Sprachkonzepte dar, die in dem Klassendiagramm in Abb. 1 angewendet werden. Eine Klasse (eine Instanz von „Class“) kann beliebig viele Eigenschaften („Feature“) haben, bei denen es sich jeweils konkret entweder um Attribute oder Operationen handelt. Etwas schwieriger sind die Konzepte nachzuvollziehen, die für die Darstellung von Assoziationen verwendet werden. Die Assoziation zwischen „Konto“ und „Kontoinhaber“ in Abb. 1 wird wie folgt konstruiert: Die Klasse „Konto“ ist eine Instanz von „Class“, die mit einer Instanz von „AssociationEnd“ verknüpft ist. Dieser Instanz ist die Kardinalität („multiplicity“) 1..* zugeordnet. Außerdem ist sie mit einer Instanz von „Association“ verknüpft, die wiederum mit einer zweiten Instanz von „AssociationEnd“ (der die Kardinalität 1 zugeordnet ist) verbunden ist. Diese Instanz von „AssociationEnd“ wiederum ist mit einer Instanz von „Class“, nämlich der Klasse „Kontoinhaber“ verknüpft ist.

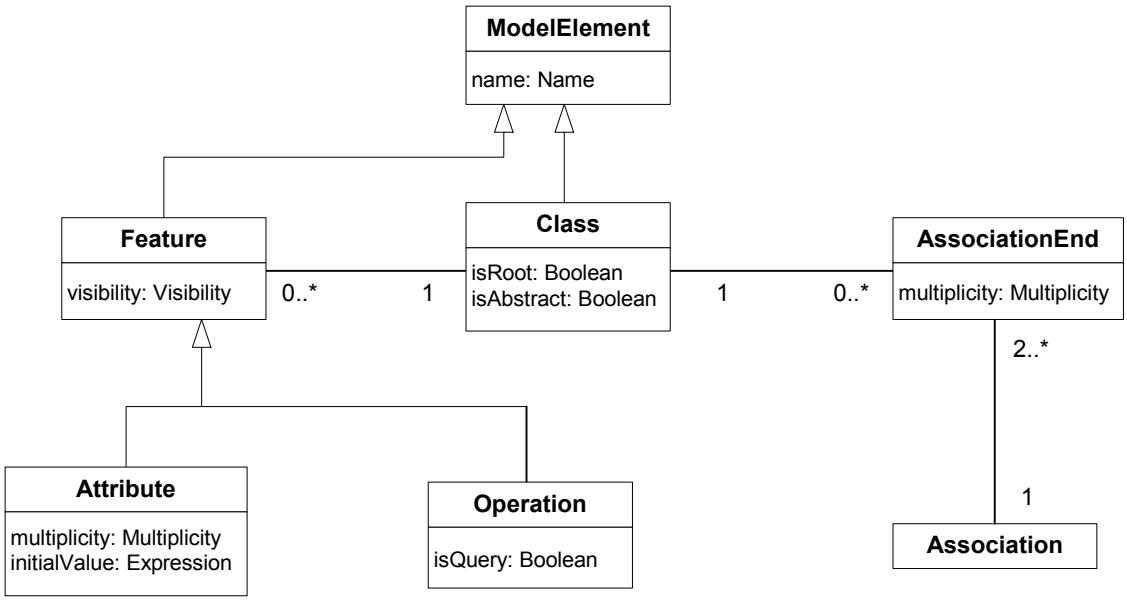

Abb. 3: Vereinfachter Ausschnitt des Metamodells der UML

Aus Sicht der Informatik ist es wünschenswert, dass ein Metamodell ein korrektes Modell der jeweiligen Sprache liefert. Die Sprachbeschreibung sollte die Identifikation unzulässiger Modelle gewährleisten und gleichzeitig erlauben, die Menge aller zulässigen Modelle zu generieren. Die Sprachbeschreibung sollte zudem vollständig sein, d.h. alle in der Sprache verwendeten Konzepte sowie die Bedingungen ihrer Verwendung sollten definiert sein. Diese Forderungen erfüllt die UML nur eingeschränkt, da sie eine Reihe semantischer Lücken enthält. <RN> Trotz ihres Umfangs enthält die Sprachspezifikation der UML eine Reihe von Mehrdeutigkeiten. </RN> Einige Konzepte, u.a. auch das wichtige Konzept „Vererbung“, sind nicht präzise definiert. Ihre Verwendung erfordert also eine entsprechende Interpretation des Benutzers. Beim Austausch bzw. der Wiederverwendung von Modellen sind deshalb Konventionen festzulegen, um die in UML enthaltenen Mehrdeutigkeiten zu eliminieren. Ergänzend zu den offiziellen Dokumenten der OMG gibt es eine Reihe von Lehrbüchern, die in den Umgang mit der UML einführen (Fowler/Scott 1997; Oestereich 1998; Page-Jones 1999). Eine kritische Bewertung der UML findet sich in Frank/Prasse 1997.

