# Kapitel 1
# Einführung

*»Zeig mir, wie Du baust,*
*und ich sage Dir, wer Du bist.«*
*– Christian Morgenstern*

## 1.1 Weshalb muss Software modelliert werden?

Zu Beginn der Arbeit an diesem Buch fragte mich ein befreundeter Programmierer, weshalb man Software überhaupt modellieren sollte. Er sei im Laufe seiner Ausbildung nur am Rande mit dem Thema in Berührung gekommen und sehe nicht die Notwendigkeit, diese »Zusatzaufgabe« während seiner Entwicklungsarbeit durchzuführen. Dies koste nur zusätzlich Zeit, und seine Programme liefen, auch ohne sie vorher modelliert zu haben.

Um die Unumgänglichkeit der Modellierung großer Softwaresysteme zu veranschaulichen, lassen Sie uns den Softwareentwicklungsprozess mit dem Bau eines Hauses vergleichen.

Bevor die eigentlichen Bauarbeiten beginnen, setzt sich der Bauherr zunächst mit einem Architekten in Verbindung. Er erklärt dem Architekten möglichst genau, wie sein Traumhaus aussehen sollte, wie die Räumlichkeiten aufgeteilt werden sollen und was für eine Lage er bevorzugen würde. Der Architekt setzt die Vorstellungen des Bauherrn in einen Bauplan des Hauses um.

In weiteren gemeinsamen Gesprächen klären der Architekt und der Bauherr alle Details des Bauplans und besprechen mögliche Veränderungen oder auch Einschränkungen am geplanten Haus. Vielleicht fiel dem Bauherrn in der Zwischenzeit ein, dass seine neue Garage doch für zehn und nicht, wie zunächst angenommen, für vier Autos ausgelegt sein sollte. Der Architekt stellt andererseits fest, dass ein Gebäude in der Größe der Cheops-Pyramide den verfügbaren Finanzrahmen sprengt.

Solche Überlegungen müssen zwingend vor dem Bau des Hauses erfolgen, um während der Bauphase bittere Enttäuschungen, Rückschläge und Verzögerungen der Fertigstellung zu vermeiden.

Nach den Abstimmungsgesprächen zwischen Bauherr und Architekt rückt die Planung der Realisierung in den Mittelpunkt der Überlegungen:

- Wie muss das Fundament gegossen werden, um das gesamte Haus tragen zu können?
- Welche Außenverklinkerung soll das Haus erhalten, welche Dachziegel, welche Fensterrahmen?
- Welche und wie viele Arbeiter, Arbeitsgeräte und Maschinen müssen wann und in welcher Menge verfügbar sein?
- Wie stellt man sicher, dass alle Bauarbeiten korrekt durchgeführt werden?

Solche und viele weitere Fragen sind ebenfalls vor Beginn der Bauphase zu klären.

Erst nachdem der Bauherr die Sicherheit erlangt hat, dass der Architekt alle seine Wünsche vollständig und korrekt erfasst und deren Umsetzung geplant hat, gibt er ihm grünes Licht, um mit den eigentlichen Bauarbeiten zu beginnen.

In der darauffolgenden Bauphase definieren die erstellten Baupläne präzise, welche Arbeiten in welcher Reihenfolge von den Bauarbeitern durchzuführen sind. Die Pläne dienen als Kommunikationsgrundlage zwischen dem Architekten und den Bauarbeitern, die bei den Architekturentscheidungen nicht dabei sein konnten.

## 1.2 Die Phasen bei der Softwareentwicklung

Eine sehr ähnliche Vorgehensweise ist auch in der Softwareentwicklung notwendig, um stabile und zuverlässige Softwaresysteme zu realisieren, die nicht an den Anwenderbedürfnissen vorbeizielen.

Die meisten Vorgehensmodelle zur Softwareentwicklung unterscheiden sechs grundlegende Phasen:

- Analyse
- Entwurf
- Implementierung
- Test
- Einsatz
- Wartung

### 1.2.1 Analyse

Während der Analyse werden die Wünsche und Vorstellungen des Auftraggebers (= Bauherr) und (falls verfügbar) der Endanwender (= Familie des Bauherrn) erfasst und modelliert.

Der Softwarearchitekt erörtert mit den oben angesprochenen Beteiligten in Interviews, was die gewünschte Software (= das Traumhaus) leisten und mit welchen







weiteren Systemen sie interagieren soll (= die Lage des Traumhauses). Ebenfalls muss klargestellt werden, welche Anforderungen die neue Software nicht erfüllen kann (= Cheops-Pyramide).

### 1.2.2 Entwurf

In der Entwurfsphase nähert sich Ihr Projekt bereits der Implementierung, weshalb es zu klären gilt, *wie* das Softwaresystem realisiert werden soll. Dabei werden auch system- und architekturorientierte Fragen geklärt, wie z. B.:

- Welche Softwarearchitektur soll verwendet werden? Ist z. B. eine Datenbank einzusetzen (= Fundament)?
- Wie ist die Benutzungsoberfläche des Programms zu gestalten (= Klinker, Dachziegel, Fensterrahmen)?
- Welche Programmiersprachen und Entwicklungsumgebungen sollen verwendet werden? Welches Know-how müssen die Entwickler mitbringen, um das Vorhaben realisieren zu können? Wann und in welchem Umfang sind die Entwickler verfügbar (= Bauarbeiter, Arbeitsgeräte und Maschinen)?
- Welche Software-Qualitätssicherungsmaßnahmen sind einzusetzen (= Kontrolle der Bauarbeiten)?

Das aus dieser Phase entstandene Modell (= Bauplan) dient dem Softwarearchitekten als Kommunikationsgrundlage mit den Programmierern (= Bauarbeitern) und definiert präzise, welche Programmierarbeiten in welcher Reihenfolge durchzuführen sind. Erst nach seiner Erstellung kann die Implementierung des eigentlichen Quellcodes beginnen.

Insgesamt kann davon ausgegangen werden, dass mit der Größe des Projekts auch die Vorteile einer Analyse und eines Entwurfs unter Einsatz einer geeigneten Modellierungssprache überproportional steigen.

### 1.2.3 Implementierung und Dokumentation

In der Implementierungsphase wird der Quelltext der Software erstellt. Die UML-Modellierung aus der Analyse und dem Entwurf repräsentiert einen Systembauplan, der bei der Codegenerierung als Grundlage dient. Darüber hinaus wird UML von CASE-Werkzeugen (**C**omputer **A**ided **S**oftware **E**ngineering) verwendet, um den Quelltext der eingesetzten Programmiersprache automatisch per Codeengineering erzeugen zu lassen. Weil hierbei auch eine ausführliche Dokumentation erarbeitet werden muss, ist es besonders hilfreich, dass CASE-Werkzeuge Ihnen auch bei dieser Aufgabe tatkräftig unter die Arme greifen.

### 1.2.4 Test

Bevor die Software bei dem Kunden eingesetzt werden kann, muss überprüft werden, ob die fachlichen Anforderungen des Kunden realisiert worden sind. Auch hierbei sind die UML-Modelle, die in den vergangenen Phasen erstellt wurden, sehr nützlich, denn auf ihrer Grundlage kann durch Vergleich geprüft werden, ob die einzelnen Details der Anwendungsfälle vollständig in der Anwendung enthalten sind. Darüber hinaus muss die Software in der Testphase ausgiebig auf Bugs untersucht werden. Im geschäftskritischen Umfeld wird die Testphase in weitere Teststufen unterteilt, denn ein Test kann vielfältige Ziele verfolgen. Beispielsweise unterscheidet man zwischen einem Entwicklertest, Validierungstest, Komponententest, Systemtest, Abnahmetest usw.

### 1.2.5 Einsatz

Nach einer erfolgreichen Testphase wird die Software vom Kunden in Betrieb genommen. In der Regel muss die Software hierbei in eine Systemlandschaft integriert werden. Auch hierbei ist der Einsatz von UML nützlich, um bei der Migration mit zahlreichen Anwendungen, Datenbanken usw. den Überblick zu behalten.

### 1.2.6 Wartung und Pflege

Selbst nachdem eine Software vom Kunden abgenommen wurde, zeigen sich häufig Fehler, die im Nachhinein zu korrigieren sind. Aber nicht nur deshalb ist Software Veränderungen ausgesetzt. In den meisten Fällen keimen Erweiterungs- oder Änderungswünsche bei den Nutzern auf, die ein sorgfältig geplantes Änderungsmanagement erforderlich machen. Hierbei zeigt sich erneut, wie vorteilhaft das gewissenhafte Dokumentieren mit UML sein kann, denn die in den vergangenen Phasen eingepflegten Dokumente sorgen nun nicht nur für eine gute Übersicht, sondern auch für einen Einblick bis in einzelne Details des Systems.

## 1.3 Was ist die UML?

Die UML (*Unified Modeling Language*) definiert eine allgemein verwendbare Modellierungssprache (auch *Notation* genannt). Ihr Einsatzgebiet beschränkt sich nicht auf die Softwareentwicklung. Sie stellt Diagramme und Notationselemente (= einzelne Bestandteile der Diagramme) zur Verfügung, mit deren Hilfe sowohl statische als auch dynamische Aspekte beliebiger Anwendungsgebiete modelliert werden können.







Die wesentlichen Vorteile der Unified Modeling Language sind:

- **Eindeutigkeit**
  Die Notationselemente besitzen eine präzise Semantik und sind von vielen Experten definiert und geprüft.
- **Verständlichkeit**
  Die einfach gehaltenen Notationselemente visualisieren grafisch die Aspekte der modellierten Systeme und erleichtern damit das Verständnis.

  Unterschiedliche Diagramme ermöglichen differenzierte Sichtweisen auf das zu modellierende System und betonen oder vernachlässigen bewusst seine Teilaspekte. Damit wird die Kommunikation aller an der Softwareentwicklung beteiligten Personen erleichtert und auf eine stabile Basis gestellt.
- **Ausdrucksstärke**
  Die Ausschöpfung aller Möglichkeiten der verfügbaren Notationselemente erlaubt die nahezu vollständige Definition aller wichtigen Details eines Softwaresystems.
- **Standardisierung und Akzeptanz**
  Weltweit ist die UML in der Softwarebranche im Einsatz. Der *Object Management Group* (OMG), die für die Spezifikation der UML verantwortlich ist, gehören mittlerweile mehr als 800 Unternehmen an.
- **Plattform- und Sprachunabhängigkeit**
  Mit der UML können Sie Softwaresysteme für jede denkbare Plattform und Programmiersprache modellieren. Sie hat ihre Stärken in der objektorientierten Welt, kann aber ohne Weiteres auch für prozedurale Sprachen eingesetzt werden.
- **Unabhängigkeit von Vorgehensmodellen**
  Die UML definiert mit ihren Diagrammen und Notationselementen »Werkzeuge«, um die Spezifizierung, Visualisierung und Dokumentation von Softwaresystemen zu erleichtern. Sie überlässt den Softwareentwicklern die Entscheidung, wie sie diese Werkzeuge am effizientesten nutzen.

## 1.4 Die Geschichte der UML

Um Ihnen einen ersten Eindruck von der UML zu vermitteln, stellt Abbildung 1.1 die Geschichte der UML als ein Zustandsdiagramm dar. Sie werden diese Diagrammart in Kapitel 10 kennenlernen (in Abbildung 1.1 werden der Einfachheit halber einige Details ausgeblendet, die das Diagramm erst vollständig machen würden, wie z. B. die Bezeichnungen der Transitionen zwischen den Zuständen). Mit der folgenden Darstellung der UML-Geschichte dürfte das Diagramm jedoch selbsterklärend sein und bereits die Verständlichkeit der UML im Ansatz demonstrieren:

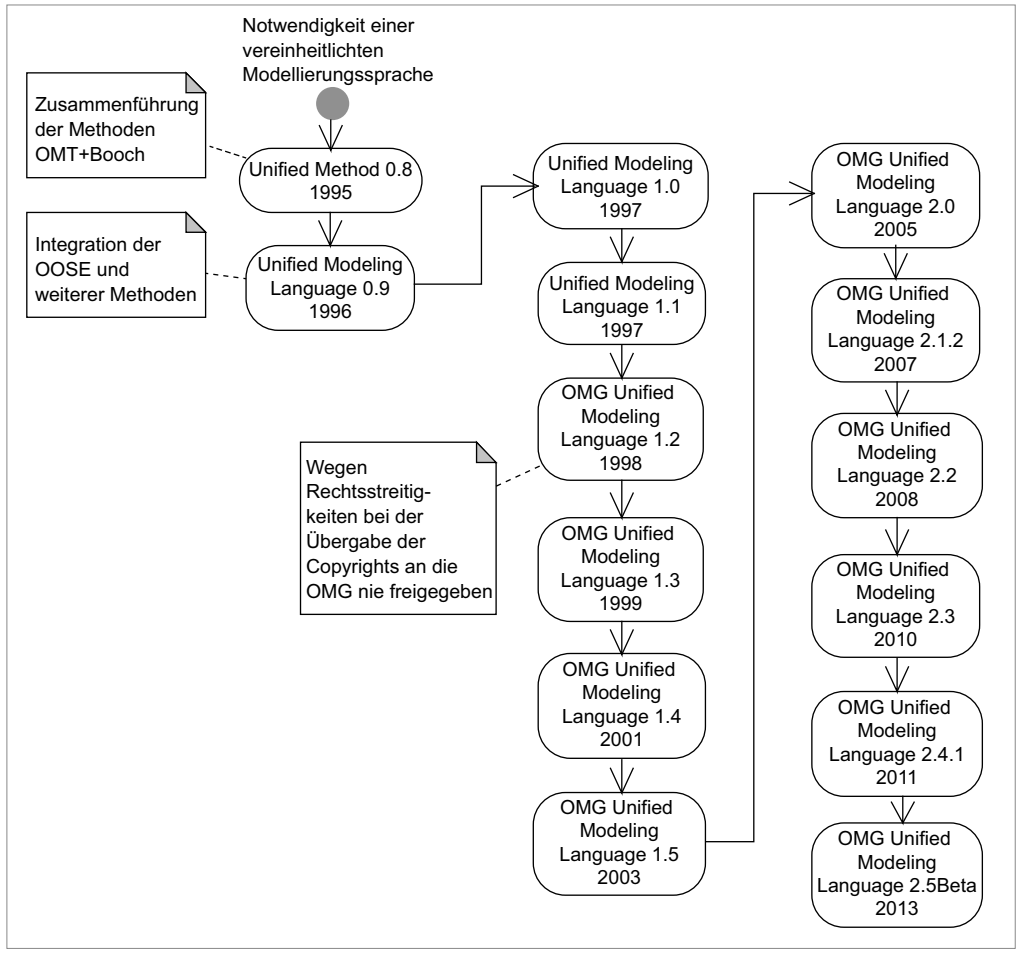


**Abbildung 1.1** Die Geschichte der UML

Die Notwendigkeit der Modellierung von Softwaresystemen wurde bereits in der »Softwarekrise« in den 70er-Jahren erkannt, als die Entwicklung immer größerer und komplexerer Softwaresysteme zunehmend unbeherrschbar wurde. In den frühen 90er-Jahren standen sich viele inkompatible und teilweise widersprüchliche Notationen mit unterschiedlichen Notationselementen gegenüber. Die *Object Modeling Technique* (OMT) von James Rumbaugh, die Booch-Methode von Grady Booch, das *Object-Oriented Software Engineering* (OOSE) von Ivar Jacobson oder die *Object-Oriented Analysis* (OOA) von Peter Coad und Edward Yourdon sind nur einige Beispiele.

Diese Vielfalt erschwerte die Kommunikation zwischen Softwareentwicklern und machte die Entwicklung von CASE-Werkzeugen (Computer Aided Software Engineering) äußerst aufwendig.







Anfang der 90er-Jahre begannen Grady Booch und Jim Rumbaugh die Arbeit an einer vereinheitlichten Vorgehensmethode und Notation, damals noch unter dem Namen *Unified Method*.

1996 steuerte Ivar Jacobson OOSE bei, wonach zum ersten Mal die Bezeichnung UML (Unified Modeling Language) verwendet wurde. Ziel der Arbeit war es, die Stärken der vielen Notationen herauszukristallisieren und zu einer einzigen Modellierungssprache zu konsolidieren.

Schnell erkannten viele namhafte Unternehmen, z. B. Microsoft, Oracle, IBM oder Rational Software, die Vorteile einer einheitlichen Modellierungssprache und schlossen sich zu den *UML Partners* zusammen. Im Januar 1997 wurde die UML 1.0 als erste offizielle Version verabschiedet.

Um einen industrieweiten Standard zu schaffen, strebten die UML Partners eine Zertifizierung durch das Standardisierungsgremium OMG (Object Management Group) an. 1999 wurde die UML 1.3 zum ersten Mal von der OMG freigegeben.

