Objektorientierte Analyse und Design

1

Le temps est un grand professeur, mais malheureusement il tue tous ses élèves.
(Die Zeit ist ein großer Lehrer, aber leider bringt sie alle ihre Schüler um.)

– Hector Berlioz

Kontext

In diesem Kapitel werden die Ziele dieses Buches und OOA/D beschrieben. Im nächsten wird die iterative und evolutionäre Entwicklung vorgestellt, die die Form vorgibt, in der OOA/D in diesem Buch beschrieben wird. Die Fallstudien werden während dreier Iterationen ausgearbeitet.

Ziele

- ❏ Die Ziele des Buches beschreiben und seinen Themenbereich abgrenzen
- ❏ Objektorientierte Analyse und Design (OOA/D) definieren
- ❏ Ein kurzes OOA/D-Beispiel zeigen
- ❏ Einen Überblick über die UML und die visuelle agile Modellierung geben

1.1 Was werden Sie lernen? Ist es nützlich?

Was bedeutet es, einen guten Objektentwurf zu haben? Dieses Buch ist ein Werkzeug, das Entwicklern und Studenten hilft, die zentralen Fähigkeiten in objektorientierter Analyse und Design (OOA/D) zu lernen. Diese Fähigkeiten sind für die Erstellung von wohlkonstruierter, robuster und wartungsfreundlicher Software mit OO-Verfahren und -Sprachen wie Java oder C# unverzichtbar.

Das Sprichwort »Wer einen Hammer hat, ist noch lange kein Architekt« gilt besonders auch für die Objekttechnik. Eine objektorientierte Sprache (wie beispielsweise Java) zu kennen, ist ein notwendiger, aber unzureichender erster Schritt, um Objektsysteme zu erstellen. Es ist von entscheidender Bedeutung, »in Objekten denken« zu können!

Dies ist eine Einführung in OOA/D unter Verwendung der Unified Modeling Language (UML) und Patterns sowie in die iterative Entwicklung mit agilen Methoden. Dabei wird der Unified Process als Beispiel für einen iterativen Prozess verwendet. Das Buch will *kein* Text für Fortgeschrittene sein; sein Schwerpunkt liegt auf der Vermittlung der Grundlagen: Wie werden Verantwortlichkeiten auf Objekte verteilt? Welche Aspekte der UML-Notation werden am häufigsten verwendet? Welche Design Patterns sind am gebräuchlichsten? Gleichzeitig, hauptsächlich in späteren Kapiteln, werden auch einige mittelschwere Themen, wie beispielsweise der Frameworkentwurf und die Analyse der Architektur behandelt.

UML im Gegensatz zu »Denken in Objekten«

Das Buch handelt nicht einfach von der UML. Die *UML* ist eine Standardnotation für die Diagrammerstellung. Eine allgemein akzeptierte Notation ist nützlich, aber es gibt wichtigere OO-Themen zu lernen – insbesondere das »Denken in Objekten«. Die UML ist kein OOA/D und keine Methode, sie ist einfach eine Notation zur Erstellung von Diagrammen. Es ist sinnlos, die UML und vielleicht die Verwendung eines UML-CASE-Werkzeugs zu lernen, aber nicht wirklich zu wissen, wie ein ausgezeichneter OO-Entwurf erstellt wird oder wie ein vorhandener Entwurf begutachtet und verbessert werden kann. Dies ist die schwierige und wichtige Fähigkeit. Folglich ist dieses Buch eine Einführung in den Objektentwurf.

Dennoch benötigen wir eine Sprache für OOA/D und für »Softwareblaupausen«, und zwar sowohl als Denkwerkzeug als auch als Kommunikationsmittel; deshalb untersucht dieses Buch auch, wie die UML im Dienst des OOA/D *angewendet* werden kann und beschreibt gebräuchliche UML-Komponenten.

OOD: Prinzipien und Patterns

Wie sollten *Verantwortlichkeiten* (engl. *responsibilities*) auf Klassen von Objekten verteilt werden? Wie sollten Objekte zusammenarbeiten? Welche Klassen sollten was tun? Dies sind kritische Fragen beim Entwurf eines Systems, und dieses Buch lehrt die klassische OO-Entwurfsmetapher: *Verantwortungsgesteuertes Design* (engl. *responsibility-driven design*). Außerdem können (und wurden) bestimmte bewährte Lösungen für Entwurfsprobleme als Best-Practice-Prinzipien, Heuristiken oder *Patterns* beschrieben werden – benannte Formeln zur Lösung spezifischer Probleme, die bestimmte Entwurfsprinzipien exemplarisch kodifizieren. Indem dieses Buch lehrt, wie Patterns oder Prinzipien *angewendet* werden, beschleunigt es den Lernprozess und den Erwerb der Fähigkeit, diese grundlegenden Idiome des Objektentwurfs erfolgreich in der Praxis einzusetzen.

Fallstudien

Diese Einführung in OOA/D wird durch einige *fortlaufende Fallstudien* illustriert, die das ganze Buch durchziehen. Dabei werden Analyse und Entwurf so eingehend behandelt, dass einige der schwierigen Detailprobleme zu Tage treten, die typisch für Aufgaben aus der Praxis sind. Es wird gezeigt, mit welchen Überlegungen diese Probleme gelöst werden können.

Use Cases

OOD (und der gesamte Softwareentwurf) ist eng mit einer Aktivität verbunden, die als die Vorbedingung des Entwurfs aufgefasst werden kann: der *Anforderungsanalyse* (engl. *requirements analysis*), die häufig schriftliche *Use Cases* (dt. *Anwendungsfälle*) umfasst. Deshalb beginnen die Fallstudien mit einer Einführung in diese Themen, obwohl diese nicht speziell objektorientiert sind.

Iterative Entwicklung, agile Modellierung und ein agiler UP

In Anbetracht der vielen möglichen Aktivitäten von den Anforderungen bis zur Implementierung stellt sich die Frage, wie ein Entwickler oder ein Team vorgehen sollte. Anforderungsanalyse und OOA/D müssen im Kontext eines Entwicklungsprozesses präsentiert und eingeübt werden. Hier wird ein *agiler* (leichter, flexibler) Ansatz zu dem bekannten *Unified Process* (UP) als das *Muster* eines *iterativen Entwicklungsprozesses* verwendet, in dem diese Themen eingeführt werden. Die behandelten Analyse- und Entwurfsthemen sind jedoch auf viele Ansätze anwendbar; wenn sie im Kontext eines agilen UP gelernt werden, vermindert dies nicht ihre Anwendbarkeit auf andere Methoden, wie beispielsweise Scrum, Feature-Driven Development, Lean Development, Crystal Methods usw.