4 Ein Beispiel

Um die Beziehungen zwischen verschiedenen Diagrammen der UML zu verdeutlichen, betrachten wir im folgenden ein einfaches Beispiel. In einem Unternehmen soll eine neue Software für die Auftragsbearbeitung erstellt werden. Zunächst wird untersucht, welche Anforderungen die zukünftigen Nutzer des Systems haben. Dazu wird für jeden Benutzer ermittelt, welche Anwendungsszenarien, also Anlässe zur Systemnutzung, es jeweils gibt. Jedes Anwendungsszenario ist dadurch gekennzeichnet, dass ein Benutzer vom System bestimmte Informationen nachfragt und/oder die Ausführung bestimmter Funktionen erwartet. Das Ergebnis einer solchen Erhebung kann mit Hilfe von Use Cases ermittelt werden. <RN> Ein Anwendungsszenario („use case“) stellt typische Nutzungsformen eines Systems dar. </RN> Abb. 4 veranschaulicht die grafische Darstellung des Anwendungsfalls „Auftragsbearbeitung“, in den weitere Anwendungsfälle eingebettet sind. Da die Grafik allein wenig aussagt, ist die Dokumentation unbedingt durch ausführliche natürlichsprachliche Beschreibungen der Anwendungsfälle zu ergänzen.

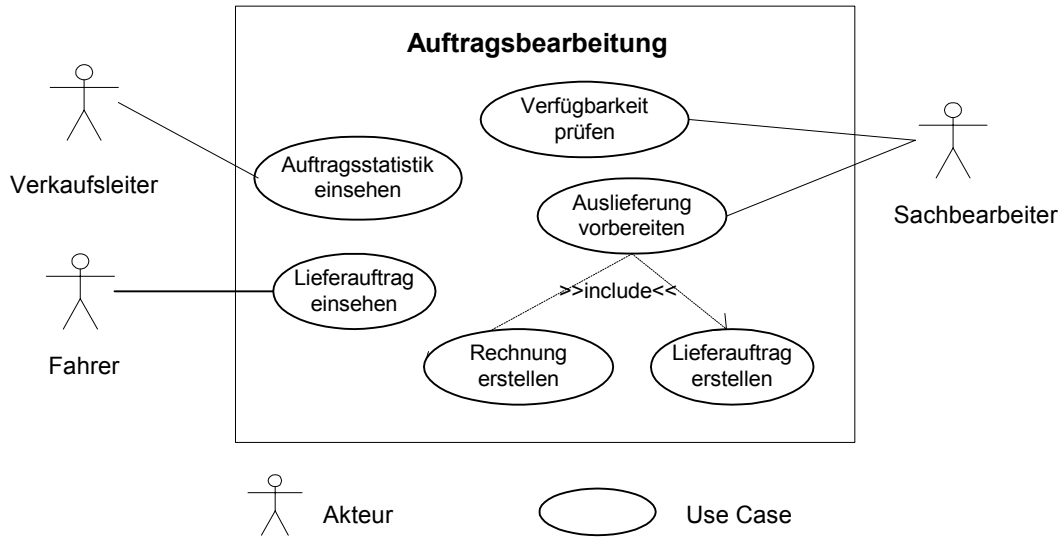

Abb. 4: Die Auftragsbearbeitung als Anwendungsfall (Use Case) dargestellt

Der objektorientierte Entwurf der Software macht es erforderlich, geeignete Objekte bzw. Klassen zu identifizieren. Es hat sich allerdings als wenig erfolgversprechend erwiesen, unmittelbar nach den Objekten in der Anwendungsdomäne zu fragen. Statt dessen ist es sinnvoller, zunächst die Aufgaben bzw. Prozesse zu betrachten: Die meisten Menschen sind eher in der Lage, ihr Arbeitsumfeld mit Hilfe von Prozessen zu beschreiben. <RN> Bei der Systemanalyse sind i.d.R. zunächst Funktionen bzw. Prozesse zu ermitteln. </RN> Abb. 5 zeigt ein Acitivity Diagram, in dem einzelne Aufgaben (Aktivitäten) zeitlich geordnet sind (von oben nach unten). Zudem können alternative Abläufe durch das Einfügen von Bedingungen (durch eine Raute veranschaulicht) dargestellt werden. Für jede Aktivität kann nun ermittelt werden, welche Objekte (bzw. deren Klassen) jeweils benötigt werden. Um die Klassen zu spezifizieren, ist darüber hinaus nach den erforderlichen Operationen und Attributen zu fragen. <RN> Durch eine nähere Betrachtung des Informationsbedarfs von Prozessen ergeben sich Rückschlüsse auf die benötigten Objekte. </RN> Wie die für die Implementierung nötige Integration von Aktivitäts- und Klassendiagrammen durchzuführen ist, wird durch die UML nicht genau spezifiziert. Aktivitäten können als Operationen interpretiert werden, so dass beide Diagrammarten über diese gemeinsamen Bestandteile verknüpft werden könnten. Im vorliegenden Beispiel werden Aktivitäten als Aufgaben im Rahmen eines Geschäftsprozesses interpretiert. <RN> Die Integration verschiedener Diagramme erfolgt über gemeinsame Konzepte. </RN> Die Ausführung dieser Aufgaben erfordert u.U. den Rückgriff auf Objekte in einem Informationssystem. Die Verwendung von Objekten/Klassen in einzelnen Aktivitäten ist in Abb. 5 ist durch gestrichelte graue Linien visualisiert.