Seitdem hat sich die UML ständig weiterentwickelt und weltweit in der Softwarebranche als Standard durchgesetzt. Die »drei Amigos« genannten Urväter Booch, Jacobson und Rumbaugh arbeiten nach wie vor an der Weiterentwicklung. Inzwischen sind jedoch beinahe alle bekannten Unternehmen der Softwarebranche in die Arbeit an der UML involviert und garantieren den hohen Standard und die Zukunftsfähigkeit der Modellierungssprache.

## 1.5 Von der UML 1.x zur UML 2.5

Bei der Betrachtung der Abbildung 1.1 ist Ihnen wahrscheinlich der Versionssprung von UML 1.5 auf 2.0 aufgefallen. Was hat die OMG bewogen, diesen großen Entwicklungsschritt durchzuführen?

Seit der Entwicklung von UML 1.0 hatten die nachfolgenden Versionen lediglich kleine »kosmetische« Änderungen erfahren. In der Zwischenzeit setzten sich neue Programmiersprachen (z. B. Java oder C#) immer weiter durch und verdrängten alte (z. B. C oder C++). Neue Anforderungen aufgrund immer weiterer Einsatzfelder der UML wurden sichtbar, z. B. der Ruf nach exakteren Modellierungsmöglichkeiten zeitlicher Aspekte. Alte Notationselemente entpuppten sich als zu nah an einer Programmiersprache entworfen (`friend`).

Durch das iterative Hinzufügen von Details wurde die UML zwar ständig mächtiger, jedoch immer weniger überschaubar und erlernbar. Die OMG hat dies erkannt und entschied sich, vollständig aufzuräumen und einen großen Schritt auf Version 2.0 zu machen. Man teilte die Definitionen der UML in zwei Dokumente auf, eines für die

Modellierung von Architektur, Profilen und Stereotypen (Infrastructure) und eines für statische und dynamische Modellelemente (Superstructure). Aus diesem Grund wurden einige Diagramme vollständig überarbeitet. Das Konzept hinter Aktivitätsdiagrammen (siehe Kapitel 9) wurde beispielsweise komplett verändert. Vollkommen neue Diagramme, z. B. das Timing-Diagramm (siehe Kapitel 13), wurden aufgenommen. Alte und bewährte Diagrammarten konnten fast ohne Modifikationen übernommen werden, beispielsweise die Anwendungsfalldiagramme (siehe Kapitel 8) oder die Klassendiagramme (siehe Kapitel 2). Die Definition der *Object Constraint Language* (OCL) und ein spezielles XML-Austauschformat wurden in weiteren Teilspezifikationen untergebracht. Mit der Version 2.2 der UML wurden weitere Profildiagramme (siehe Kapitel 15) in die UML-Spezifikation aufgenommen. Die UML 2.3 dehnte den Umfang der Spezifikation noch weiter aus, sodass die Spezifikation erneut aus den Fugen geriet. Viele Stimmen wurden laut, die die Größe und Komplexität der Spezifikation kritisierten. Deshalb versuchte man die Spezifikation zu vereinfachen. Dies gelang aber kaum. Stattdessen gelangten weitere Features in die neue Spezifikation. Beispielsweise führte die UML 2.4.1 URIs ein und ermöglichte ferner, dass Attribute als eindeutige Identifier gekennzeichnet werden können. Diese Neuerungen waren wichtig und mussten dringend hinzugefügt werden.

Für Version 2.5 entschied man sich, das Problem der exorbitanten Spezifikation aufs Neue anzugehen. Mit viel Aufwand wurde aus den zwei Teilen der Spezifikation ein neues, nur noch halb so umfangreiches und deutlich vereinfachtes Dokument. Dafür wurde auf vieles verzichtet, was sich im Laufe der Zeit als irrelevant erwiesen hatte. Beispielsweise definierte die UML vor 2.5 unterschiedliche Konformitätslevel L0, L1, L2 und L3, die ein CASE-Tool-Hersteller anstreben kann. Weil dies aber in der Praxis nicht genutzt wurde, besteht seit Version 2.5 nur noch ein einziger Konformitätslevel. Es ist zwar erlaubt, dass ein CASE-Tool nur eine Untermenge der Notationen oder der Metamodelle einsetzt, aber dies muss der Hersteller mit einem expliziten Hinweis anzeigen. Ansonsten ist ein CASE-Tool von jetzt an entweder UML-konform oder eben nicht. Darüber hinaus konnte sehr viel Text eingespart werden, indem alle redundanten Informationen ganz einfach entfernt wurden.

Insgesamt erschuf die OMG eine neue UML, die kürzer und konsistenter als all ihre Vorgänger geworden ist. Die inhaltlichen Änderungen der UML-Version 2.5 gegenüber der Vorgängerversion 2.4.1 sind hingegen marginal. So haben sich auch die Diagramme nur sehr wenig verändert. Für Sie als Anwender ist es daher von Bedeutung, dass die UML-Diagramme, die auf Basis älterer UML-Versionen modelliert wurden, nach wie vor valide sind.







## 1.6 Diagramme der UML 2.5

Im neu strukturierten Spezifikationsdokument der Version 2.5 fällt auf, dass es die UML-Elemente nicht UML-Diagrammen, sondern den Modellierungsarten zuordnet. Dabei werden grundsätzlich folgende Modellierungsarten unterschieden:

- Strukturmodellierung (*Structural Modeling*)
- Verhaltensmodellierung (*Behavioral Modeling*)
- Ergänzungsmodellierung (*Supplemental Modeling*)

Auf diese Weise verdeutlicht die OMG, dass die Benutzung vieler UML-Elemente nicht auf ein bestimmtes UML-Diagramm beschränkt ist.

Bei den Elementen der Strukturmodellierung handelt es sich beispielsweise um Klassen, Assoziationen, Pakete, Komponenten usw.

Bei der Verhaltensmodellierung werden die Elemente für die Zustandsmaschinen, die Aktivitäten und die Interaktionen gezeigt.

Zuletzt erfolgt die Ergänzungsmodellierung, die mithilfe von Anwendungsfällen, Deployments und Informationsflüssen zusätzliche Möglichkeiten bietet, um das System in den jeweiligen Projektphasen modellieren zu können.

Der Anhang der Spezifikation geht schließlich erneut auf eine Unterteilung von UML-Diagrammen ein. Allerdings taucht dort eine andere, nämlich die in der Praxis übliche Sortierung auf. Um in diesem Buch eine klare Linie vorzugeben, die auch in der alltäglichen Arbeit gängig ist, wurden die Kapitel dieses Buches dieser Sortierung entsprechend strukturiert.

An dieser Stelle finden Sie zunächst eine Übersicht der UML-Diagramme (siehe Abbildung 1.2), bevor deren Einzelheiten in den folgenden Kapiteln behandelt werden.

Zur Gruppe der **Strukturdiagramme** gehören:

- **Klassendiagramm**
  Ein Klassendiagramm (engl. *Class Diagram*) beinhaltet die statischen Strukturbestandteile eines Systems, deren Eigenschaften und Beziehungen. Es fungiert als eine Art allgemeiner Bauplan für Objekte (siehe Abbildung 1.3). Details zu Klassendiagrammen finden Sie in Kapitel 2.

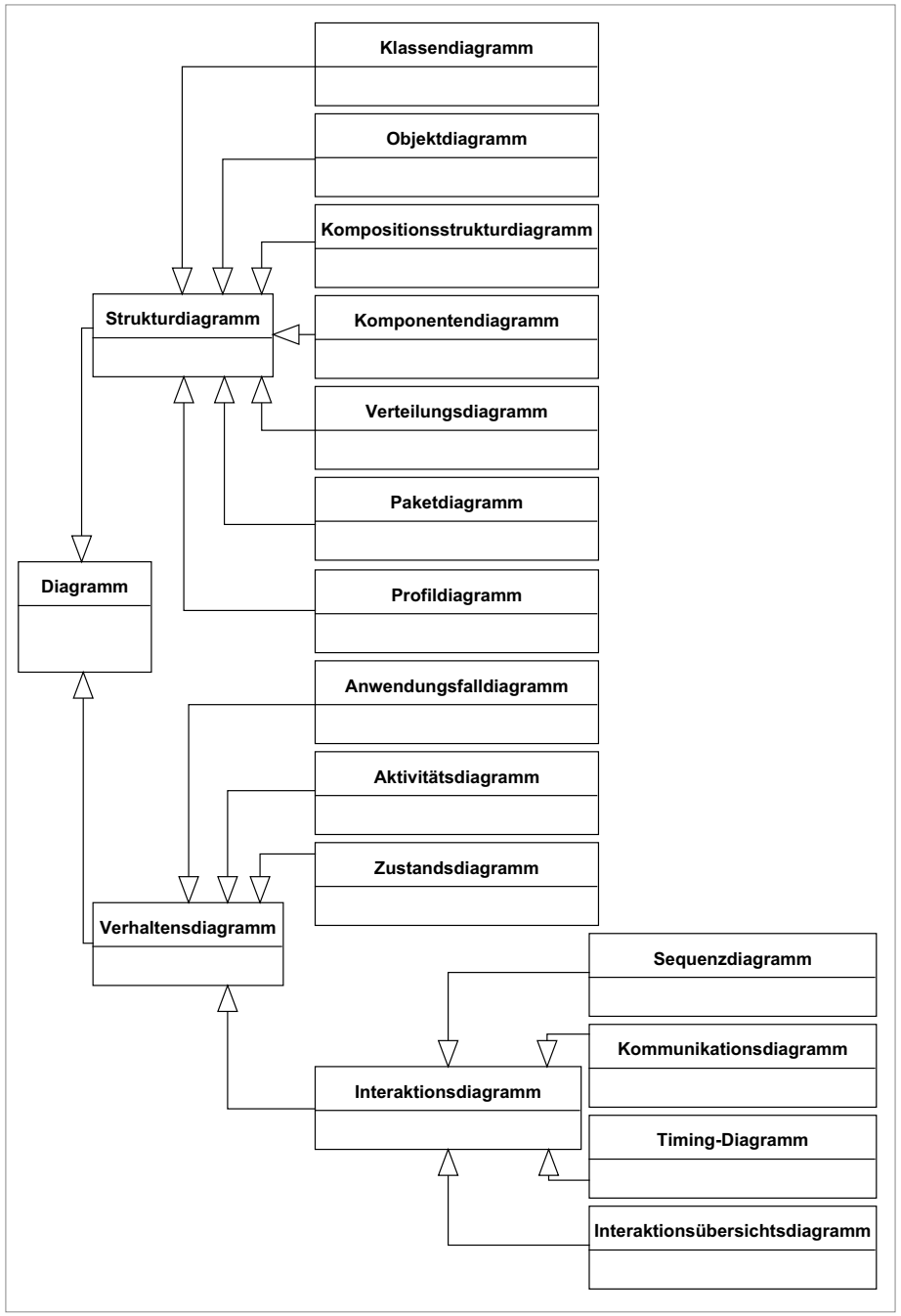


**Abbildung 1.2** Übersicht über die UML-Diagramme







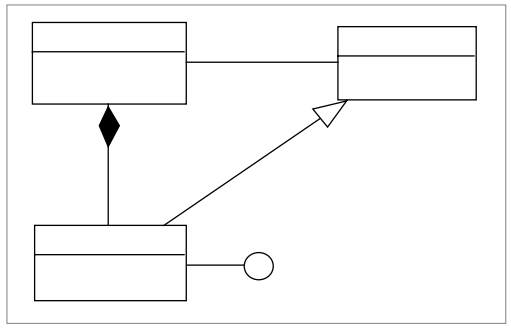

**Abbildung 1.3** Klassendiagramm

- **Objektdiagramm**
  Ein Objektdiagramm (engl. *Object Diagram*) stellt eine Momentaufnahme der Objekte eines Systems dar, die nach dem Bauplan eines Klassendiagramms gebildet wurden. Kapitel 3 behandelt alle wichtigen Aspekte von Objektdiagrammen.

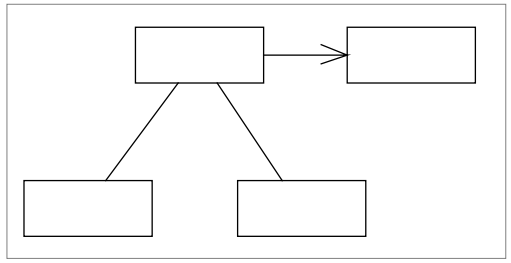

**Abbildung 1.4** Objektdiagramm

- **Kompositionsstrukturdiagramm**
  Ein Kompositionsstrukturdiagramm (engl. *Composite Structure Diagram*, siehe Kapitel 4) beschreibt die interne Struktur einer Komponente und deren Interaktionspunkte zu weiteren Komponenten des Systems.

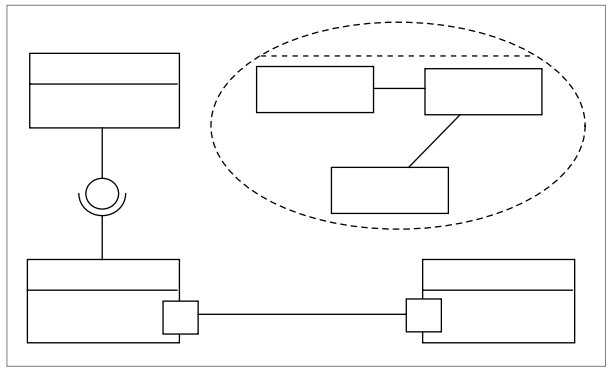

**Abbildung 1.5** Kompositionsstrukturdiagramm

- **Komponentendiagramm**
  Dieses Diagramm zeigt die Organisation und Abhängigkeiten der Komponenten. Komponentendiagramme (engl. *Component Diagrams*) sind das Thema von Kapitel 5.

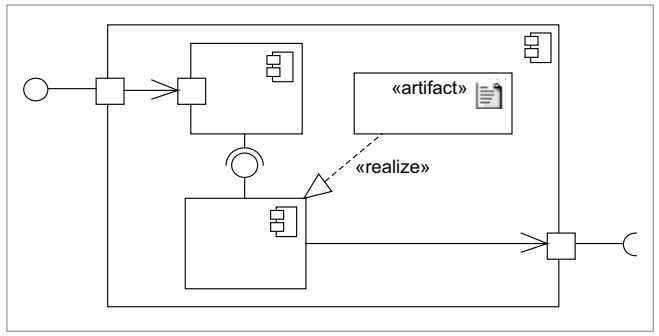


**Abbildung 1.6** Komponentendiagramm

- **Verteilungsdiagramm**
  Ein Verteilungsdiagramm (engl. *Deployment Diagram*, siehe Kapitel 6) definiert die Architektur eines verteilten Systems, wie sie zur Laufzeit vorgefunden wird. Die Definition beschränkt sich nicht auf Software-Umgebungen, sondern umfasst auch Hardware und Kommunikationswege.

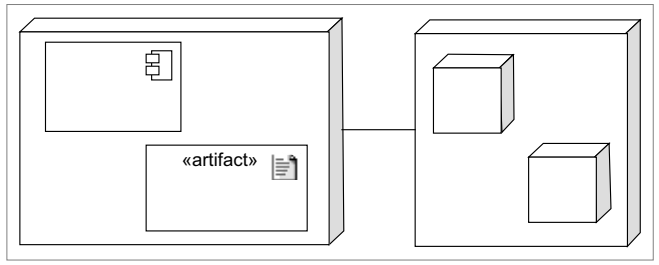


**Abbildung 1.7** Verteilungsdiagramm

- **Paketdiagramm**
  Das Thema von Kapitel 7 sind Paketdiagramme (engl. *Package Diagrams*), die UML-Elemente in Pakete organisieren.

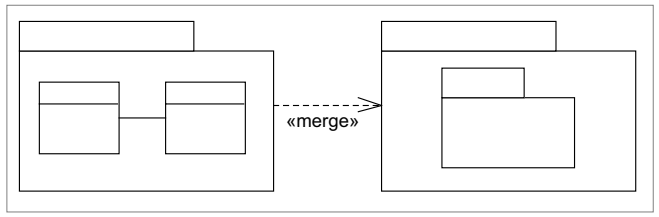


**Abbildung 1.8** Paketdiagramm







- **Profildiagramm**
  Profildiagramme (engl. *Profile Diagrams*) stellen einen leichtgewichtigen Mechanismus dar, mit dem die UML erweitert werden kann. Da sämtliche UML-Metaklassen erweitert werden können, werden Profildiagramme erst in Kapitel 15 behandelt, nachdem Sie alle wichtigen UML-Elemente kennengelernt haben.