Zusammenfassend kann man also sagen, dass dieses Buch Studenten und Entwicklern hilft …

- ❏ Prinzipien und Patterns anzuwenden, um bessere Objektentwürfe zu erstellen
- ❏ iterativ einem Satz gebräuchlicher Aktivitäten in Analyse und Entwurf zu folgen, die auf einem agilen Ansatz basieren (hier wird der UP als Beispiel verwendet)
- ❏ häufig verwendete Diagramme in der UML-Notation zu erstellen

Diese Fähigkeiten werden anhand durchgehender Fallstudien vermittelt, die über mehrere Iterationen entwickelt werden.

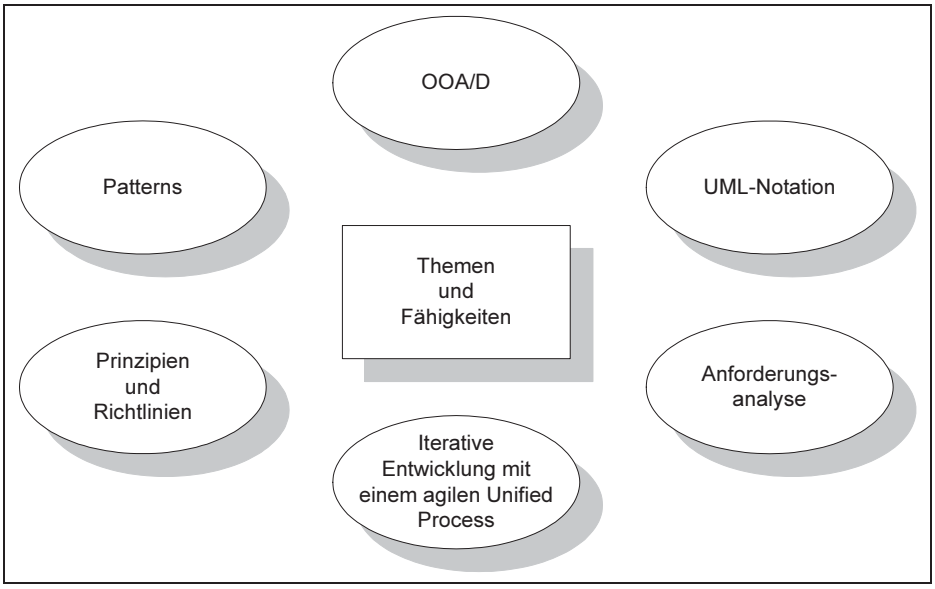


Abbildung 1.1: Die behandelten Themen und Fähigkeiten

Viele andere Fähigkeiten sind wichtig!

Dies ist nicht das *Komplette Buch der Software*; es ist hauptsächlich eine Einführung in OOA/D, die UML und die iterative Entwicklung, während verwandte Themen nur gestreift werden. Software zu erstellen, erfordert zahlreiche andere Fähigkeiten und Schritte; beispielsweise sind Usability Engineering, User Interface Design und Datenbankentwurf ebenfalls erfolgsentscheidend.

1.2 Das wichtigste Lernziel?

Bei einer Einführung in OOA/D gibt es viele mögliche Aktivitäten und Artefakte und zahlreiche Prinzipien und Richtlinien. Nehmen Sie an, wir müssten eine einzige praktische Fähigkeit aus allen hier genannten Themen auswählen – eine Überlebensfähigkeit –, welche wäre das?

> Eine kritische Fähigkeit in der OO-Entwicklung ist die gekonnte Zuweisung von Verantwortlichkeiten an Softwareobjekte.

Warum? Weil dies eine Aktivität ist, die ausgeführt werden muss – entweder beim Zeichnen eines UML-Diagramm oder beim Programmieren – und die einen großen Einfluss auf die Robustheit, die Wartbarkeit und die Wiederverwendbarkeit von Softwarekomponenten hat.

Natürlich gibt es andere wichtige Fähigkeiten im OOA/D, aber die *Zuweisung von Verantwortlichkeiten* wird in dieser Einführung betont, weil es einerseits schwierig ist, diese Fähigkeit zu meistern (es gibt viele »Freiheitsgrade« oder Alternativen), sie andererseits aber lebenswichtig ist. Bei einem echten Projekt hat ein Entwickler möglicherweise keine Gelegenheit, irgendwelche anderen Modellierungsaktivitäten durchzuführen – der Entwicklungsprozess besteht hauptsächlich im »Druck, mit dem Codieren anzufangen«. Doch selbst in dieser Situation ist es unvermeidlich, Verantwortlichkeiten zuzuweisen.

Folglich betonen die Entwurfsschritte in diesem Buch Prinzipien der Zuweisung von Verantwortlichkeiten.

> Es werden neun grundlegende Prinzipien des Objektentwurfs und der Zuweisung von Verantwortlichkeiten präsentiert und angewendet. Sie werden in Form einer Lernhilfe namens *GRASP* zusammengefasst und haben Namen wie *Information Expert* oder *Creator.*

1.3 Was sind Analyse und Design?

Analyse ist eine Untersuchung des Problems und der Anforderungen, nicht der Lösung. Ein Beispiel: Wenn ein neues Online-Handelssystem gewünscht wird, wie soll es eingesetzt werden? Was sind seine Funktionen?

»Analyse« ist ein umfassender Terminus, der am besten qualifiziert oder eingeschränkt wird. So ist beispielsweise die *Anforderungsanalyse* eine Untersuchung der Anforderungen oder *objektorientierte Analyse* eine Untersuchung der Domänenobjekte.

Entwurf oder *Design* betont die *konzeptuelle Lösung* (in Software und Hardware), die die Anforderungen erfüllt, im Gegensatz zur Implementierung der Lösung. Ein Beispiel: Eine Beschreibung eines Datenbankschemas und von Softwareobjekten. Entwurfsideen lassen häufig systemnahe oder »offensichtliche« Details weg – offensichtlich für den angestrebten Konsumenten. Letztlich können Entwürfe implementiert werden, und die Implementierung (wie beispielsweise in Code) drückt den tatsächlichen und vollständigen realisierten Entwurf aus.

Wie der Terminus »Analyse« wird auch der Terminus »Entwurf« am besten qualifiziert und eingeschränkt: *objektorientierter Entwurf* oder *Datenbankentwurf.*

Eine brauchbare Analyse und ein passender Entwurf sind in dem Ausdruck *Tue das Richtige (Analyse), und tue es richtig (Entwurf)* zusammengefasst worden.