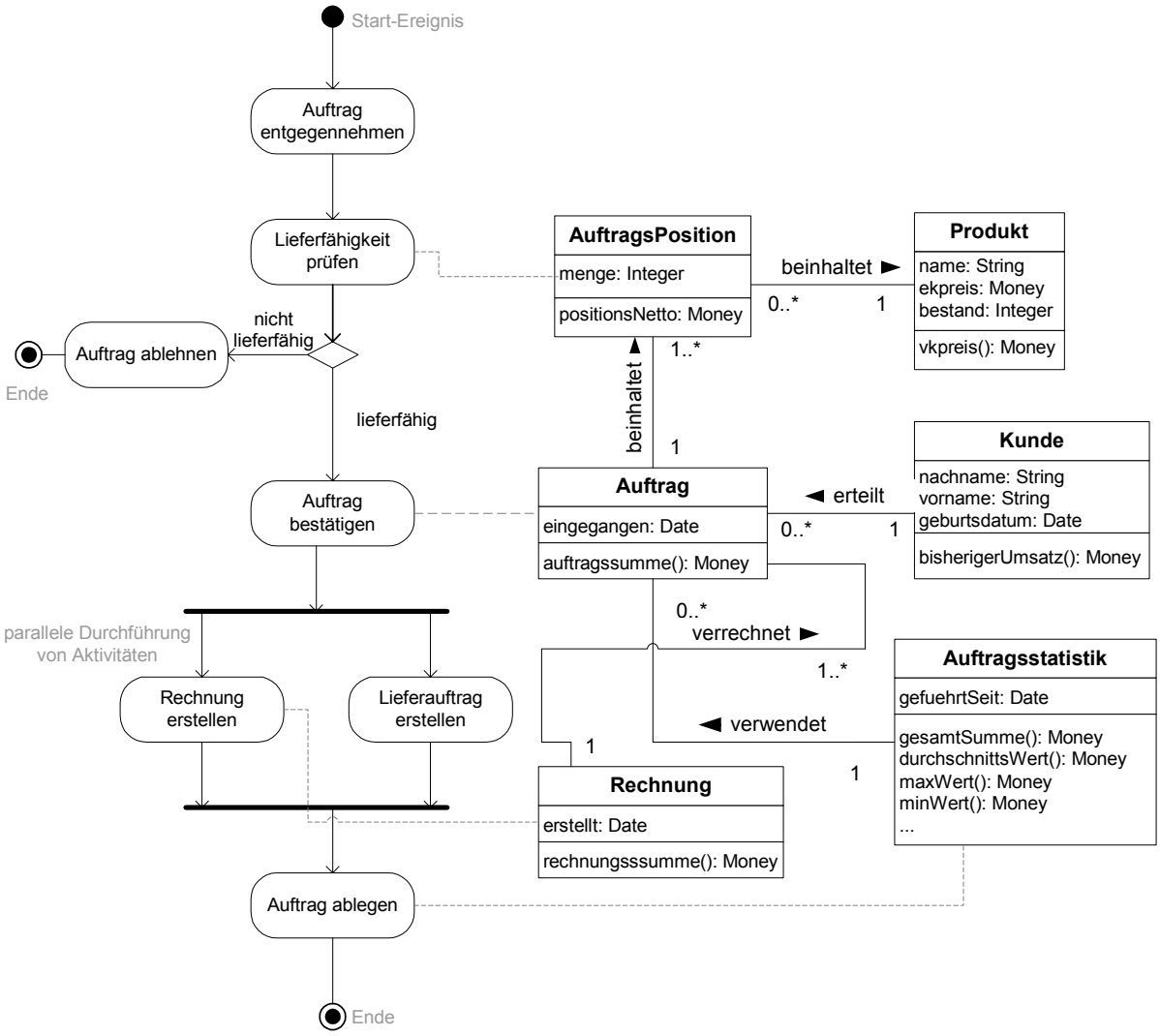

Abb. 5: Aktivitätsdiagramm und zugehöriges Klassendiagramm

Die Beispiele basieren auf stark vereinfachenden Annahmen. So sind Kunden häufig nicht nur Personen, sondern auch Organisationen. Ein Auftrag kann auf mehrere Rechnungen verteilt werden etc. Gerade wegen seiner Unzulänglichkeiten sollte das Beispiel verdeutlichen, wie wichtig es ist, die in einem Modell implizit enthaltenen Annahmen sorgfältig darauf zu prüfen, ob sie allen Einzelfällen in dem betrachteten Realitätsausschnitt gerecht werden.