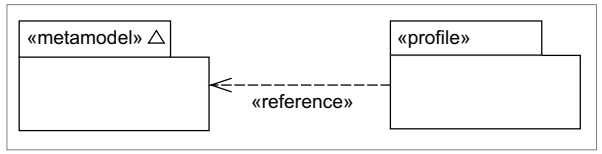


**Abbildung 1.9** Profildiagramm

**Verhaltensdiagramme** sind:

- **Anwendungsfalldiagramm**
  Ein Anwendungsfalldiagramm zeigt die Beziehungen zwischen Akteuren und den Anwendungsfällen. Anwendungsfälle stellen eine Menge von Aktionen dar, die ein Akteur während der Interaktion mit einem System abrufen kann. Umfangreiche Informationen zu Anwendungsfalldiagrammen (engl. *Use Case Diagrams*) finden Sie in Kapitel 8.

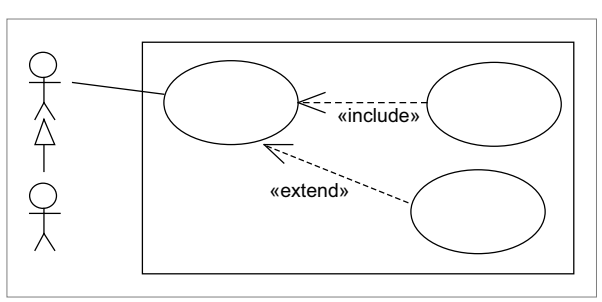


**Abbildung 1.10** Anwendungsfalldiagramm

- **Aktivitätsdiagramm**
  Aktivitätsdiagramme (engl. *Activity Diagrams*, siehe Kapitel 9) beschreiben das Verhalten einer Klasse oder Komponente. Sie bedienen sich dabei eines Kontroll- und Datenflussmodells.

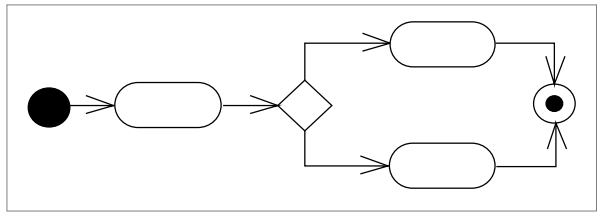

**Abbildung 1.11** Aktivitätsdiagramm

- **Zustandsdiagramm**
  Die möglichen Zustände, Zustandsübergänge, Ereignisse und Aktionen im »Leben« eines Systems werden in einem Zustandsdiagramm (engl. *State Machine Diagram*, siehe Kapitel 10) modelliert. Zustandsdiagramme basieren auf dem Konzept der deterministischen endlichen Automaten.

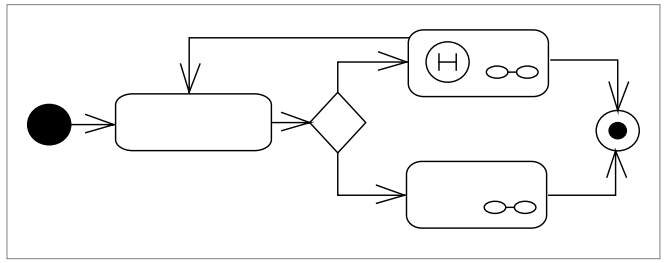


**Abbildung 1.12** Zustandsdiagramm

Die folgenden Verhaltensdiagramme werden unter der Bezeichnung **Interaktionsdiagramme** (engl. *Interaction Diagrams*) zusammengefasst:

- **Sequenzdiagramm**
  Sequenzdiagramme (engl. *Sequence Diagrams*, siehe Kapitel 11) definieren Interaktionen zwischen Objekten, wobei sie sich auf den Nachrichtenfluss konzentrieren. Sie zeigen den zeitlichen Ablauf der Nachrichten, beinhalten jedoch keine Informationen über Beziehungen der Objekte.

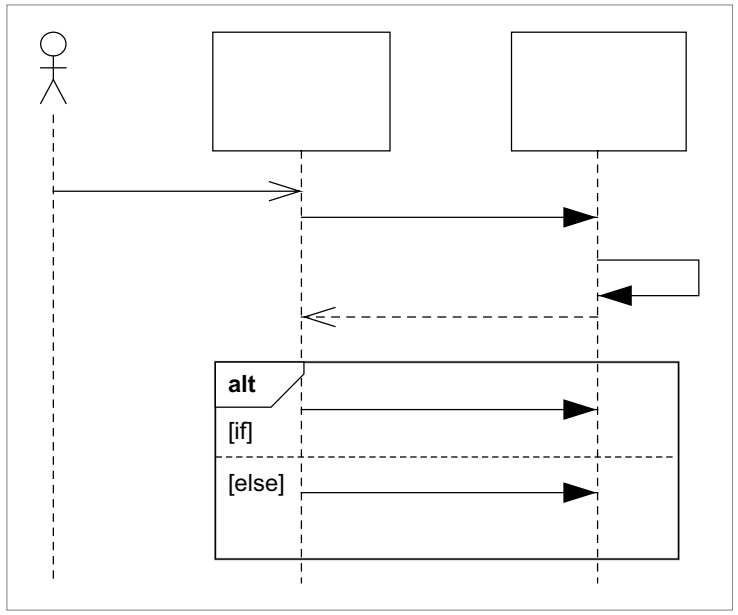


**Abbildung 1.13** Sequenzdiagramm







- **Kommunikationsdiagramm**
  Ein Kommunikationsdiagramm (engl. *Communication Diagram*, siehe Kapitel 12) beschreibt die Interaktion zwischen Objekten. Im Gegensatz zum Sequenzdiagramm setzt es den Fokus auf die Kommunikationsbeziehungen der Objekte während einer Interaktion.

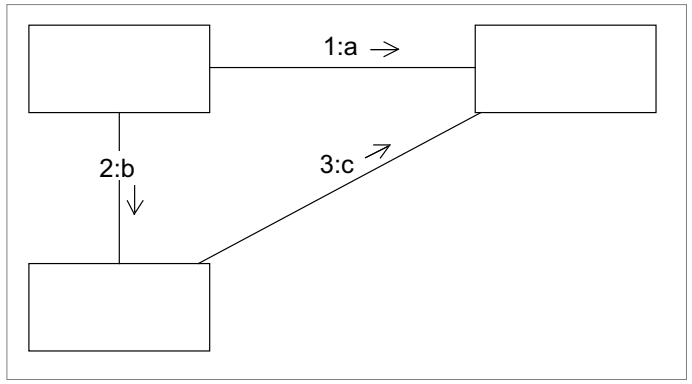


**Abbildung 1.14** Kommunikationsdiagramm

- **Timing-Diagramm**
  Timing-Diagramme zeigen die Zustandswechsel von Objekten innerhalb einer Zeitspanne als Antworten auf eintreffende Ereignisse. Kapitel 13 enthält alle Details zu Timing-Diagrammen (engl. *Timing Diagrams*).

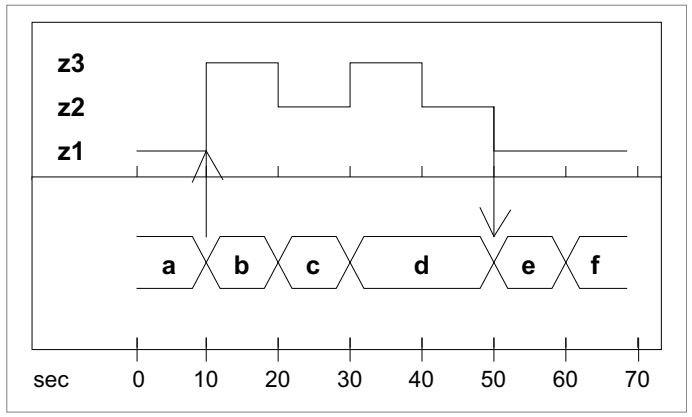


**Abbildung 1.15** Timing-Diagramm

- **Interaktionsübersichtsdiagramm**
  Beim Interaktionsübersichtsdiagramm handelt es sich um eine Diagrammart, die den Kontrollfluss zwischen Interaktionen mithilfe der Notationselemente von Aktivitätsdiagrammen beschreibt. Die einzelnen Kontrollflussknoten können vollständige Interaktionsdiagramme repräsentieren. Kapitel 14 befasst sich mit

den Feinheiten von Interaktionsübersichtsdiagrammen (engl. *Interaction Overview Diagrams*).

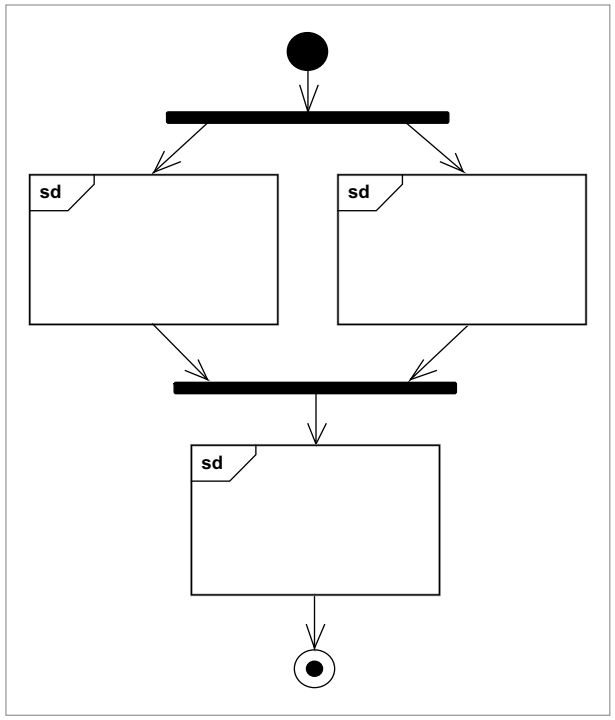


**Abbildung 1.16** Interaktionsübersichtsdiagramm

Nach dieser ersten Übersicht behandeln die folgenden Kapitel die einzelnen Diagramme und deren Notationselemente im Detail. Die gerade vorgestellte Gruppierung der Diagramme spiegelt sich dabei in der Reihenfolge der Kapitel wider.

## 1.7 Realisierung in Java und C#

Wir zeigen zu jedem Diagramm beispielhaft eine Realisierung in Java und in C#.

Für Codebeispiele in C#, die der Java-Version besonders ähnlich sind, haben wir angenommen, dass ein Abdruck im Buch keinen Vorteil bringt, und bieten sie ausschließlich zum Download an.

**Hinweis: Codebeispiele herunterladen**

Alle Codebeispiele zum Buch finden Sie unter *www.rheinwerk-verlag.de/4546/*. Scrollen Sie etwas herunter bis zu den »Materialien zum Buch«.







Die Programmiersprachen, auch die objektorientierten, sind unterschiedlich mit Sprachmitteln ausgestattet, was mal eine sehr direkte Umsetzung ermöglicht, mal ein etwas kreativeres Design erfordert. So bietet Java z. B. keine Möglichkeit, für die Parameter eines Konstruktors Vorgabewerte zu definieren, wie es das Objektdiagramm in Abschnitt 3.3.1 modelliert. Einen Weg, dies zu lösen, zeigen und erläutern wir im Listing zum Diagramm.

Natürlich gibt es nicht *die* eine Lösung bei der Implementierung eines Modells.

In der Programmiersprache C# in der Version 7, wie sie hier verwendet wird, gibt es zum Beispiel mehrere Möglichkeiten, Attribute und Methoden zu definieren. Hier wurde dem Compiler viel sogenannter *Syntactic Sugar* mitgegeben, der es dem Entwickler erlaubt, ohne viel Aufwand und ohne unübersichtlichen Code zu erzeugen, Vereinfachungen zu nutzen.

Bei den Beispielen in C#, die diesem Buch beiliegen, wurden diese Möglichkeiten genutzt, insofern es für die Programmierpraxis sinnvoll erschien. Damit weicht der Code zuweilen von den UML-Diagrammen im strikten Sinne ab.

Als Beispiel seien hier die automatischen Properties genannt, die es erlauben, ohne öffentliche Attribute auszukommen, sodass Sie den Code später ändern können, ohne dass abhängige Assemblies neu kompiliert werden müssten. Dadurch sind die Beispiele näher an dem, was heute in der professionellen Softwareentwicklung in C# verwendet wird.

Außerdem folgen die C#-Beispiele bei der Groß- und Kleinschreibung den Konventionen von Microsoft. Zwar weichen diese von den Konventionen in Java und von denen in den UML-Diagrammen ab (die denen von Java folgen), das führt aber aller Voraussicht nach nicht zu Missverständnissen.

Wenn es für den Leser nötig wird, dass die UML-Diagramme exakt mit dem Code übereinstimmen, sei hier auf die Möglichkeit der »Stereotypen« hingewiesen, mit denen z. B. Methoden als »csharp properties« markiert werden können.

# Kapitel 3
# Objektdiagramm

*Das Objektdiagramm zeigt eine Momentaufnahme der Objekte eines Systems.*

## 3.1 Anwendungsbereiche

Ein Objektdiagramm (engl. *Object Diagram*) kann als Sonderfall eines Klassendiagramms angesehen werden.

Während ein Klassendiagramm die allgemeinen Baupläne und alle möglichen Beziehungen der Objekte untereinander modelliert, stellt das zugehörige Objektdiagramm die tatsächlich erzeugten Objekte, deren Attributwerte und Beziehungen innerhalb eines begrenzten Zeitraums zur Laufzeit dar.

Aus diesem Grund werden Objektdiagramme in allen Phasen der Softwareentwicklung parallel zu Klassendiagrammen eingesetzt. Ihre Verwendung wird zumeist jedoch nur notwendig, wenn komplexe Klassendiagramme anhand von Beispielen verdeutlicht und überprüft werden sollen.

## 3.2 Übersicht

Abbildung 3.1 präsentiert die wichtigsten Notationselemente von Objektdiagrammen.

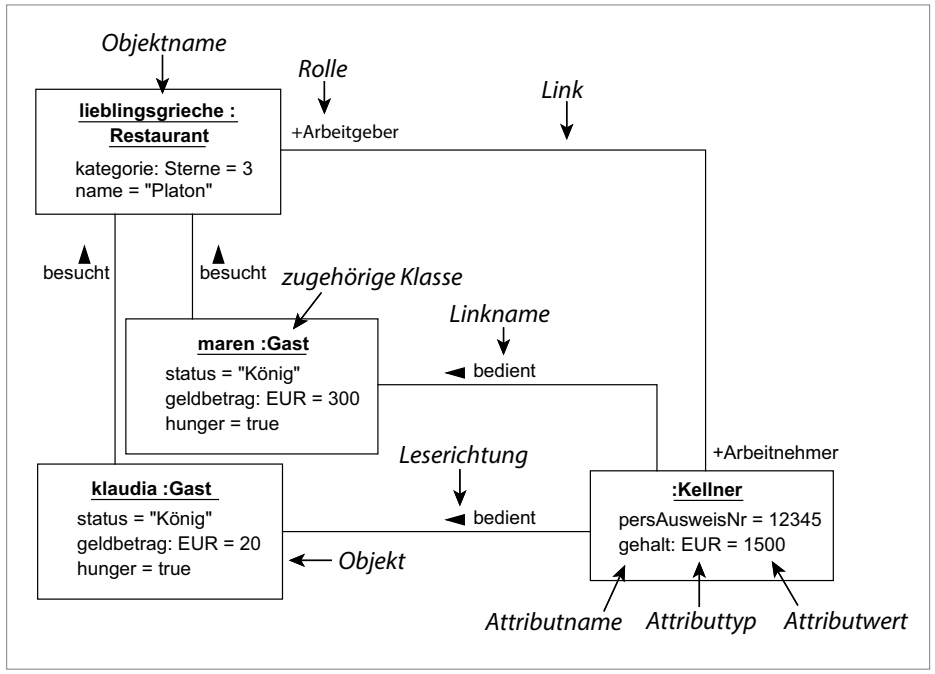


**Abbildung 3.1** Notationselemente von Objektdiagrammen

## 3.3 Notationselemente

### 3.3.1 Objekt

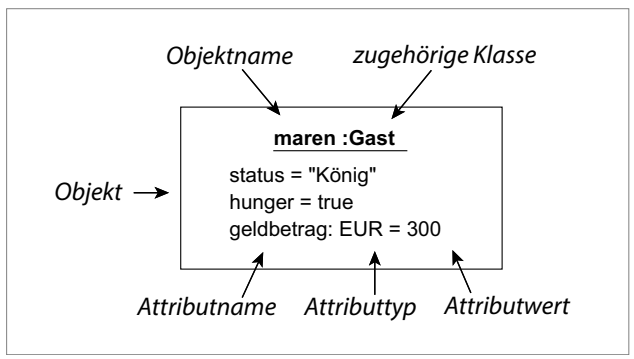


**Abbildung 3.2** Objekt

#### Beschreibung

Ein **Objekt** (engl. *Object*) entsteht bei der Realisierung eines Bauplans, den eine Klasse spezifiziert. Es wird auch als **Instanz** oder **Exemplar einer Klasse** bezeichnet.