1.4 Was sind objektorientierte Analyse und Design?

Bei der *objektorientierten Analyse* liegt die Betonung darauf, die Objekte – oder Konzepte – in dem Problembereich zu finden und zu beschreiben. Ein Beispiel: Bei einem Fluginformationssystem sind wichtige Konzepte unter anderem: `Plane` (Flugzeug), `Flight` (Flug) und `Pilot`.

Beim *objektorientierten Entwurf* (synonym: *Design*; abgekürzt einfach: *Objektentwurf*) liegt die Betonung darauf, geeignete Softwareobjekte und ihr Zusammenwirken (engl. *collaboration*; dt. *Zusammenarbeit*) zu definieren, um die Anforderungen zu erfüllen.[1] Ein Beispiel: Ein `Plane`-Softwareobjekt kann ein `tailNumber`-Attribut und eine `getFlightHistory`-Methode haben (siehe Abbildung 1.2).

Schließlich werden während der Implementierung oder der objektorientierten Programmierung Entwurfsobjekte wie beispielsweise eine `Plane`-Klasse in Java implementiert.

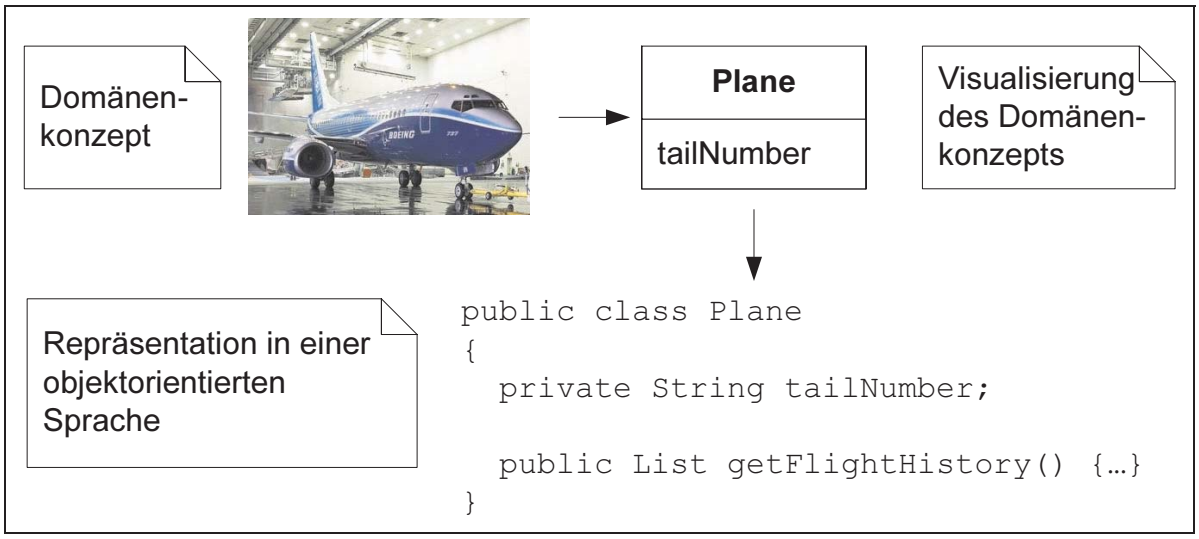
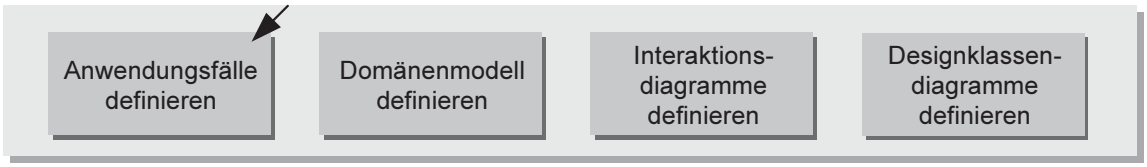


Abbildung 1.2: Objektorientierung betont die Repräsentation von Objekten.

1.5 Ein kurzes Beispiel

Bevor wir uns in die Einzelheiten der iterativen Entwicklung, Anforderungsanalyse, UML und OOA/D vertiefen, wollen wir in diesem Abschnitt aus der Vogelperspektive einen Blick auf einige Schlüsselschritte und -diagramme werfen. Zu diesem Zweck verwenden wir ein einfaches Beispiel – ein »Würfelspiel«, in dem Software einen Spieler simuliert, der zwei Würfel wirft. Bei der Summe sieben gewinnt er, andernfalls verliert er.

Use Cases definieren



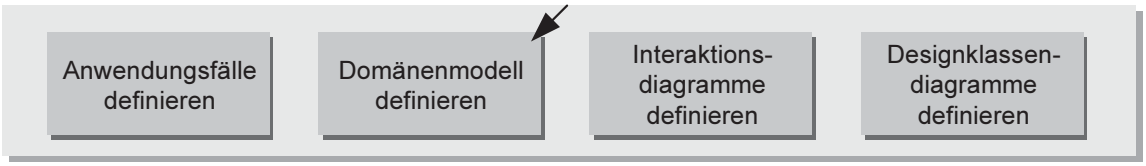
1. *Kollaborieren* und *zusammenarbeiten* werden in diesem Buch als Synonyme verwendet, wobei *kollaborieren* nicht die übliche, politisch motivierte negative Konnotation hat!

Die Anforderungsanalyse kann Geschichten oder Szenarios umfassen, wie Leute die Anwendung einsetzen; diese können als *Use Cases* (dt. *Anwendungsfälle*; im Folgenden wird durchgehend die englische Bezeichnung verwendet) geschrieben werden.

Use Cases sind kein objektorientiertes Artefakt, sondern einfach niedergeschriebene Geschichten. Sie sind jedoch ein beliebtes Werkzeug der Anforderungsanalyse. Beispielsweise ist hier eine kurze Version des Use Cases *Play a Dice Game*:

Play a Dice Game: Player (Spieler) fordert einen Wurf der Würfel an. System präsentiert Ergebnisse: Wenn die Summe der gewürfelten Augen sieben ist, gewinnt Spieler; andernfalls verliert Spieler.

Ein Domänenmodell definieren



Die objektorientierte Analyse befasst sich mit der Erstellung einer Beschreibung der Domäne aus der Perspektive von Objekten. Sie identifiziert Konzepte, Attribute und Assoziationen, die für die Aufgabenstellung als relevant erachtet werden.