5 Abschließende Bemerkungen

Konzeptionelle Modelle stellen ein Medium für die Kommunikation zwischen Software-Entwicklern, Auftraggebern und Anwendern dar. Die UML ist deshalb nicht nur eine Sprache für Informatiker oder Wirtschaftsinformatiker, sondern auch für Wirtschaftswissenschaftler. <RN> Die UML ist auch für Wirtschaftswissenschaftler von erheblicher Bedeutung. </RN> Dabei ist vor allem an solche Diagrammarten zu denken, die einen offenkundigen Bezug zur Anwendungsdomäne haben und weniger mit software-technischen Details behaftet sind. Dazu gehören in erster Linie die in dem dargestellten Beispiel verwendeten Diagrammarten. Auch wenn die Standardisierung einer Sprache für die konzeptionelle Modellierung erhebliche Vorteile verspricht (s. 3.2), reicht ein Sprachstandard allein nicht aus, um Software-Entwicklungsprojekte erfolgreich durchzuführen. Vielmehr ist es erforderlich, entsprechende Projekte sorgfältig zu planen und zu überwachen. Die OMG hat bewusst darauf verzichtet, einen Prozess für die Anwendung der UML zu standardisieren, da die Vielfalt denkbarer

Software-Entwicklungsprojekte allenfalls sehr abstrakte und deshalb im Einzelfall wenig hilfreiche Vorgaben ermöglichen würde. Es gibt allerdings mittlerweile eine Reihe von Lehrbüchern, die darauf zielen, diese Lücke zu füllen (vgl. Kruchten 1999; Jacobson et al. 1999). Sie geben Empfehlungen für die Vorgehensweise, Heuristiken und Beispiele. <RN> Die Durchführung einschlägiger Projekte erfordert eine sorgfältige Planung des Ablaufs. </RN>

Aus der Sicht der Wirtschaftsinformatik ist die UML trotz der Fülle der angebotenen Diagrammarten nicht vollständig. Insbesondere fehlen Modellierungssprachen, die eine detaillierte Darstellung von Geschäftsprozessen oder strategischer Zusammenhänge (z.B. in Form von Wertketten) ermöglichen. <RN> Die UML beinhaltet keine speziellen betriebswirtschaftlichen Konzepten. Sie ist deshalb für die Unternehmensmodellierung nur begrenzt geeignet. </RN> Neben eigenständigen Ansätzen zur Unternehmensmodellierung aus der Wirtschaftsinformatik, gibt es seit einiger Zeit einschlägige Forschungsansätze, die auf eine entsprechende Erweiterung der UML um sog. „Profiles“ – spezielle Modellierungssprachen auf der Basis der UML – gerichtet sind. Es bleibt abzuwarten, ob solche Erweiterungen eines Tages in die UML mit aufgenommen werden.

Literatur

Booch, G.: Object-oriented Analysis and Design with applications. Redwood City, Ca. 1994

Fowler, M.; Scott, K.: UML Destilled: Applying the Standard Object Modeling Language. Reading, Mass., et al. 1997

Jacobson, I.: Object-Oriented Software Engineering. Reading, Mass. 1993

Jacobson, I.; Booch, G.; Rumbaugh, J.: Unified Software Development Process. Reading, Mass., et al. 1999

Frank, U.; Prasse, M.: Ein Bezugsrahmen zur Beurteilung objektorientierter Modellierungssprachen - veranschaulicht am Beispiel von OML und UML. Arbeitsberichte des Institut für Wirtschaftsinformatik der Universität Koblenz-Landau, Nr. 7, 1997

Kruchten, P.: Der Rational Unified Process: Eine Einführung. München 1999

Oestereich, B.: Objektorientierte Softwareentwicklung. München, Wien 1998

Page-Jones, M.: Fundamentals of Object-Oriented Design in UML. Reading, Mass., et al. 1999

OMG (Hg.): OMG Unified Modeling Language Specification. Version 1.3 1999, (digitales Dokument, verfügbar unter www.rational.com)

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, B.: Object-oriented Modeling and Design. Englewood Cliffs 1991

Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual. Reading, Mass., et al. 1999