Abbildung 3.2 zeigt ein Objekt `maren` und dessen Attribute mit Attributwerten, die ihren Zustand festlegen. Im Unterschied zum Notationselement einer Klasse werden der Objektname und die Klassenzugehörigkeit unterstrichen dargestellt. Auf die Darstellung von Operationen wird bei Objekten verzichtet.

Das Objekt `maren` der Klasse `Gast` wird zu einem Zeitpunkt seines Lebens gezeigt, in dem seine Attribute `status` den Wert `"König"`, `geldbetrag` vom Typ `EUR` den Wert `300` und `hunger` den Wert `true` aufweisen. Anschaulich ausgedrückt, besitzt der `Gast` `maren` `300 EUR`, ist `hungrig` und wird wie ein `König` behandelt. Die Attributwerte müssen erwartungsgemäß zu den Typen der Attribute passen.

Das Objektdiagramm beschreibt nicht, wie `maren` in diesen Zustand gekommen ist oder welches ihr nächster Zustand ist. Die Modellierung aller Zustände und Zustandsübergänge ist Aufgabe des Zustandsdiagramms, das Sie in Kapitel 10 kennenlernen.

Die Angabe der Attribute und Attributwerte eines Objektes kann unvollständig sein und nur diejenigen umfassen, die gerade für seinen Zustand bedeutend sind.

Abbildung 3.3 zeigt ein Objekt der Klasse `Gast` mit dem Namen `maren` während der Erteilung einer Bestellung.

Zu diesem Zeitpunkt hatte `maren` demnach `300 EUR` und den `status` `"König"`. Das Attribut `hunger` wird ausgeblendet, da es bei der Erteilung einer Bestellung nicht zwingend eine Rolle spielen muss.

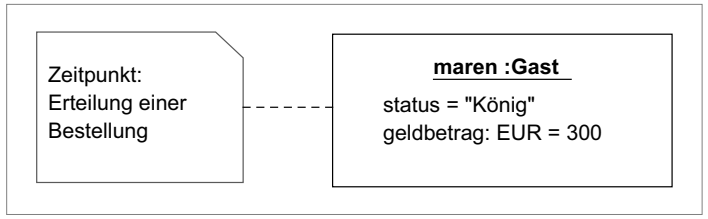


**Abbildung 3.3** Unvollständige, aber zulässige Objektspezifikation

Die Instanziierung einer Klasse durch ein Objekt kann auch mithilfe der `<<instantiate>>`-Abhängigkeit modelliert werden:

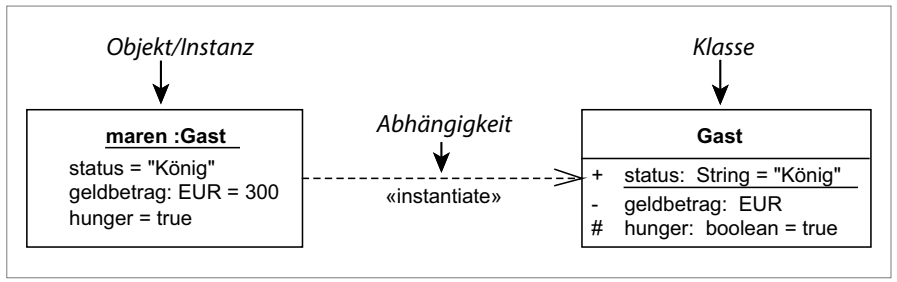


**Abbildung 3.4** Instanziierung einer Klasse

Abbildung 3.4 zeigt, dass das Objekt `maren` eine Instanz der Klasse `Gast` darstellt. Man erkennt dies zwar auch an der Klassenbezeichnung hinter dem Doppelpunkt im Objektnamen. Die Darstellung aus Abbildung 3.4 hebt jedoch den Zusammenhang zwischen Objekt und Klasse hervor und zeigt den »Bauplan« und das »Produkt« in einem Diagramm.

### Verwendung

Objektdiagramme sind hilfreich, um den konkreten Zustand einzelner Objekte in einem gegebenen Kontext darzustellen. Sie stellen im Allgemeinen die Werte der Attribute dar, die diesen Zustand charakterisieren. Alle anderen Attribute werden aus Gründen der Überschaubarkeit weggelassen.

Bei der Darstellung der Objekte können Klassendiagramme auf deren Vollständigkeit und Korrektheit überprüft werden. Benötigen Sie in einem der Objektdiagramme ein Objekt, das keiner der Klassen zugeordnet werden kann, ist Ihr Klassendiagramm unvollständig. Klassen dagegen, nach deren Bauplan kein Objekt erzeugt werden musste, können eventuell aus dem Klassendiagramm entfernt werden.

### Realisierung in Java

Zur Implementierung wird das Beispiel aus Abbildung 3.4 herangezogen.

Zunächst wird eine Klasse `EUR` erstellt, die den Datentyp für `geldbetrag` realisiert. Sie kapselt lediglich eine Fließkommazahl (`float`):

```
class EUR
{
  public float betrag;
  public EUR(float b)
  {
    betrag = b;
  }
}
```

**Listing 3.1** /beispiele/java/kap3/kap_3_3_1/EUR.java
(Download der Beispiele: www.rheinwerk-verlag.de/4546)

Nun kann die Klasse `Gast` implementiert werden:

```
class Gast
{
  public static String status = "König";

  private EUR geldbetrag; ❶
  protected boolean hunger;
```







```
  public Gast(EUR g, boolean h) ❷
  {
    geldbetrag = g;
    hunger = h;
  }

  public Gast(EUR g) ❷
  {
    geldbetrag = g;
    hunger = true;
  }
}
```

**Listing 3.2** /beispiele/java/kap3/kap_3_3_1/Gast.java

❶ Der `geldbetrag` erhält den geforderten Typ `EUR`.

❷ Java bietet keine Möglichkeit, Vorgabewerte für Übergabeparameter zu definieren, was für das Attribut `hunger` notwendig wäre. Daher bedient sich dieses Beispiel der Überladung von Konstruktoren und emuliert damit eine Art Vorgabewert.

Eine einfache Hauptoperation demonstriert die Instanziierung eines Objekts der Klasse `Gast`:

```
public static void main(String[] args)
{

  Gast maren = new Gast(new EUR(300), true); ❶
}
```

**Listing 3.3** /beispiele/java/kap3/kap_3_3_1/Test.java

❶ Das in Abbildung 3.4 gezeigte Objekt `maren` mit einem `geldbetrag` von `300 EUR` und `hunger = true` wird mithilfe des `new`-Operators instanziiert.

Das Attribut `status` muss nicht gesetzt werden, da es als Klassenattribut für alle Objekte der Klasse `Gast` verfügbar ist und bereits bei seiner Deklaration mit `"König"` vorbelegt wird.

### Realisierung in C#

Die Umsetzung und Instanziierung der Klassen `EUR` und `Gast` unterscheiden sich in C# nicht wesentlich von der in Java:

```
public class EUR
{
    private decimal betrag; ❶
```

```
    public EUR(decimal betrag) => this.betrag = betrag; ❷
}

public class Gast
{
    public static string status = "König";
    private EUR geldbetrag;
    protected bool hunger; ❸

    public Gast(EUR geldbetrag, bool hunger = true) ❹
    {
        this.geldbetrag = geldbetrag; ❺
        this.hunger = hunger;
    }
}
```

**Listing 3.4** /beispiele/c#/kap3/kap_3_3_1/Kap_3_3_1.cs

❶ Der Datentyp `decimal` ist für Geldbeträge besser geeignet als `float` oder `double`.

❷ Das Schlüsselwort `this` muss verwendet werden, wenn wie hier der Parametername gleich dem Namen des Attributs ist.

❸ In C# muss statt `boolean` der Datentyp `bool` verwendet werden.

❹ In C# können Vorgabewerte für Parameter angegeben werden.

Dadurch ist es möglich, die Anzahl der Konstruktoren zu verringern und die Lesbarkeit zu erhöhen.

Wird im Beispiel der Konstruktor mit

```
Gast maren = new Gast(new EUR(300))
```

aufgerufen, also ohne einen Wert für den Parameter `hunger` anzugeben, dann wird der Vorgabewert `true` verwendet.

❺ Wieder muss das Schlüsselwort `this` verwendet werden, wenn wie hier der Parametername gleich dem Namen des Attributs ist.

```
static void Main(string[] args)
{
  Gast maren = new Gast(new EUR(300), true); ❶
}
```

**Listing 3.5** /beispiele/c#/kap3/kap_3_3_1/Kap_3_3_1.cs

❶ Das in Abbildung 3.4 dargestellte Objekt `maren` der Klasse `Gast` wird mit einem `geldbetrag` von `300 EUR` und `hunger = true` angelegt, was in C# ebenfalls mit dem `new`-Operator durchgeführt wird.





Wie oben erklärt, kann der Parameter `true` hier auch weggelassen werden, da er dem Vorgabewert für den Parameter `hunger` entspricht.



### 3.3.2 Link

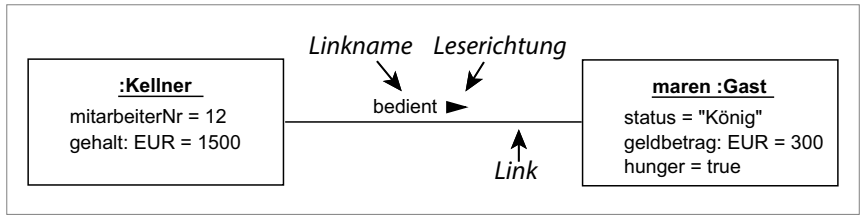


**Abbildung 3.5** Link

#### Beschreibung

Ein **Link** repräsentiert eine **Beziehung zwischen zwei Objekten**.

In einem Klassendiagramm werden Beziehungen zwischen Klassen als *Assoziationen* bezeichnet. In einem Objektdiagramm ist ein Link die konkrete Ausprägung einer Assoziation.

Grafisch sind Links und Assoziationen nicht unterscheidbar. Durch ihre Verwendung zwischen Klassen bzw. Objekten sind sie jedoch nicht zu verwechseln.

Links erhalten alle Aspekte ihrer zugehörigen Assoziation, wie z. B. Rollen, Namen, Leserichtungen oder Eigenschaften. Allein eine Multiplizität höher als `1` ist nicht erlaubt, weil Links immer genau zwei Objekte verbinden. Hat eines der instanziierten Assoziationsenden eine Multiplizität größer als `1`, können mehrere Links zu mehreren Objekten modelliert werden.

#### Verwendung

Genauso wie ein Klassendiagramm ohne Assoziationen nicht viel mehr als eine Anhäufung von beziehungslosen Klassen ist, stellt ein Objektdiagramm ohne Links eine Anhäufung von beziehungslosen Objekten dar. Verbinden Sie daher Objekte mit Links auf dieselbe Art, wie Sie Klassen mit Assoziationen verbinden.

Sollte Ihnen während der Erstellung eines Objektdiagramms auffallen, dass ein Link benötigt wird, dem keine entsprechende Assoziation im Klassendiagramm gegenübersteht, ist Ihr Klassendiagramm unvollständig und muss erweitert werden. Assoziationen aus Klassendiagrammen, die in keinem der Objektdiagramme verwendet wurden, sind darauf zu überprüfen, ob sie im Klassendiagramm wirklich benötigt werden.

### Realisierung in Java

Das Objektdiagramm aus Abbildung 3.6 zeigt im oberen Teil die Klassen `Kellner` und `Gast`, die durch eine Assoziation verbunden sind.

Der untere Teil der Abbildung beinhaltet die Objekte und einen Link, die diese beiden Klassen sowie die Assoziation instanziieren.

Dieses Objektdiagramm soll im Folgenden umgesetzt werden.

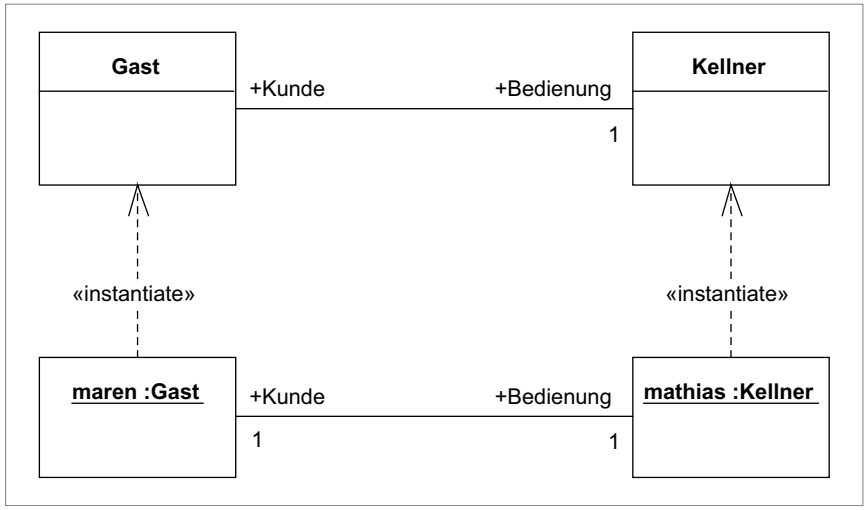


**Abbildung 3.6** Link-Beispiel

Zunächst werden die Klassendefinitionen realisiert:

```
class Kellner
{

  public Gast kunde; ❶
  public void setGast(Gast g) ❷
  {
    kunde = g;
    g.bedienung = this;
  }
}
class Gast
{

  public Kellner bedienung; ❸
  public void setKellner(Kellner k)
  {
    bedienung = k;
    k.kunde = this;
  }
}
```

**Listing 3.6** /beispiele/java/kap3/kap_3_3_2/Kellner.java & Gast.java

❶ Die Assoziation zum `Gast` wird deklariert.

❷ Da die Assoziation in beide Richtungen navigierbar ist, werden beide Assoziationsenden gleichzeitig instanziiert: sowohl auf der Seite des `Kellners` als auch auf der Seite des `Gastes`.

Erst damit ist die Assoziation vollständig und konsistent instanziiert: Ein Link entsteht.

❸ Bei der Klasse `Kunde` wird auf dieselbe Art verfahren.

In einer Hauptoperation werden die Objekte instanziiert und die Links gesetzt:

```
public static void main(String[] args)
{

  Gast maren = new Gast(); ❶
  Kellner mathias = new Kellner();
  maren.setKellner(mathias); ❷
}
```

**Listing 3.7** /beispiele/java/kap3/kap_3_3_2/Test.java

❶ Objekte werden in Java mithilfe des `new`-Operators instanziiert.

❷ Die gegenseitigen Assoziationen werden gesetzt und damit zu Links instanziiert. Der Aufruf einer der beiden `set`-Operationen ist ausreichend, da jede von ihnen beide Assoziationsenden konsistent setzt und somit den gesamten Link instanziiert.

### Realisierung in C#

Objekte werden in C# ebenfalls mit dem `new`-Operator instanziiert, sodass die Implementierung gegenüber dem Java-Programmcode keine erwähnenswerten Unterschiede aufweist. Sie finden den vollständigen Programmcode im Ordner *beispiele/c#/kap3/kap_3_3_2*.

## 3.4 Lesen eines Objektdiagramms

Ein Objektdiagramm wird üblicherweise verwendet, um ein Klassendiagramm zu illustrieren. Aus diesem Grund wird zunächst ein Klassendiagramm (siehe Abbildung 3.7) entworfen.

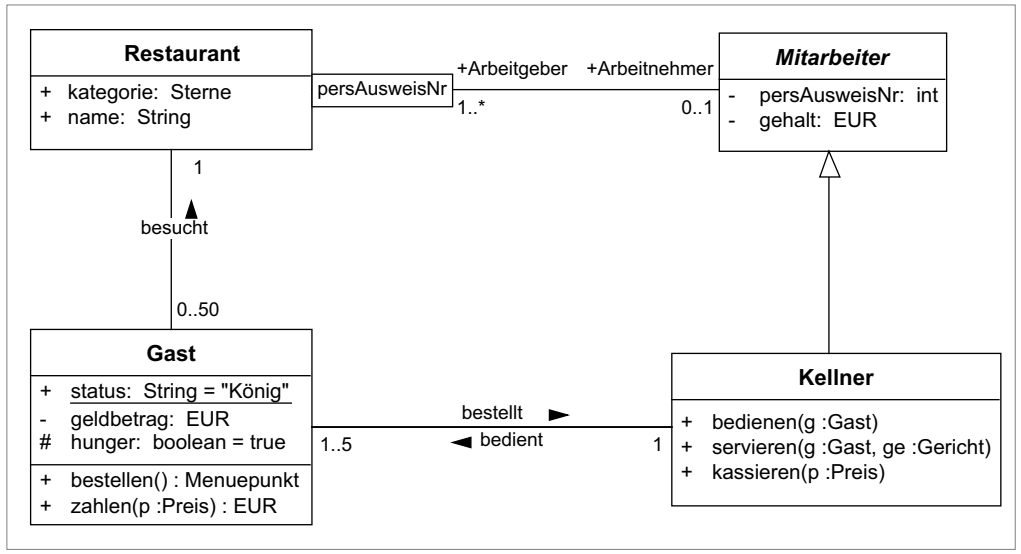


**Abbildung 3.7** Klassendiagramm als Grundlage für ein Objektdiagramm

Für das Klassendiagramm aus Abbildung 3.7 wird ein Objektdiagramm (siehe Abbildung 3.8) erstellt.