Das Ergebnis kann in einem *Domänenmodell* ausgedrückt werden, das die *relevanten* konzeptuellen Klassen oder Objekte zeigt.

Beispielsweise wird in Abbildung 1.3 ein Teilmodell der Domäne gezeigt.

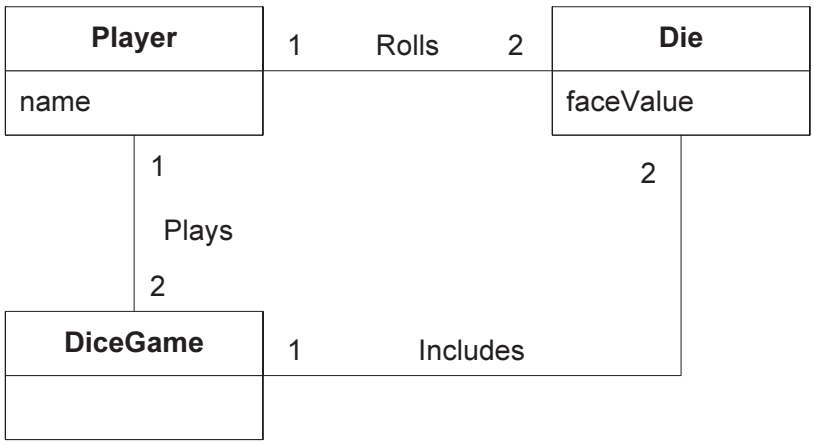


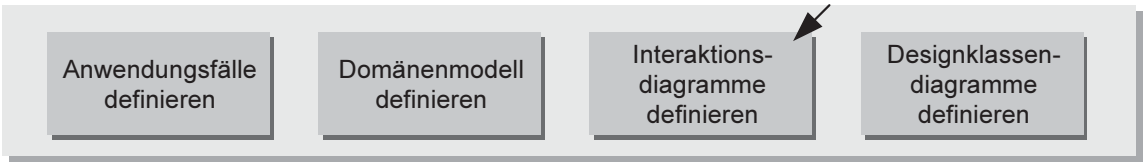
Abbildung 1.3: Teilmodell der Domäne des Würfelspiels

Dieses Modell zeigt die beachtenswerten Konzepte `Player` (Spieler), `Die` (Würfel) und `DiceGame` (Würfelspiel) mit ihren Assoziationen und Attributen.

Beachten Sie, dass ein Domänenmodell keine Beschreibung von Softwareobjekten darstellt; es ist eine Visualisierung der Konzepte oder der mentalen Modelle eines Gegenstandsbereiches.[2] Deshalb wird das Modell auch als *konzeptuelles* (*begriffliches) Objektmodell* bezeichnet.

2. A.d.Ü.: Larman verwendet hier den Begriff *real-world domain*, der aber der Sache nicht gerecht wird. Das Attribut *real-world* suggeriert eine »wirkliche« oder »physische« (Außen-)Welt; *Domäne* kann letztlich alles sein, was in Software modelliert und Gegenstand einer Anwendung sein soll, also beispielsweise auch nur Gedachtes.

Objekten Verantwortlichkeiten zuweisen und Interaktionsdiagramme zeichnen



Der objektorientierte Entwurf befasst sich damit, Softwareobjekte, ihre Verantwortlichkeiten und Kollaborationsbeziehungen zu definieren. Das *Sequenzdiagramm* (eine Art von UML-Interaktionsdiagramm) ist eine gebräuchliche Notation, um diese Kollaborationsbeziehungen zu veranschaulichen. Es zeigt den Austausch von Nachrichten zwischen Softwareobjekten, und deshalb den Aufruf von Methoden.

Ein Beispiel: Das Sequenzdiagramm aus Abbildung 1.4 illustriert einen OO-Softwareentwurf, in dem Nachrichten an Instanzen der Klassen `DiceGame` und `Die` gesendet werden. Beachten Sie, dass diese Abbildung zeigt, wie die UML praktisch angewendet wird: durch Skizzierung auf einem Whiteboard (einer weißen Wandtafel für Filzschreiber).

Während die Würfel im echten Leben von einem *Spieler* geworfen werden, simuliert in dem Softwareentwurf das `DiceGame`-Objekt diese Aufgabe (das heißt, es sendet Nachrichten an die `Die`-Objekte). Softwareobjektentwürfe und Programme nehmen den Gegenstandsbereich als Vorlage, aber sie sind *keine* direkten Modelle oder Simulationen des Gegenstandsbereichs.

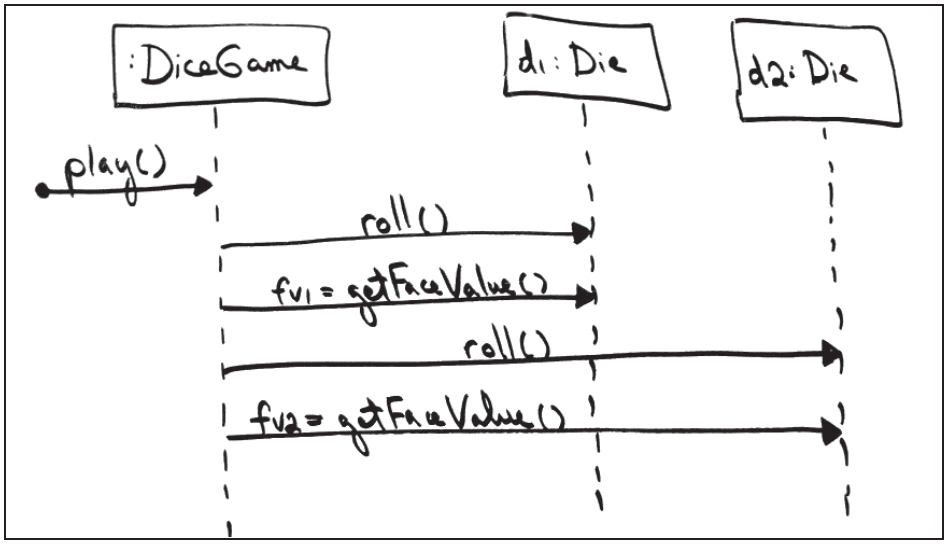
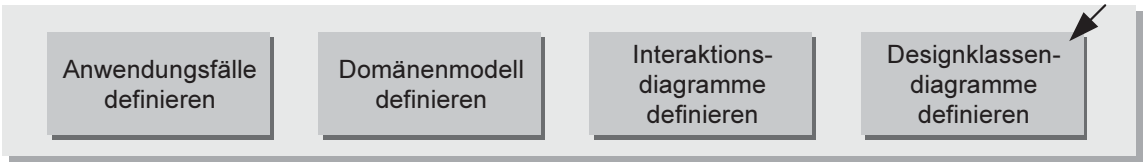


Abbildung 1.4: Sequenzdiagramm, das die Nachrichten zwischen Softwareobjekten illustriert

Entwurfsklassendiagramme definieren



Zusätzlich zu einer *dynamischen* Sicht kollaborierender Objekte, die in Interaktionsdiagrammen gezeigt werden, gibt es eine *statische* Sicht der Klassendefinitionen; sie wird in einem *Entwurfsklassendiagramm* gezeigt und illustriert die Attribute und Methoden der Klassen.