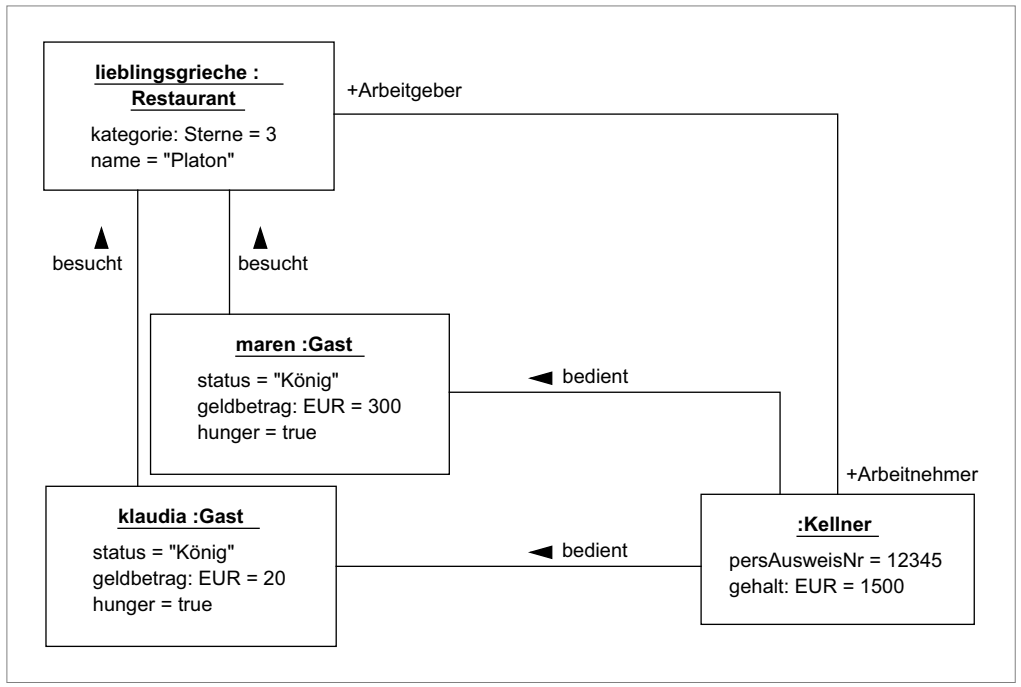


**Abbildung 3.8** Beispiel-Objektdiagramm







Das Objektdiagramm aus Abbildung 3.8 zeigt vier Objekte:

- `lieblingsgrieche` der Klasse `Restaurant`
- namenloser `Kellner`
- `maren` der Klasse `Gast`
- `klaudia` der Klasse `Gast`

Zu einem nicht näher spezifizierten Zeitpunkt besitzt der `lieblingsgrieche` den `namen` `"Platon"` und die `kategorie` `3` `Sterne`.

Als `Arbeitgeber` hat er einen Link zu seinem `Arbeitnehmer`. Zurzeit handelt es sich dabei um einen einzigen `Kellner`, dessen Objektname nicht spezifiziert wurde, weil er hier als nicht wichtig angesehen wird.

Beachten Sie, dass kein Objekt der Klasse `Mitarbeiter` im Objektdiagramm aufgeführt wird. Wie in Abschnitt 2.3.14 erläutert wurde, dienen abstrakte Klassen als Vorlagen für weitere Klassen und können nicht selbst instanziiert werden. Bei der Klasse `Mitarbeiter` handelt es sich um solch eine abstrakte Klasse. Sie wird von der Klasse `Kellner` spezialisiert, weshalb das namenlose `Kellner`-Objekt über alle Attribute eines `Mitarbeiters` verfügt und sie mit den Werten `12345` (`persAusweisNr`) und `1500` (`gehalt`) belegen kann.

Das Klassendiagramm aus Abbildung 3.7 definiert, dass ein `Kellner` `1` bis `5` `Gäste` bedienen kann. Im Objektdiagramm kann daher ein unbekannter `Kellner` zwei `Gäste` (`maren` und `klaudia`) bedienen, mit denen er über zwei Links verbunden ist.

`maren` und `klaudia` sind zwei Objekte der Klasse `Gast` und besitzen damit alle Attribute eines `Gastes`. Beide Objekte besuchen gerade dasselbe `Restaurant` und sind daher mit dem `lieblingsgriechen` über Links verbunden.

Bei den `Gast`-Objekten sollte zusätzlich erwähnt werden, dass jedes der `Gast`-Objekte das Attribut `status` mit dem Wert `"König"` besitzt, da es in der Klasse als statisches Attribut deklariert und mit ebendiesem Wert vorbelegt wird (statische Attribute wurden bereits in Abschnitt 2.3.2 behandelt).

## 3.5 Irrungen und Wirrungen

Abbildung 3.9 zeigt ein fehlerhaftes Objektdiagramm, das ein Beispiel des Klassendiagramms aus Abbildung 3.7 sein *sollte*:

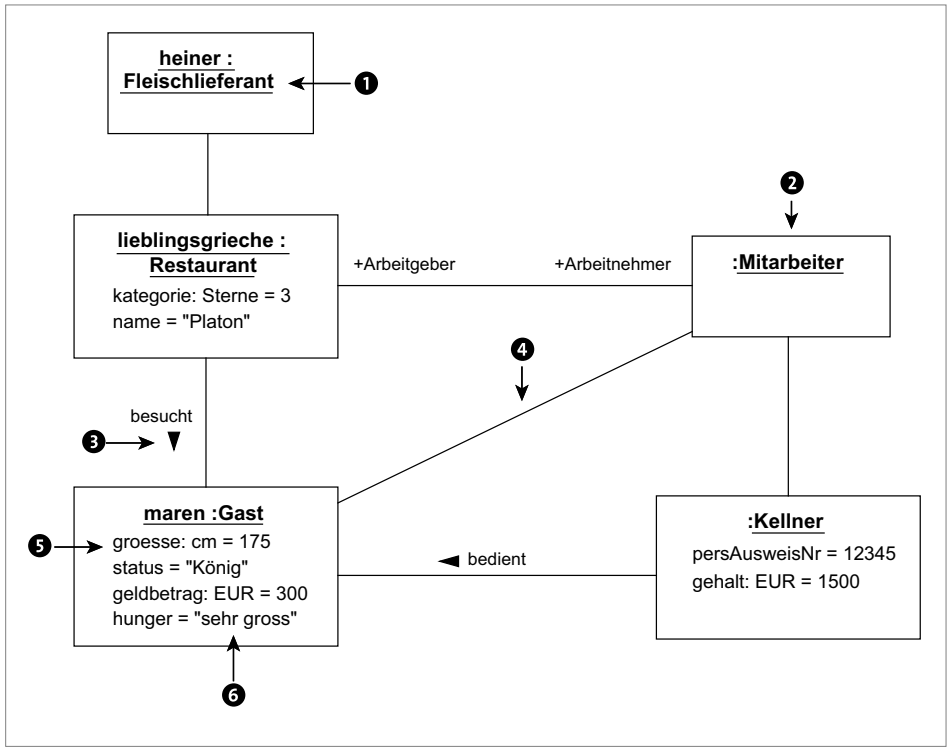


**Abbildung 3.9** Mögliche Fehler in Objektdiagrammen

❶ **Unbekannte Klasse**
Es dürfen nur Objekte erstellt werden, die aus Klassen des zugrunde liegenden Klassendiagramms erzeugt werden können.

Werden Objekte neuer Klassen benötigt, ist das Klassendiagramm unvollständig und muss erweitert werden.

❷ **Instanz einer abstrakten Klasse**
Aus abstrakten Klassen können keine Instanzen erzeugt werden.

❸ **Falsche Leserichtung**
Die Links eines Objektdiagramms müssen den Assoziationen des Klassendiagramms in den Assoziationsnamen, Leserichtungen, Rollen, Navigationsrichtungen und Eigenschaften entsprechen und dürfen sie nicht verändern.

Zeichnet sich bei der Erstellung eines Objektdiagramms ab, dass eine andere Assoziation benötigt wird, muss das Klassendiagramm modifiziert werden.

❹ **Neue Assoziation**
Ein Objektdiagramm darf keine Links beinhalten, die nicht im Klassendiagramm als Assoziationen vorhanden sind. Andererseits sollten Assoziationen aus dem zugehörigen Klassendiagramm auch als Links im Objektdiagramm nicht fehlen.







Macht das Objektdiagramm deutlich, dass ein zusätzlicher Link notwendig ist, muss das Klassendiagramm um die entsprechende Assoziation ergänzt werden.

❺ **Neues Attribut**
Objekte dürfen keine Attribute mit Werten belegen, die nicht bereits in der Klasse deklariert sind. Das Objekt würde andernfalls die Definition der Klasse und damit seinen eigenen Bauplan verändern.

Neue benötigte Attribute müssen zunächst in der Klasse hinzugefügt werden.

❻ **Attributwert falsch**
Die Attributwerte der Objekte müssen dem Datentyp der Attribute entsprechen. Attribut `hunger` ist vom Typ `boolean` und erlaubt damit nur die Werte `true` und `false`.

## 3.6 Zusammenfassung

Die wichtigsten Notationselemente von Objektdiagrammen und deren Bedeutung werden im Folgenden noch einmal rekapituliert:

- Ein **Objekt** wird auch als Instanz oder Ausprägung einer Klasse bezeichnet und entsteht als Produkt der Realisierung eines Klassen-Bauplans.

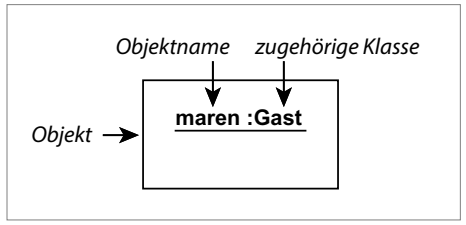


**Abbildung 3.10** Objekt

- **Links** zeigen Beziehungen zwischen Objekten.

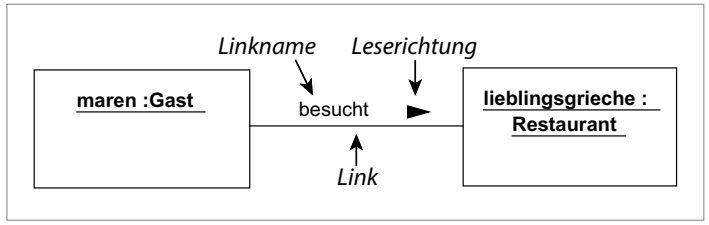


**Abbildung 3.11** Link

# Kapitel 7
# Paketdiagramm

*Paketdiagramme organisieren und strukturieren Elemente in Paketen.*



## 7.1 Anwendungsbereiche

Paketdiagramme (engl. *Package Diagrams*) werden zumeist in den frühen Phasen der Softwareentwicklung (wie Analyse/Definition und Entwurf/Design) verwendet, um das Modell sowohl horizontal als auch vertikal zu strukturieren.

Mit der horizontalen Strukturierung wird die Möglichkeit bezeichnet, beliebige UML-Elemente, die logisch zusammengehören, in Paketen zusammenzufassen und damit das UML-Modell zu modularisieren.

Pakete können Unterpakete enthalten und erlauben damit eine vertikale Strukturierung. Das Paket auf der obersten Ebene kann beispielsweise das gesamte Projekt repräsentieren, während die Pakete auf den tieferen Ebenen sich den Projektdetails nähern.

Mithilfe der vertikalen Strukturierung werden unterschiedliche Abstraktionsebenen eines Modells definiert, und es wird die Möglichkeit geschaffen, aus einer übersichtsartigen Darstellung schrittweise in die Details zu zoomen.

Strukturieren Sie Ihr Modell sowohl horizontal als auch vertikal, um es möglichst überschaubar und damit verständlich zu gestalten.

## 7.2 Übersicht

In Abbildung 7.1 sehen Sie die wichtigsten Notationselemente von Paketdiagrammen.

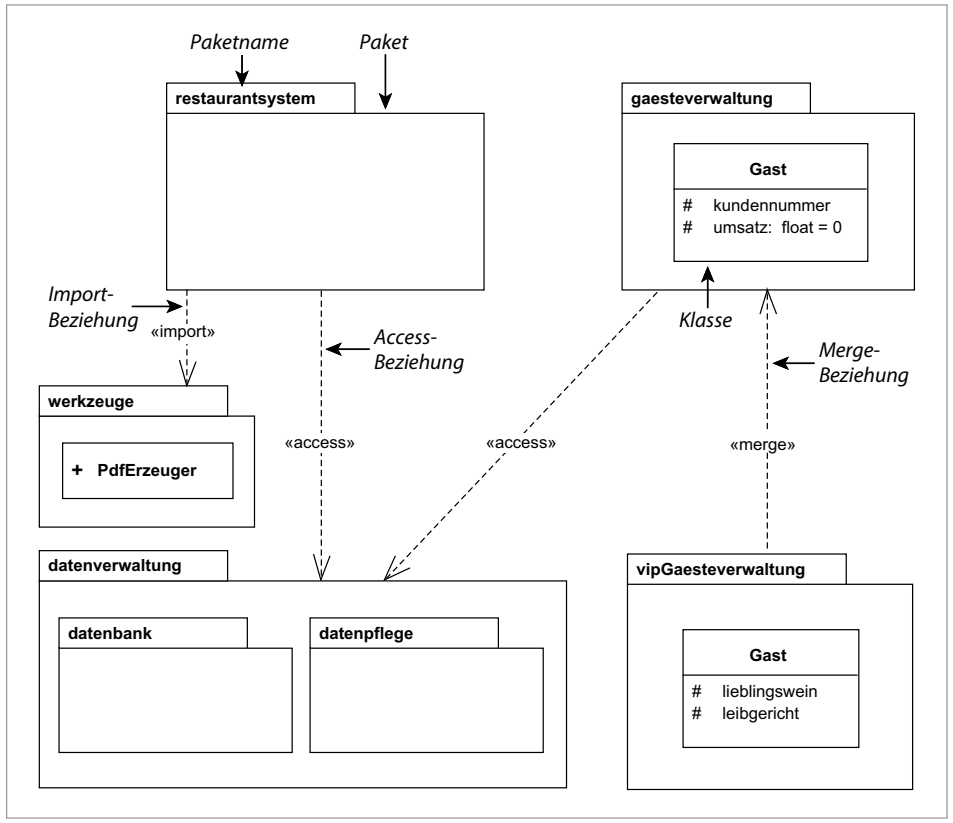


**Abbildung 7.1** Notationselemente von Paketdiagrammen

## 7.3 Notationselemente

### 7.3.1 Paket

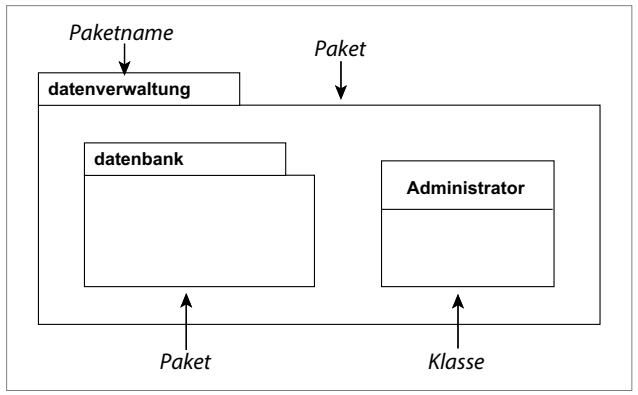


**Abbildung 7.2** Paket





#### Beschreibung

**Pakete** (engl. *Packages*) **gruppieren** Elemente und **definieren Namensräume** (engl. *Namespaces*), in denen sich diese Elemente befinden.

Abbildung 7.2 zeigt ein Paket `datenverwaltung`, das ein Unterpaket `datenbank` und eine Klasse `Administrator` enthält und damit Elemente gruppiert, die mit der Datenverwaltung im Zusammenhang stehen.

Die UML stellt hierfür eine weitere Notationsmöglichkeit zur Verfügung (siehe Abbildung 7.3).



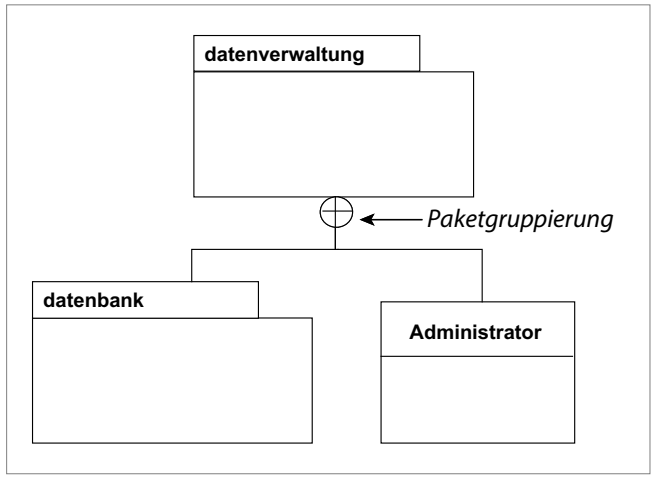


**Abbildung 7.3** Gruppierung von Elementen in einem Paket

Die Diagramme aus Abbildung 7.2 und Abbildung 7.3 sind semantisch gleich.

Alle Elemente innerhalb eines durch ein Paket definierten Namensraumes müssen unterschiedliche Namen besitzen. Innerhalb von unterschiedlichen Paketen ist es jedoch durchaus möglich, zwei Elemente mit demselben Namen zu definieren (siehe Abbildung 7.4).

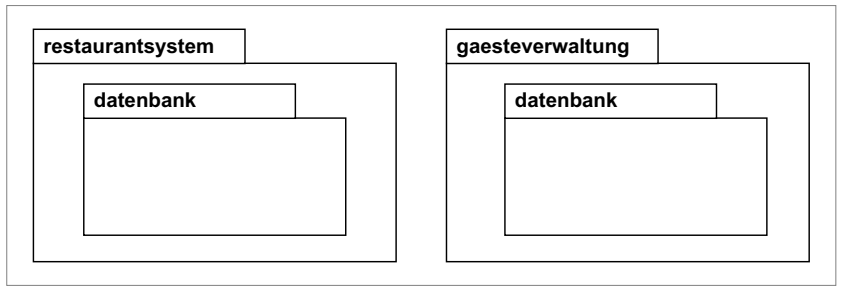


**Abbildung 7.4** Pakete definieren Namensräume.

Die Namen der Unterpakete `datenbank` in Abbildung 7.4 verursachen trotz ihrer Gleichheit keinen Namenskonflikt, weil sie in unterschiedlichen Paketen und damit in unterschiedlichen Namensräumen verwendet werden.

Außerhalb ihrer Pakete können sie jedoch nicht mehr über ihren **unqualifizierten Namen** `datenbank` angesprochen werden, weil dieser sie nicht eindeutig identifiziert. UML definiert hierfür die **qualifizierten Namen**, in denen zusätzlich die Paketnamen getrennt durch zwei Doppelpunkte angegeben werden müssen. Für die in Abbildung 7.4 modellierten Unterpakete lauten sie beispielsweise `restaurantsystem::datenbank` bzw. `gaesteverwaltung::datenbank`.

Der Inhalt ist mit seinem Paket untrennbar verbunden. Alle Elemente eines Pakets werden aus dem Modell entfernt, wenn das Paket gelöscht wird.

Wenn ein Paket keine Elemente aufzeigt, heißt dies nicht automatisch, dass es keine besitzt. Die UML erlaubt, den Inhalt von Paketen auszublenden, um das Paketdiagramm übersichtlicher zu gestalten.

Alle in einem Paket gruppierten Elemente sind innerhalb eines Pakets untereinander sichtbar. Ihre Sichtbarkeit außerhalb ihres Pakets kann eingeschränkt werden, indem sie als `protected` (`#`), `private` (`-`) oder `package` (`~`) (siehe auch Abschnitt 2.3.2) definiert wird (siehe Abbildung 7.5).

In Abbildung 7.5 wird ein `Goldbarren` als nur innerhalb des `tresors` sichtbar definiert (`private`). Von außerhalb darf damit nicht auf den Goldbarren zugegriffen werden (was manchen Einbrechern durchaus Schwierigkeiten bereiten könnte).

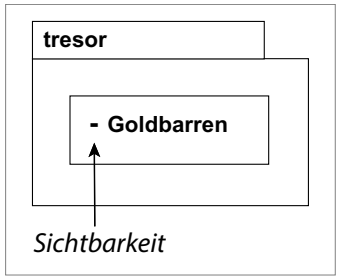


**Abbildung 7.5** Sichtbarkeit eines Paketelements

Verzichtet man auf die Spezifikation einer Sichtbarkeit, wird `public` (`+`) angenommen, womit das Element auch außerhalb des Pakets über seinen qualifizierten Namen referenziert werden kann.

### Verwendung

Durch die Verwendung von Paketen teilen Sie das System horizontal auf. Sie strukturieren die modellierten Klassen und sogar ganze Systeme in logisch und funktionell





zusammengehörende Einheiten, modularisieren sie und gestalten ein Modell damit einfacher und überschaubarer.

Hierarchieebenen von Paketen erlauben Ihnen eine vertikale Strukturierung des Modells. Sie gliedern ein Gesamtsystem damit auf abstrakter Ebene bereits in Teilsysteme und deren Bestandteile auf und schaffen damit eine wichtige Grundlage für Komponentendiagramme (siehe Kapitel 5).



### Realisierung in Java

Als Beispiel soll das Paketdiagramm aus Abbildung 7.6 verwendet werden.

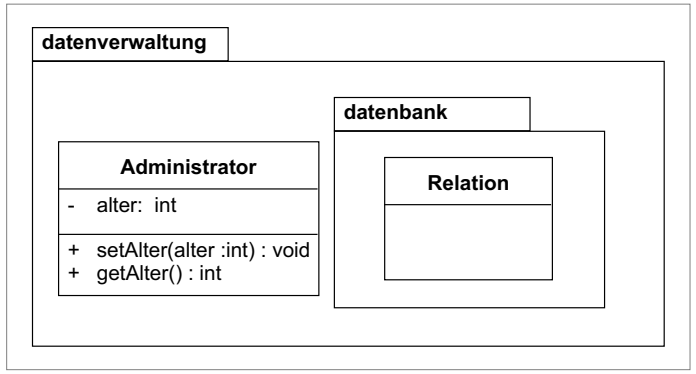


**Abbildung 7.6** Beispiel eines Paketdiagramms

Java verlangt die Abbildung modellierter Pakete auf Dateisystem-Ordner und die Implementierung von Klassen in separaten Dateien mit dem Namen der Klasse und einer Dateiendung `.java`.

Soll demnach das Paketdiagramm aus Abbildung 7.6 in Java implementiert werden, muss ein Ordner mit dem Namen `datenverwaltung` im Dateisystem erstellt werden. In ihm muss eine Programmcode-Datei mit der Bezeichnung `Administrator.java` sowie ein weiterer Ordner `datenbank` angelegt werden, in dem sich wiederum eine Datei namens `Relation.java` befindet.

Die Zugehörigkeit einer Klasse zu einem Paket wird mit dem Schlüsselwort `package` deklariert:

```
package datenverwaltung; ❶
public class Administrator
{

  private int alter; ❷
  public void setAlter(int alter)
  {
    this.alter = alter;
```

```
  }
  public int getAlter()
  {
    return this.alter;
  }
}
```

**Listing 7.1** /beispiele/java/kap7/datenverwaltung/Administrator.java
(Download der Beispiele: www.rheinwerk-verlag.de/4546)

❶ Die Klasse `Administrator` gehört zum Paket `datenverwaltung`. Während die UML die Default-Sichtbarkeit von Elementen mit `public` definiert, gilt in Java `package` als Vorgabewert. Um mit dem Paketdiagramm aus Abbildung 7.6 konform zu sein, muss daher die Sichtbarkeit der Klasse explizit mit `public` angegeben werden.

❷ Alle Attribute und Operationen der Klasse müssen in Java in derselben Datei implementiert werden.

```
package datenverwaltung.datenbank; ❶
public class Relation
{
}
```

**Listing 7.2** /beispiele/java/kap7/datenverwaltung/datenbank/Relation.java

❶ In Java erfolgt die Deklaration von Unterpaketen mithilfe des Punkt-Operators. Hier wird definiert, dass die Klasse `Relation` ein Bestandteil des Pakets `datenbank` ist, das sich selbst wiederum im Paket `datenverwaltung` befindet.

### Realisierung in C#

Das Paket-Konzept von C# weist einige Unterschiede zum Paket-Konzept von Java auf. C# verlangt nicht, dass jedes Paket durch einen eigenen Ordner im Dateisystem abgebildet wird. Ebenso entfällt die Beschränkung, jede Klasse in einer separaten Datei implementieren zu müssen. C# erlaubt, unterschiedliche Pakete, Unterpakete und Klassen in derselben Datei zu deklarieren und zu implementieren:

```
namespace Datenverwaltung ❶
{
  public class Administrator ❷
  {
      private int _Alter;
      public int Alter
      {
          get => _Alter;
```





```
          set => _Alter = value;
      }
  }
}

namespace Datenbank ❸
{
    public class Relation ❹
    {
    }
}
```

**Listing 7.3** /beispiele/c#/kap7/kap_7_3_1/Kap_7_3_1.cs

❶ Pakete definieren Namensräume für Elemente und werden daher in C# mit dem Schlüsselwort `namespace` deklariert, was an dieser Stelle für das Paket `Datenverwaltung` erfolgt.

❷ Innerhalb des Namensraums wird eine Klasse `Administrator` implementiert. Bezüglich der Default-Sichtbarkeit weist C# denselben Unterschied zur UML auf wie Java (UML: `public`, C#: `package`).

Zur Vereinfachung könnte hier ein automatisches Property verwendet werden, bei dem das Attribut, welches zur eigentlichen Datenablage verwendet wird, erst vom Compiler hinzugefügt wird.

❸ Ein Unterpaket kann in C# intuitiv innerhalb des umfassenden Pakets deklariert werden. Hier wird ein Paket (`namespace`) `Datenbank` innerhalb des Pakets `Datenverwaltung` angelegt.

❹ Das Unterpaket `Datenbank` enthält eine Klasse `Relation`.

Seit der Version 2.0 ist es in C# möglich, die Implementierung einzelner Klassen auf unterschiedliche Dateien zu verteilen. So kann beispielsweise die Deklaration der Attribute von der Implementierung der Operationen getrennt und auf beliebig viele Quellcodedateien aufgeteilt werden.

```
namespace Datenverwaltung
{
  public partial class Administrator ❶
  {
      private int _Alter; ❷
  }
}
```

**Listing 7.4** /beispiele/c#/kap7/kap_7_3_1/Admin_Attribute.cs

❶ Die Aufteilung einer Klasse auf mehrere Quellcodedateien wird mit dem Schlüsselwort `partial` signalisiert.

❷ Es wird nur das private Attribut `_Alter` deklariert.

In einer weiteren Datei können dann beispielsweise die Zugriffsoperationen implementiert werden:

```
namespace Datenverwaltung
{
   partial class Administrator ❶
   {
       public int Alter ❷
       {
             get => _Alter;
             set => _Alter = value;
       }
   }
}
```

**Listing 7.5** /beispiele/c#/kap7/kap_7_3_1/Admin_Operationen.cs

❶ Auch in dieser Quellcodedatei muss die partielle Implementierung mit dem Schlüsselwort `partial` signalisiert werden.

❷ Es werden nur die Operationen der Klasse implementiert, also hier der Getter und Setter des Properties `Alter`.

### 7.3.2 Paket-Import

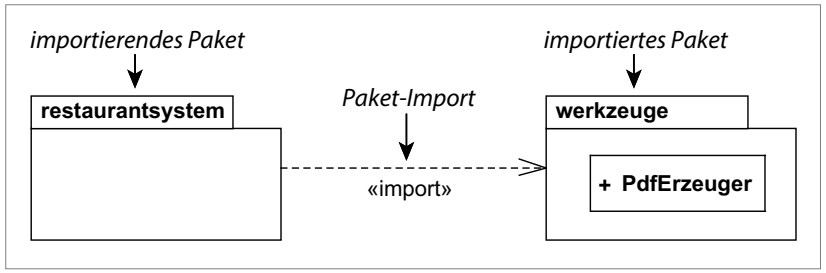


**Abbildung 7.7** Paket-Import

#### Beschreibung

Ein **Paket-Import** (engl. *Package Import*) ist eine Beziehung, die alle **Namen öffentlicher Elemente** eines Pakets dem importierenden Paket **als öffentlich** hinzufügt.





Damit wird die Referenzierung von Elementen eines Pakets über unqualifizierte Namen möglich, so als wenn das importierende Paket diese Elemente selbst enthalten würde. Wird das importierende Paket aus dem Modell entfernt, bleiben die Elemente im importierten Paket erhalten.

Laut Paketdiagramm aus Abbildung 7.7 kann `PdfErzeuger` im Paket `restaurantsystem` direkt über seinen unqualifizierten Namen angesprochen werden. In allen anderen Paketen, die `werkzeuge` nicht importieren, kann seine Referenzierung nur über den qualifizierten Namen `werkzeuge::PdfErzeuger` erfolgen.

Die importierten Elemente sind im importierenden Paket als öffentlich sichtbar. Damit kann ein weiteres Paket die Elemente erneut importieren (siehe Abbildung 7.8).

Im Paket `restaurantkette` aus Abbildung 7.8 kann `PdfErzeuger` direkt über seinen unqualifizierten Namen angesprochen werden. Er ist sowohl in `werkzeuge` als auch im `restaurantsystem` als öffentlich zugänglich und damit auch in `restaurantkette` verfügbar.

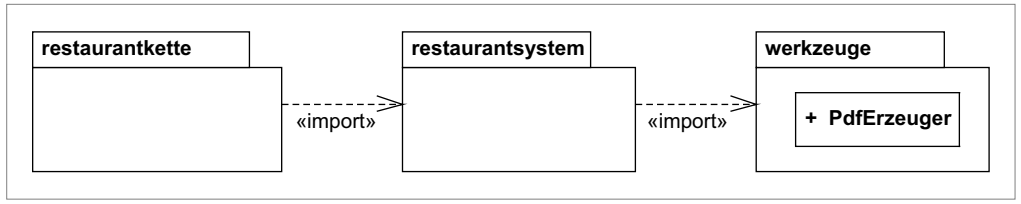


**Abbildung 7.8** Mehrfacher Paket-Import

Um dies zu verhindern, bietet die UML den **Paket-Access** als eine Einschränkung des Paket-Imports.

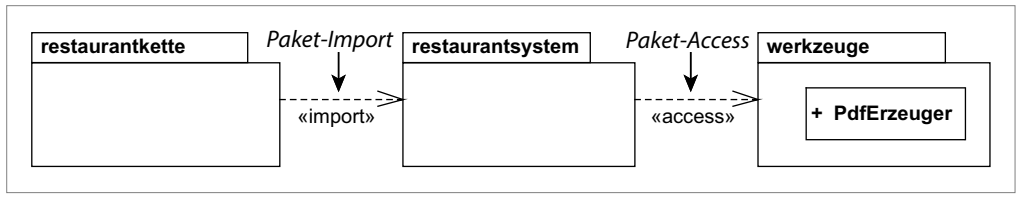


**Abbildung 7.9** Paket-Access

Ein **Paket-Access** (engl. *Package Access*) ist eine Beziehung, die alle **Namen öffentlicher Elemente** eines Pakets dem importierenden Paket **als privat** hinzufügt.

In Abbildung 7.9 importiert das Paket `restaurantsystem` alle Elementnamen des Pakets `werkzeuge` über eine `<<access>>`-Beziehung, wodurch sie als privat (`private`) deklariert werden.

Trotz der <<import>>-Beziehung zwischen restaurantkette und restaurantsystem kann damit in restaurantkette kein Zugriff auf den PdfErzeuger über seinen unqualifizierten Namen erfolgen.

Etwas kompakter kann die <<import>>- und <<access>>-Beziehung auch innerhalb von Paketen notiert werden (siehe Abbildung 7.10). Die Elemente müssen dabei über ihre qualifizierten Namen eindeutig identifiziert werden.