Ein Beispiel: In dem Würfelspiel führt eine Betrachtung des Sequenzdiagramms zu dem partiellen Entwurfsklassendiagramm aus Abbildung 1.5. Da an ein `DiceGame`-Objekt eine `play`-Nachricht gesendet wird, muss die `DiceGame`-Klasse eine `play`-Methode enthalten, während die Klasse `Die` die Methoden `roll` und `getFaceValue` enthalten muss.

Im Gegensatz zu dem Domänenmodell, das Klassen des Gegenstandsbereiches zeigt, stellt dieses Diagramm Softwareklassen dar.

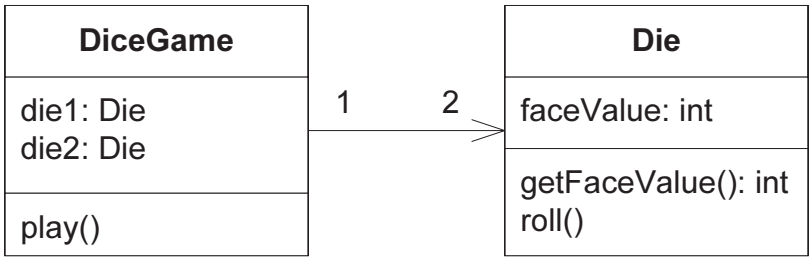


Abbildung 1.5: Partielles Entwurfsklassendiagramm

Beachten Sie, dass einige Klassennamen und Inhalte ähnlich sind, obwohl dieses Entwurfsklassendiagramm nicht dasselbe wie das Domänenmodell ist. Auf diese Weise können OO-Entwürfe und -Sprachen eine so genannte *Lower Representational Gap* (*LRG*; dt. wörtlich »schmalere Repräsentationslücke«) zwischen den Softwarekomponenten (des Entwurfsmodells) und unseren mentalen Modellen (Konzepten, Begriffen) einer Domäne unterstützen. Das verbessert das Verständnis.

Zusammenfassung

Das Würfelspiel ist ein einfaches Problem, das hier gezeigt wurde, um den Fokus auf einige Schritte und Artefakte bei Analyse und Entwurf zu lenken. Um diese Einführung einfach zu halten, wurde nicht die gesamte verwendete UML-Notation erklärt. In späteren Kapiteln werden Analyse und Entwurf sowie diese Artefakte ausführlicher dargestellt.

1.6 Was ist die UML?

Ein Zitat:

> Die Unified Modeling Language ist eine visuelle Sprache zur Spezifikation, Konstruktion und Dokumentation der Artefakte von Systemen [OMG03a].

Das Wort *visuell* in der Definition ist ein Schlüsselpunkt: Die UML ist de facto die *Standardnotation für die Erstellung von Zeichnungen* (mit etwas Text), die mit Software – hauptsächlich OO-Software – zu tun haben.

Dieses Buch behandelt nicht alle winzigen Aspekte der UML, deren Notation sehr umfangreich ist. Es konzentriert sich auf die häufig verwendeten Diagramme, die gebräuchlichsten Features innerhalb dieser Diagramme und die Kernnotation, die wahrscheinlich auch in künftigen UML-Versionen stabil bleiben werden.

Die UML definiert verschiedene *UML-Profile*, die spezielle themenspezifische Untermengen der Notation enthalten. Ein Beispiel ist das *UML EJB Profile* für die Diagrammerstellung für Enterprise JavaBeans.

Auf einer tieferen Ebene – die hauptsächlich für Anbieter von CASE-Werkzeugen für die *Model Driven Architecture* (MDA) von Interesse ist – gibt es das so genannte *UML-Metamodell*, das der UML-Notation zugrunde liegt und die Semantik der Modellierungselemente beschreibt. Es zählt nicht zu den Dingen, die ein Entwickler lernen muss.

1.6.1 Drei Methoden, um die UML anzuwenden

In [Fowler03] werden drei Methoden vorgestellt, wie Entwickler die UML anwenden:

- *UML als Skizze* – Informelle und unvollständige Diagramme (häufig Handskizzen auf Whiteboards), die erstellt werden, um schwierige Teile des Problems oder des Lösungsraums zu analysieren und die Stärken einer visuellen Sprache zu nutzen.
- *UML als Blaupause* – verhältnismäßig detaillierte Entwurfsdiagramme, die für die folgenden Zwecke gebraucht werden: 1) Reverse Engineering, um vorhandenen Code mit UML-Diagrammen zu visualisieren und besser zu verstehen. 2) Codegenerieren (eine vorwärts gerichtete Technik).
  - Beim Reverse Engineering liest ein UML-Werkzeug die Quell- oder Binärprogramme und generiert daraus (üblicherweise) UML-Paket-, -Klassen- und -Sequenzdiagramme. Diese »Blaupausen« können dem Leser helfen, sich ein Gesamtbild von den Elementen, der Struktur und den Kollaborationsbeziehungen zu machen.
  - Vor dem Programmieren können einige detaillierte Diagramme Hinweise für die Code-Erzeugung (beispielsweise in Java) liefern. Der Code kann entweder manuell oder automatisch mit einem Werkzeug generiert werden. Üblicherweise werden die Diagramme für einen Teil des Codes benutzt, während andere Teile des Codes durch einen Entwickler beim Codieren ergänzt werden (möglicherweise ebenfalls anhand von UML-Skizzen).
- *UML als Programmiersprache* – Vollständige ausführbare Spezifikation eines Softwaresystems in UML. Ausführbarer Code wird automatisch generiert, wird aber normalerweise nicht von Entwicklern gesehen oder modifiziert; die Entwickler arbeiten nur in der UML-»Programmiersprache«. Diese Anwendung der UML erfordert eine praktische Methode, um das gesamte Verhalten oder die gesamte Logik in Diagrammen darzustellen (wobei wahrscheinlich Interaktions- oder Zustandsdiagramme verwendet werden). Diese Methode ist noch nicht ausgereift, was die Theorie, die Robustheit der Werkzeuge und die Nutzbarkeit angeht.

Agile Modellierung betont *UML als Skizze*; dies ist eine gebräuchliche Methode, um die UML anzuwenden; häufig zahlt sich der (normalerweise geringe) Zeitaufwand mehrfach aus. UML-Werkzeuge können nützlich sein, aber ich rate Entwicklern, auch bei der Anwendung der UML einen agilen Modellierungsansatz in Betracht zu ziehen.

➔ ***Siehe auch:*** *Abschnitt 2.7, Was ist agile Modellierung?*

UML und das »Silver Bullet«-Denken

Es gibt einen bekannten Aufsatz von Dr. Frederick Brooks aus dem Jahre 1968 mit dem Titel »No Silver Bullet«, der auch in seinem klassischen Buch *Mythical Man-Month* (dt. *Vom Mythos des Mann-Monats* im mitp-Verlag) veröffentlicht wurde. Unbedingt lesenswert! Ein wesentlicher Punkt darin ist, dass es ein grundlegender Fehler ist (der bis jetzt endlos oft wiederholt wurde), anzunehmen, dass es ein spezielles Softwarewerkzeug oder -verfahren gibt, das die Produktivität, die Anzahl der Fehler, die Zuverlässigkeit oder die Einfachheit drastisch um Größenordnungen steigern bzw. verringern kann. *Und Werkzeuge sind kein Ersatz für mangelnde Entwurfsfähigkeiten.*

Dennoch werden Sie – normalerweise von Tool-Anbietern – Behauptungen hören, dass durch Zeichnen von UML-Diagrammen alles viel besser wird; oder dass Tools für eine Model Driven Architecture (MDA), die auf der UML basieren, die lang gesuchte »Silver Bullet« sind und einen Durchbruch in der Softwareentwicklung darstellen.

Zeit für einen Realitätscheck: Die UML ist einfach eine Standardnotation zur Diagrammerstellung – Kästen, Linien usw. Visuelle Modellierung mit einer gebräuchlichen Notation kann eine großartige Hilfe sein, aber sie ist kaum so wichtig wie die Fähigkeit, entwerfen und in Objekten denken zu können. Solche Entwurfsfähigkeiten sind ganz andere und wichtigere Fähigkeiten, und sie lassen sich nicht erwerben, indem man die UML-Notation oder die Bedienung eines CASE- oder MDA-Werkzeugs erlernt. Eine Person, die nicht über gute OO-Entwurfs- und Programmierfähigkeiten verfügt, aber UML-Diagramme zeichnen kann, zeichnet einfach nur schlechte Entwürfe. Ich lege Ihnen den Artikel *Death by UML Fever* [Bell04] (der von dem UML-Schöpfer Grady Booch gebilligt wird) ans Herz, wenn Sie mehr über dieses Thema wissen wollen. In die gleiche Richtung zielt *What UML Is and Isn't* [Larman04].

Deshalb ist dieses Buch eine Einführung in OOA/D und die *Anwendung* der UML zur Unterstützung eines gekonnten OO-Entwurfs.

1.6.2 Drei Perspektiven zur Anwendung der UML

Die UML beschreibt rohe Diagrammtypen wie beispielsweise Klassendiagramme und Sequenzdiagramme. Es zwingt diesen keine Modellierungsperspektive auf. Ein Beispiel: Dieselbe UML-Klassendiagramm-Notation kann verwendet werden, um Bilder von Konzepten der Domäne (des Gegenstandbereiches, der »wirklichen« Welt) oder von Softwareklassen in Java zu zeichnen.

Diese Einsicht wurde von der objektorientierten Methode *Syntropy* hervorgehoben [CD94]. Das bedeutet: Dieselbe Notation kann für drei Perspektiven und Typen von Modellen verwendet werden (Abbildung 1.6):

1. *Konzeptuelle (begriffliche) Perspektive* – die Diagramme werden so interpretiert, dass sie Dinge in einer Situation des interessierenden Gegenstandsbereiches beschreiben.
2. *Spezifikations-(Software-)Perspektive* – die Diagramme beschreiben (mit derselben Notation wie die konzeptuelle Perspektive) Softwareabstraktionen oder Komponenten mit Spezifikationen und Interfaces, aber ohne Bindung an eine bestimmte Implementierung (wie beispielsweise eine spezielle Klasse in C# oder Java).
3. *Implementierungs-(Software-)Perspektive* – die Diagramme beschreiben Softwareimplementierungen in einem bestimmten Verfahren (wie beispielsweise Java).