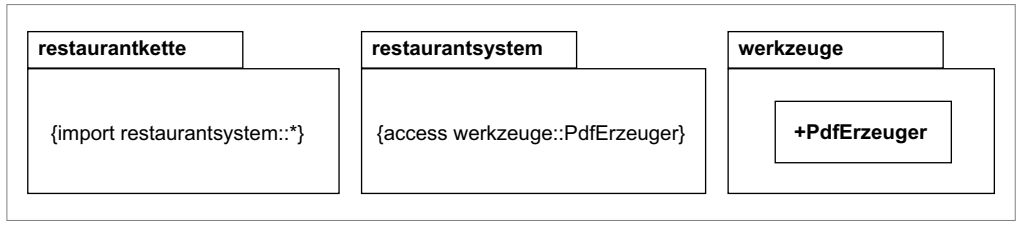


**Abbildung 7.10** Alternative Notation für Paket-Import und -Access

Das Paket restaurantkette aus Abbildung 7.10 importiert alle Elemente des Pakets restaurantsystem, was an dem notierten * (Stern) hinter restaurantsystem zu erkennen ist. Das Paket restaurantsystem macht dagegen von der Möglichkeit Gebrauch, das zu importierende Element genau zu spezifizieren, indem es explizit den PdfErzeuger aus dem Paket werkzeuge benennt. Eventuelle weitere Elemente von werkzeuge werden nicht importiert.

### Verwendung

Verwenden Sie Paket-Importe, wenn Sie Elemente eines Pakets häufig in weiteren Paketen wiederverwenden möchten. Paket-Importe ermöglichen nicht nur die Referenzierung über unqualifizierte Namen. Sie verdeutlichen auch die Struktur des Modells und die Beziehungen der Pakete untereinander.

Soll der Zugriff auf die importierten Elemente in weiteren Paketen eingeschränkt werden, ist die Access-Beziehung vorzuziehen.

### Realisierung in Java

Java enthält nativ keine Unterstützung des öffentlichen Imports von Paketen, der mit dem Stereotyp <<import>> gekennzeichnet wird. Alle importierten Elementnamen sind nur im jeweils importierenden Paket verfügbar, was der <<access>>-Beziehung in der UML entspricht. Eine <<import>>-Beziehung muss daher in Java durch <<access>>-Beziehungen ersetzt werden. Dies kann z. B. auf die in Abbildung 7.11 dargestellte Art durchgeführt werden.





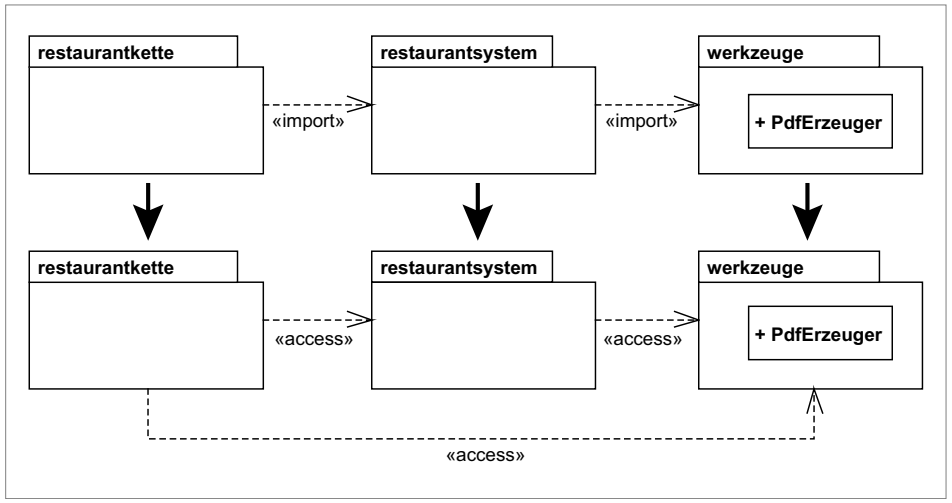


**Abbildung 7.11** Umwandlung von <<import>> in <<access>>

Im oberen Teil der Abbildung 7.11 kann vom Paket restaurantkette sowohl auf die (nicht gezeigten) Bestandteile von restaurantsystem als auch auf die von werkzeuge über deren unqualifizierte Namen zugegriffen werden.

Um vergleichbare Zugriffsmöglichkeiten mithilfe von <<access>>-Beziehungen zu modellieren, muss von jedem einzelnen Paket zu allen zuvor direkt oder indirekt über <<import>>-Beziehungen erreichbaren Paketen eine eigene <<access>>-Beziehung modelliert werden (unterer Teil von Abbildung 7.11).

Es muss jedoch darauf hingewiesen werden, dass der obere und untere Teil der Abbildung 7.11 nicht exakt äquivalent sind: Ein weiteres Paket, das restaurantkette importiert, hätte im oberen Teil der Abbildung direkten Zugriff auf PdfErzeuger, im unteren Teil nicht.

Als Vorlage für die Java-Realisierung soll das folgende Paketdiagramm verwendet werden:

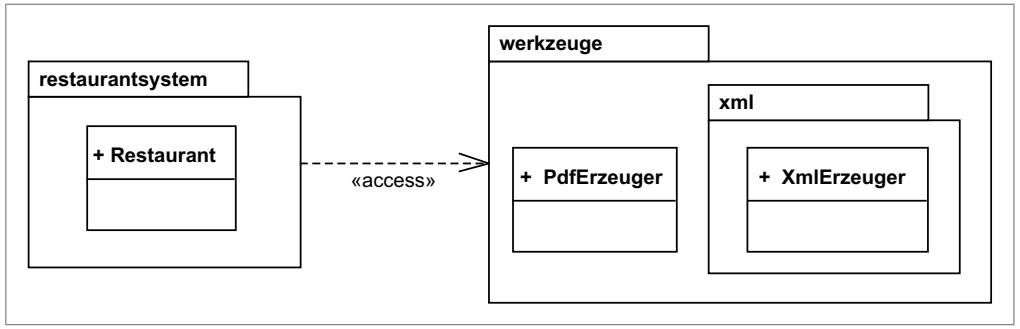


**Abbildung 7.12** Beispiel für einen Paket-Import

```
package restaurantsystem; ❶
import werkzeuge.PdfErzeuger; ❷

import werkzeuge.xml.*; ❸
public class Restaurant
{
  PdfErzeuger p = new PdfErzeuger();
  XmlErzeuger x = new XmlErzeuger();
}
```

**Listing 7.6** /beispiele/java/kap7/restaurantsystem/Restaurant.java

❶ Die im Folgenden implementierte Klasse `Restaurant` befindet sich im Paket `restaurantsystem`.

❷ Der Klassenname `PdfErzeuger` aus dem Paket `werkzeuge` wird in das aktuelle Paket importiert. Im Unterschied zur UML, in der das *Paket* externe Elemente importiert, ist dies in Java die Aufgabe einer jeden *Klasse* des jeweiligen Pakets.

Obwohl es sich im Sinne der UML streng genommen um eine `<<access>>`-Beziehung handelt, verwendet Java hierfür das Schlüsselwort `import`.

Es sollte an dieser Stelle nochmals betont werden, dass *nicht die Klasse* selbst, sondern *nur deren Name* importiert wird. Es wird also lediglich die Referenzierung der Klasse über ihren unqualifizierten Namen `PdfErzeuger` ermöglicht. Wie von der UML definiert, ist die Klasse auch ohne den Import über ihren qualifizierten Namen `werkzeuge.PdfErzeuger` erreichbar.

❸ Unterpakete werden in Java mit dem Punkt-Operator referenziert. Das an dieser Stelle verwendete *-Zeichen stellt in Java eine Art Joker dar, durch den alle Klassennamen des Pakets importiert werden.

#### Realisierung in C#

Auch C# unterstützt die `<<import>>`-Beziehung nicht. Der folgende Programmcode realisiert das Diagramm aus Abbildung 7.12:

```
namespace Restaurantsystem ❶
{
    using Werkzeuge; ❷
    using Werkzeuge.Xml;

    class Restaurant
    {
        PdfErzeuger p = new PdfErzeuger();
        XmlErzeuger x = new XmlErzeuger();
    }
}
```

**Listing 7.7** /beispiele/c#/kap7/kap_7_3_2/Kap_7_3_2.cs

❶ Die im Folgenden implementierten Klassen und Pakete befinden sich im Paket (`namespace`) `Restaurantsystem`.

❷ C# verwendet das Schlüsselwort `using`, um einen Paket-Access zu deklarieren.

Es werden immer alle Klassennamen des Pakets importiert. Der Import eines einzelnen Klassennamens wird von C# nicht unterstützt.

Der Zugriff auf Unterpakete erfolgt wie in Java mithilfe des Punkt-Operators.

### 7.3.3 Paket-Merge

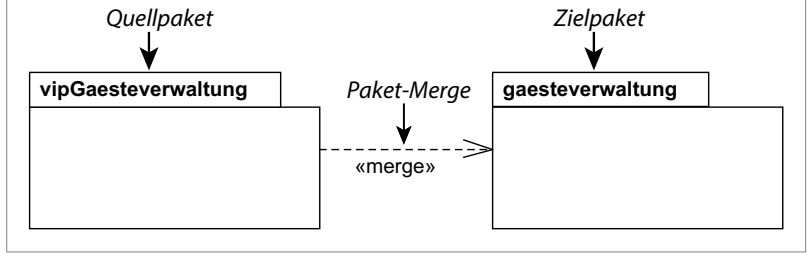


**Abbildung 7.13** Paket-Merge

#### Beschreibung

Ein **Paket-Merge** (engl. *Package Merge*) definiert eine Beziehung zwischen zwei Paketen, bei der die **nicht privaten Inhalte** des Zielpakets mit den Inhalten des Quellpakets **verschmolzen werden**.

Im Beispiel aus Abbildung 7.13 wird der Inhalt des Pakets `gaesteverwaltung` mit dem Inhalt des Pakets `vipGaesteverwaltung` verschmolzen.

Im Prinzip stellt eine Merge-Beziehung eine verkürzende Notation für alle Transformationen dar, die bei der Verschmelzung des Zielpakets mit dem Quellpaket benötigt werden. Das Beispiel aus Abbildung 7.14 verdeutlicht die prinzipielle Funktionsweise eines Paket-Merge und zeigt die entsprechenden Transformationen.

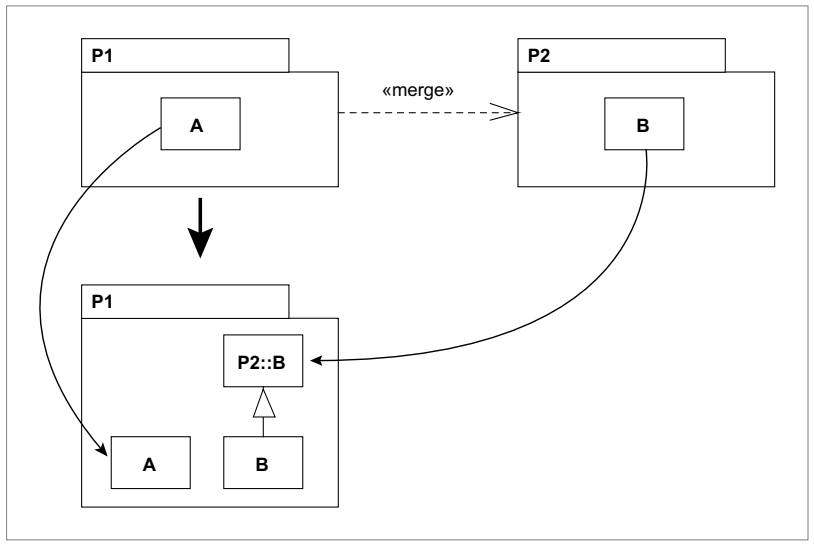


**Abbildung 7.14** Merge zweier Pakete mit unterschiedlichen Elementen

Im oberen Teil der Abbildung 7.14 wird ein Paket-Merge des Pakets P2 in das Paket P1 modelliert. Paket P1 definiert ein Element A, Paket P2 ein Element B.

Der untere Teil der Abbildung zeigt die Auswirkungen des Merge auf das Paket P1. Das ursprünglich bereits vorhandene Element A bleibt erwartungsgemäß erhalten. Da das Element B zuvor nicht in P1 existierte, wird ein neues Element B definiert, das das Element B des Pakets P2 (P2::B) spezialisiert und damit nach den Regeln der Generalisierung über alle Attribute und Operationen des Elements P2::B verfügt. (Generalisierung wurde in Abschnitt 2.3.12 vorgestellt.)

Was passiert aber, wenn beide Pakete bereits Elemente mit denselben Namen enthalten?

Die Pakete P1 und P2 enthalten im oberen Teil der Abbildung 7.15 beide ein Element A. Bei einem Paket-Merge wird zwischen dem Element aus Paket P2 (P2::A) und dem Element aus Paket P1 (A) eine Generalisierungsbeziehung hinzugefügt, wie im unteren Teil der Abbildung dargestellt ist.

Damit erweitert das Element A seine eigenen Attribute und Operationen durch diejenigen des Elements P2::A. Es entsteht somit ein neues Element, das nach den Regeln der Generalisierung alle Attribute und Operationen der Elemente P1::A und P2::A besitzt.







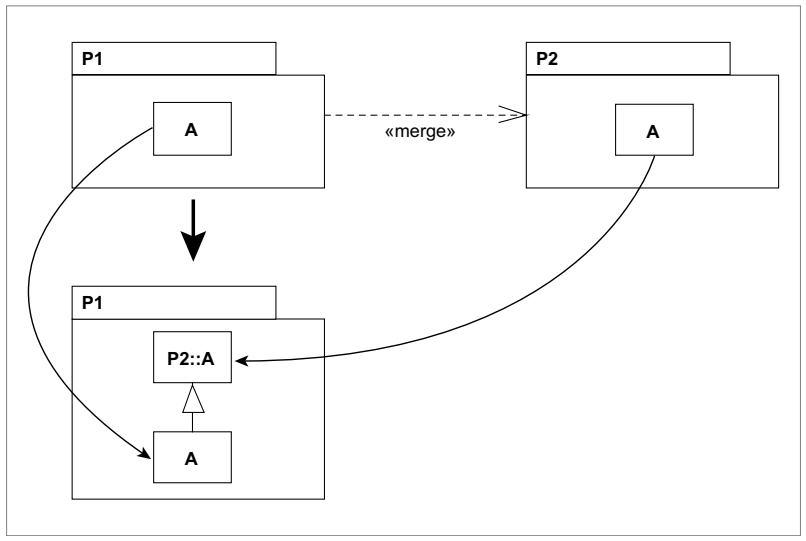


**Abbildung 7.15** Merge von Elementen mit demselben Namen

Betrachten wir noch ein weiteres Beispiel, in dem ein leeres Paket zwei weitere Pakete mergt, die beide dasselbe Element enthalten (siehe Abbildung 7.16).

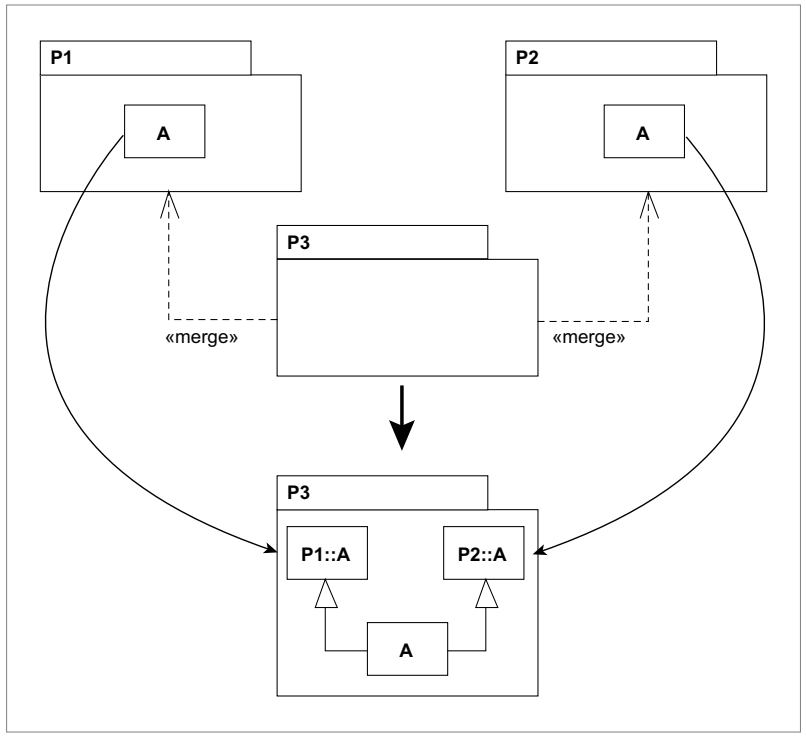


**Abbildung 7.16** Ein leeres Paket mergt Elemente mit demselben Namen.

In Abbildung 7.16 werden die zwei Pakete P1 und P2, die jeweils ein Element A definieren, in ein leeres Paket P3 gemergt. Bei der Verschmelzung wird ein neues Element A definiert, das von P1::A und P2::A erbt und damit alle Attribute und Operationen beider Elemente nach den Regeln der Generalisierung in sich vereinigt.