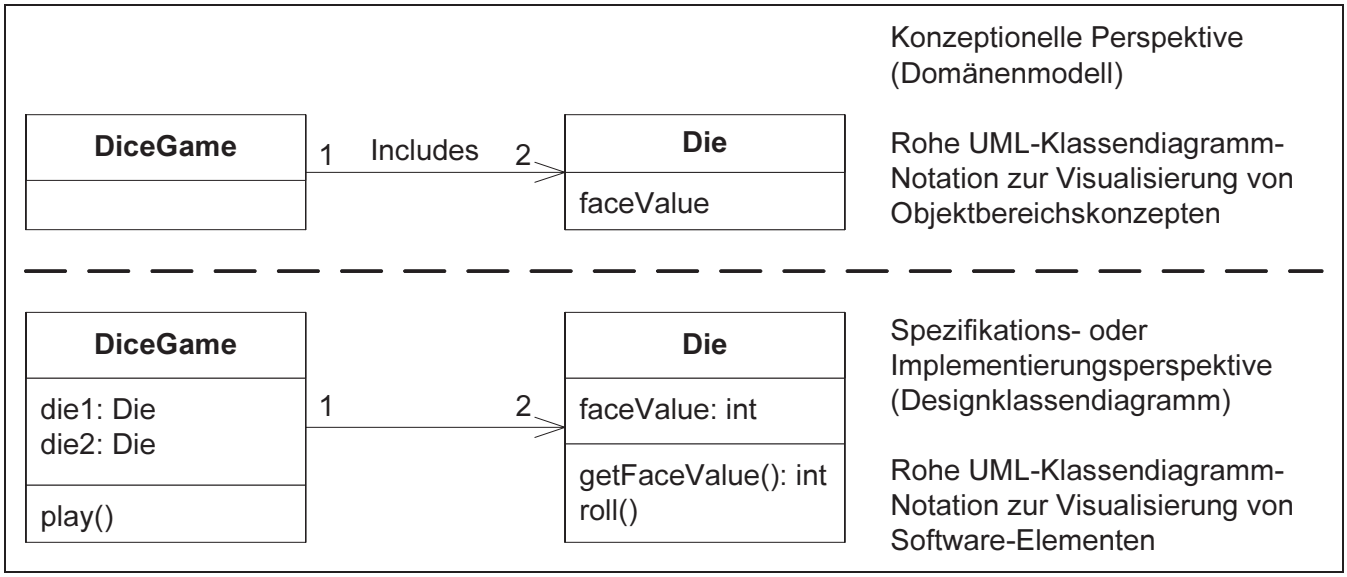


Abbildung 1.6: Verschiedene Perspektiven mit UML

Wir haben bereits in den Abbildungen 1.3 und 1.5 Beispiele dafür gesehen, dass dieselbe UML-Klassendiagramm-Notation verwendet wird, um ein Domänenmodell und ein Entwurfsmodell zu visualisieren.

In der Praxis wird die Spezifikationsperspektive (in der Entscheidungen über das Zielverfahren wie beispielsweise Java im Gegensatz zu .NET aufgeschoben sind) selten für den Entwurf verwendet; der größte Teil der softwareorientierten UML-Diagrammerstellung geht von einer Implementierungsperspektive aus.

Die Bedeutung des Terminus »Klasse« in verschiedenen Perspektiven

In der rohen UML werden die rechteckigen Kästen, die in Abbildung 1.6 zu sehen sind, als *classes* (Klassen) bezeichnet, aber dieser Terminus umfasst verschiedene Phänomene – physische Dinge, abstrakte Begriffe, Softwaredinge, Ereignisse usw. (Eine UML-Klasse ist ein spezieller Fall des allgemeinen UML-Modellelements *classifier* (*Klassifizierer*, dt. auch *Classifier*) – etwas, das eine Struktur und ein Verhalten hat, einschließlich Klassen, Akteure, Interfaces und Use Cases.)

Eine Methode stülpt der rohen UML eine alternative Terminologie über. Wenn beispielsweise in dem UP in dem Domänenmodell UML-Kästen gezeichnet werden, werden sie als *domain concepts* oder *conceptual classes* (*Domänenkonzepte* oder *konzeptuelle Klassen*) bezeichnet; das Domänenmodell zeigt eine konzeptuelle Perspektive. Sie werden im UP als *design classes* (*Designklassen* oder *Entwurfsklassen*) bezeichnet, wenn UML-Kästen in dem Entwurfsmodell gezeichnet werden; das Entwurfsmodell zeigt je nach Wunsch des Modellierers eine Spezifikations- oder Implementierungsperspektive.

Um keine Verwirrung zu stiften, werden die klassenbezogenen Termini in diesem Buch konsistent mit der UML und dem UP verwendet, und zwar folgendermaßen:

- *Konzeptuelle Klasse* (engl. *conceptual class*) – Konzept oder Ding des Gegenstandsbereiches. Eine konzeptuelle oder wesentliche Perspektive. Das UP Domain Model enthält konzeptuelle Klassen.
- *Softwareklasse* (engl. *software class*) – eine Klasse, die eine Softwarekomponente in einer Spezifikations- oder Implementierungsperspektive repräsentiert, und zwar unabhängig von dem Prozess oder der Methode.
- *Implementierungsklasse* (engl. *implementation class*) – eine Klasse, die in einer speziellen OO-Sprache wie beispielsweise Java implementiert ist.

UML 1 und UML 2

Ende 2004 wurde die UML-Version 2 freigegeben. Dieser Text basiert auf UML 2; tatsächlich wurde die hier verwendete Notation von einigen Schlüsselmitgliedern des UML-2-Spezifikationsteams sorgfältig begutachtet.

Warum kommt die UML in den ersten Kapiteln so selten vor?

Schwerpunktmäßig ist dies kein Buch über die UML-Notation, sondern es untersucht ein größeres Feld, in dem die UML, Patterns und ein iterativer Prozess im Kontext des OOA/D und der verwandten Anforderungsanalyse angewendet werden. Vor dem OOA/D erfolgt normalerweise eine Anforderungsanalyse. Deshalb werden in den ersten Kapiteln die wichtigen Themen der Use Cases und der Anforderungsanalyse eingeführt. Danach folgen Kapitel über OOA/D und nähere Einzelheiten über die UML.

1.7 Visuelle Modellierung ist etwas Gutes

Auf die Gefahr hin, auf Gemeinplätzen herumzureiten, möchte ich betonen, dass das Zeichnen und Lesen der UML impliziert, dass wir visueller arbeiten und die Stärken unseres Gehirns nutzen, schnell Symbole, Einheiten und Beziehungen in einer (vorwiegend) 2D-Kästen-und-Linien-Notation zu erfassen.

Diese alte, einfache Einsicht geht häufig unter all den UML-Details und -Werkzeugen verloren. Dies sollte nicht sein! Diagramme helfen uns, eher das größere Bild zu sehen oder zu erforschen und Beziehungen zwischen Analyse- oder Softwareelementen zu erkennen, während sie es uns gleichzeitig erlauben, uninteressante Einzelheiten zu ignorieren oder zu verbergen. Das ist der einfache und wesentliche Wert der UML – oder einer Diagrammerstellungssprache überhaupt.

1.8 Geschichte

Die Geschichte des OOA/D hat viele Zweige, und die folgende kurze Zusammenfassung kann nicht allen gerecht werden, die zu ihr beigetragen haben. In den 60er und 70er Jahren wurden die ersten OO-Programmiersprachen wie Simula und Smalltalk entwickelt. Die Schlüsselpersonen waren beispielsweise Kristen Nygaard und insbesondere Alan Kay, der visionäre Computerwissenschaftler, der Smalltalk erfand. Kay prägte die Termini *objektorientiertes Programmieren* und *Personal Computing* und trug dazu bei, die Ideen des modernen PC zu entwickeln, während er am Xerox PARC arbeitete. (Kay begann seine Arbeit an der OO und dem PC in den 60er Jahren als graduierter Student. Im Dezember 1979 besuchte Steve Jobs, Mitbegründer und CEO von Apple, auf Wunsch von Apples bedeutendem Jef Raskin (dem leitenden Entwickler des Mac) Alan Kay und Forschungsteams (einschließlich Dan Ingalls, der die Vision von Kay implementierte) am Xerox PARC, um sich den Smalltalk-Personal-Computer demonstrieren zu lassen. Überwältigt von dem, was er sah – ein Bitmap-Display mit einem grafischen UI und überlappenden Fenstern, OO-Programmierung und vernetzte PCs –, kehrte er mit einer neuen Vision zu Apple zurück (auf die Raskin gehofft hatte), und der Apple Lisa und der Macintosh wurden geboren.

Aber OOA/D wurde in dieser Zeit nur informell betrieben, und erst 1982 trat das OOD als eigenständiges Thema in Erscheinung. Dieser Meilenstein wurde gesetzt, als Grady Booch (ebenfalls einer der UML-Begründer) den ersten Aufsatz mit dem Titel *Object-Oriented Design* schrieb und damit wahrscheinlich diesen Begriff prägte [Booch82]. Viele andere bekannte OOA/D-Pioniere entwickelten ihre Ideen während der 80er Jahre: Kent Beck, Peter Coad, Don Firesmith, Ivar Jacobson (ein UML-Begründer), Steve Mellor, Bertrand Meyer, Jim Rumbaugh (ein UML-Begründer), Rebecca Wirfs-Brock und andere. Meyer veröffentlichte 1988 eines der frühen einflussreichen Bücher, *Object-Oriented Software Construction*. Und Mellor und Schlaer veröffentlichten im selben Jahr *Object-Oriented Systems Analysis*, in dem sie den Begriff der *objektorientierten Analyse* prägten. Peter Coad entwickelte in den späten 80er Jahren eine komplette OOA/D-Methode und veröffentlichte 1990 und 1991 die Zwillingsbände *Object-Oriented Analysis* und *Object-Oriented Design*. Ebenfalls in 1990 beschrieben Wirfs-Brock und andere den verantwortungsgesteuerten Entwurfsansatz zum OOD in ihrem populären *Designing Object-Oriented Software*. In 1991 wurde zwei sehr populäre OOA/D-Bücher veröffentlicht. Das eine von Rumbaugh et al. beschrieb die OMT-Methode, *Object-Oriented Modeling and Design*. Das andere beschrieb die Booch-Methode, *Object-Oriented Design with applications*. 1992 veröffentlichte Jacobson das populäre *Object-Oriented Software Engineering*, das nicht nur OOA/D, sondern auch Use Cases für Anforderungen propagierte.

Die UML begann 1994 als gemeinsames Unternehmen von Booch und Rumbaugh nicht nur, um eine gemeinsame Notation zu schaffen, sondern auch um ihre beiden Methoden – die Booch- und die OMT-Methode – zu vereinigen. Deshalb wurde der erste öffentliche Entwurf von dem, was heute die UML ist, als die *Unified Method* vorgestellt. Bald stieß Ivar Jacobson, der Schöpfer der Objectory-Methode, zu ihnen und der Rational Corporation hinzu; und als Gruppe wurden sie unter dem Namen die *three Amigos* bekannt. Zu diesem Zeitpunkt entschieden sie sich, den Umfang ihrer Bemühungen zu reduzieren und sich auf eine allgemeine Diagrammerstellungsnotation – die UML – statt auf eine allgemeine Methode zu konzentrieren. Dies war nicht nur ein Versuch, den Umfang zu reduzieren; die Object Management Group (OMG, ein Branchenverband, der sich um die Schaffung von OO-Standards bemüht) wurde von verschiedenen Werkzeuganbietern davon überzeugt, dass ein offener Standard benötigt wurde. Deshalb wurde der Prozess geöffnet, und eine OMG Task Force unter dem Vorsitz von Mary Loomis und Jim Odell koordinierte die ersten Anstrengungen, die 1997 zu UML 1.0 führten. Viele andere trugen zur UML bei; besonders erwähnenswert ist vielleicht Cris Kobryn, ein Vorreiter der weitergehenden Verfeinerung der UML.

Die UML hat sich de facto und de jure zur Standardnotation für Diagramme in der objektorientierten Modellierung entwickelt und wurde in neuen OMG-UML-Versionen immer weiter verfeinert. Die Standards sind im Web verfügbar: `www.omg.org` oder `www.uml.org`.

1.9 Empfohlene Ressourcen

In späteren Kapiteln werden viele verschiedene OOA/D-Texte empfohlen, die für das jeweilige Thema, beispielsweise den OO-Entwurf, relevant sind. Die Bücher im Geschichtsabschnitt sind alle studierenswert – und ihre Kernratschläge sind immer noch anwendbar.

Eine sehr lesbare und beliebte Zusammenfassung der wesentlichen UML-Notation ist *UML Distilled* von Martin Fowler. Sehr empfehlenswert; Fowler hat viele nützliche Bücher geschrieben, in denen er eine praktische und »agile« Haltung vertritt.

Als detaillierte Beschreibung der UML-Notation lohnt sich *The Unified Modeling Language Reference Manual* von Rumbaugh. Beachten Sie, dass dieser Text nicht dazu dient, die Objektmodellierung oder OOA/D zu vermitteln – es ist eine Referenz der UML-Notation.

Die definitive Beschreibung der aktuellen UML-Version finden Sie online unter `www.uml.org` oder `www.omg.org`. Suchen Sie nach *UML Infrastructure Specification* und *UML Superstructure Specification*.

Die visuelle UML-Modellierung mit einer agilen Haltung wird in *Agile Modeling* von Scott Ambler beschrieben. Besuchen Sie auch `www.agilemodeling.com`.

Eine große Sammlung von Links zu OOA/D-Methoden finden Sie unter `www.cetus-links.org` und `www.iturls.com` (suchen Sie den großen englischen Unterabschnitt »Software Engineering« und nicht den chinesischen Abschnitt).

Es gibt viele Bücher über Software Patterns, aber der bahnbrechende Klassiker ist *Design Patterns* von Gamma, Helm, Johnson und Vlissides. Dieser Text ist ein Muss, wenn Sie den Objektentwurf ernsthaft studieren wollen. Doch es handelt sich nicht um eine Einführung. Am besten lesen Sie diesen Text, wenn Sie mit den Grundlagen des Objektentwurfs und der objektorientierten Programmierung vertraut sind. Unter `www.hillside.net` und `www.iturls.com` (englischer Unterabschnitt »Software Engineering«) finden Sie auch Links zu vielen Pattern-Sites.