Es ist durchaus üblich, dass in den einzelnen Paketen bereits Generalisierungen und Assoziationen definiert sind. Eine Merge-Beziehung verändert die Generalisierungen und Assoziationen nicht.

Sie verändert lediglich die Elemente, die an ihnen teilnehmen, was am Beispiel der Abbildung 7.17 verdeutlicht wird.

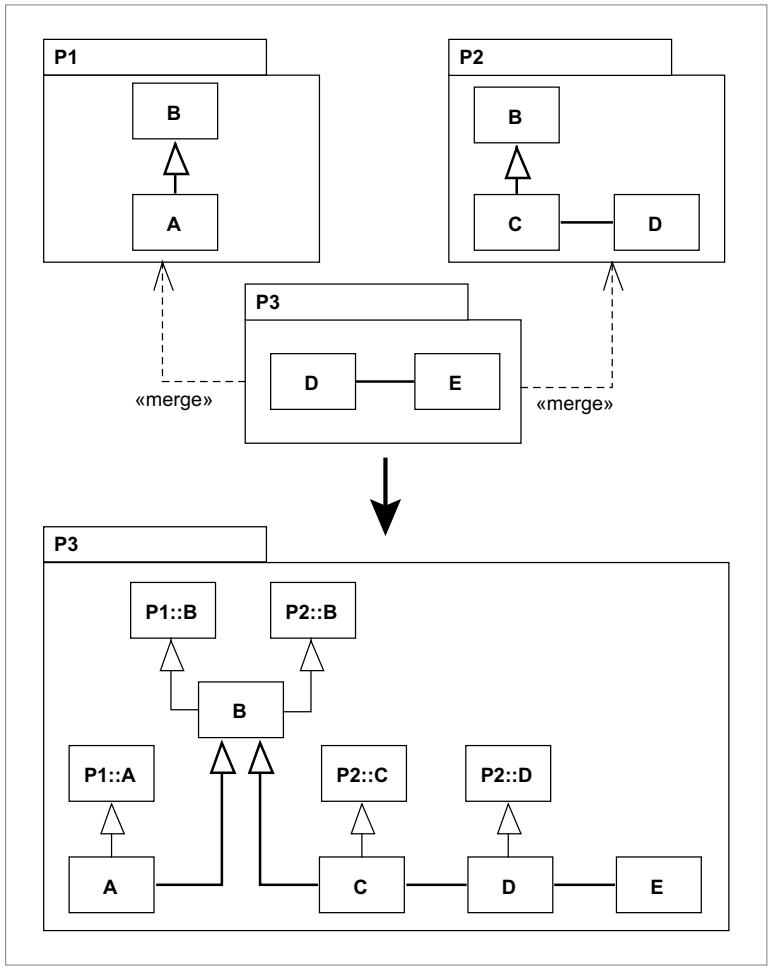


**Abbildung 7.17** Merge von Elementen mit Generalisierungen und Assoziationen

Im oberen Teil der Abbildung 7.17 definiert Paket P1 ein Element B, das von A spezialisiert wird. Im Paket P2 wird dagegen B vom Element C spezialisiert, das wiederum eine





Assoziation zu D besitzt. Paket P3 mergt die beiden Pakete P1 und P2 und definiert gleichzeitig zwei Elemente D und E, die miteinander durch eine Assoziation verbunden sind.

Der untere Teil der Abbildung 7.17 zeigt die fünf durch den Merge entstandenen Elemente A, B, C, D und E. Deren Struktur hat sich durchaus verändert, die ursprünglichen Generalisierungen und Assoziationen (hervorgehoben) bleiben jedoch erhalten.

Es bleibt noch zu klären, was bei einem Merge mit Paketen passiert, die selbst in weiteren Paketen enthalten sind. Den einfachsten Fall zeigt Abbildung 7.18.

Im oberen Teil der Abbildung 7.18 beinhaltet das Paket P1 das Paket P3 nicht. Daher wird beim Merge (unterer Teil der Abbildung) in P1 ein neues Paket P3 definiert, das mit dem Unterpaket P3 aus P2 (P2::P3) eine Merge-Beziehung eingeht.

Wie durch dieses Beispiel angedeutet, folgen die Unterpakete bei Merge denselben Regeln wie innere Elemente von Paketen. Während die Elemente durch die Verwendung von Generalisierungen verschmolzen werden, geschieht dies bei Paketen durch Merge-Beziehungen.

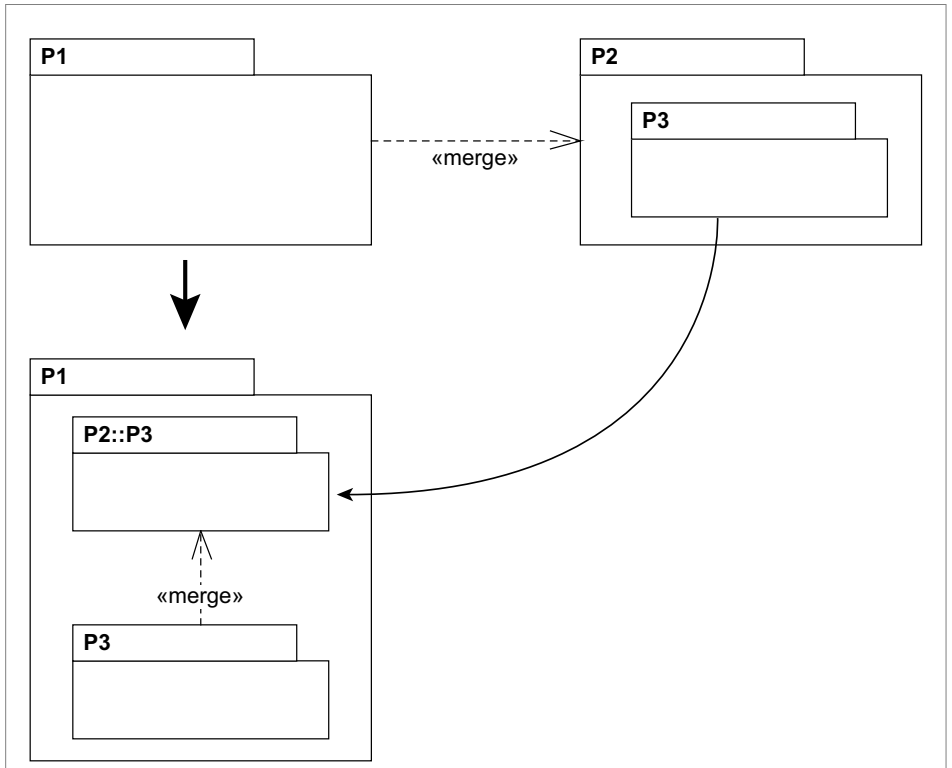


**Abbildung 7.18** Merge eines Unterpakets

Weitere Beziehungen zwischen Paketen (<<import>> und <<access>>) bleiben bei einem Paket-Merge erhalten.

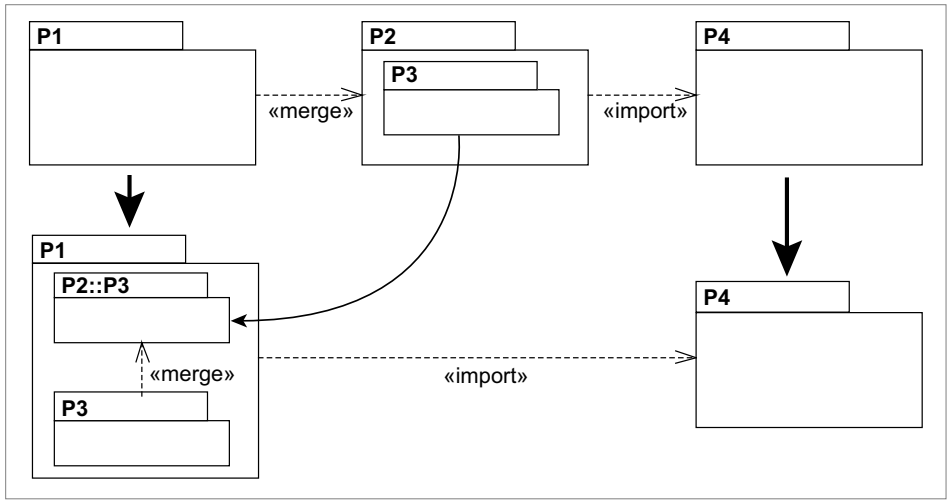


**Abbildung 7.19** Import-Beziehungen bleiben beim Merge erhalten.

Der obere Teil der Abbildung 7.19 modelliert ein Merge zwischen dem Paket P1 und dem Paket P2, das seinerseits P4 importiert.

Trotz des Merge muss die Import-Beziehung erhalten bleiben, da das Paket P2::P3 andernfalls nicht mehr funktionsfähig wäre. Daher entsteht eine <<import>>-Beziehung zwischen dem Quellpaket P1 und dem Paket P4, wie im unteren Teil von Abbildung 7.19 dargestellt ist.

### Verwendung

Merge-Beziehungen werden verwendet, wenn im Modell Pakete vorhanden sind, deren Inhalte und Konzepte sich ergänzen und daher zu neuer Gesamtheit zusammengesetzt werden können.

Wie eingangs erwähnt, stellt eine Merge-Beziehung lediglich eine abkürzende Notation für Transformationen der einzelnen Elemente aus Paketen unter Zuhilfenahme von Generalisierungen dar. Erkennen Sie, dass Bedarf an der Spezialisierung vieler Objekte aus unterschiedlichen Paketen besteht, um ein ähnlich arbeitendes Paket zu erhalten, können Sie die Merge-Beziehung einsetzen.

Sie sollten die entstehende Vererbungshierarchie jedoch genau überprüfen, da mit der Verwendung der Merge-Beziehung auch unerwünschte Effekte auftreten können (Beispiel in Abbildung 7.20).







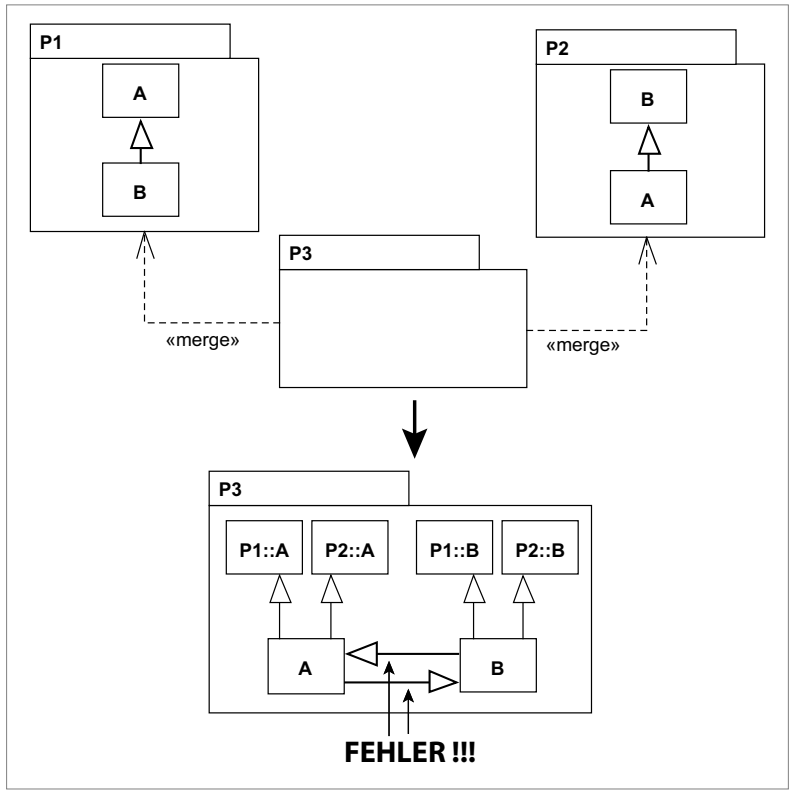


**Abbildung 7.20** Gefahr der Merge-Beziehung

Abbildung 7.20 zeigt zwei Pakete P1 und P2. In Paket P1 spezialisiert das Element B ein Element A, in Paket P2 ist es genau umgekehrt: A spezialisiert B.

Werden die beiden Pakete mithilfe der Merge-Beziehung in Paket P3 verschmolzen, entsteht eine fehlerhafte Generalisierungshierarchie, in der sowohl A von B als auch B von A erbt, wovor bereits in Abschnitt 2.5, »Irrungen und Wirrungen«, gewarnt wurde.

Verwenden Sie die Merge-Beziehung daher nur mit großer Vorsicht.

### Realisierung in Java

Java bietet kein Sprachmittel, das einem Paket-Merge entspräche, sodass die entstehende Hierarchie der Generalisierungen einzeln realisiert werden muss. Wie man in Java eine Generalisierung implementiert, wird in Abschnitt 2.3.12 erläutert.

### Realisierung in C#

Für C# gilt bei der Merge-Beziehung dasselbe wie für Java.

## 7.4 Lesen eines Paketdiagramms

Das Paketdiagramm aus Abbildung 7.21 definiert sieben Pakete:

- `restaurantsystem`
- `gaesteverwaltung`
- `werkzeuge`
- `datenverwaltung` mit zwei Unterpaketen `datenbank` und `datenpflege`
- `vipGaesteverwaltung`

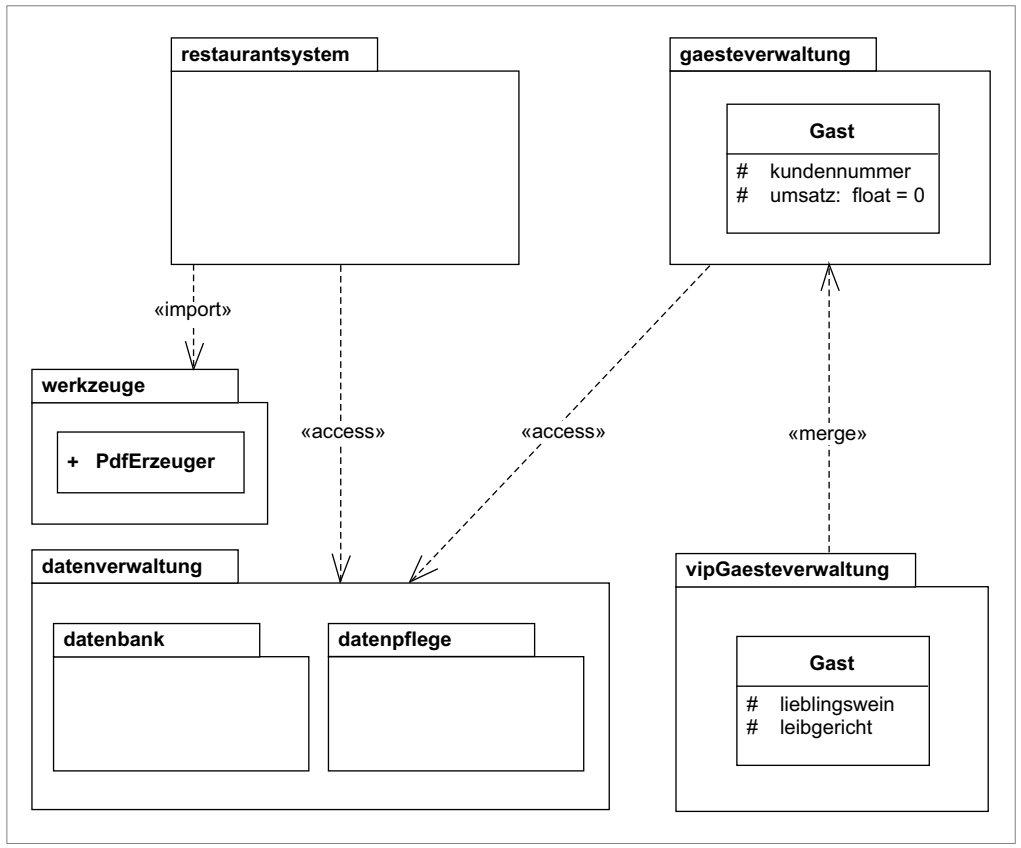


**Abbildung 7.21** Beispiel für ein Paketdiagramm

Das Paket `restaurantsystem` importiert den Klassennamen `PdfErzeuger` aus dem Paket `werkzeuge` als öffentlich, die Bestandteile der `datenverwaltung` als privat. Damit wird der direkte Zugriff auf Elemente der `datenverwaltung` von allen weiteren Paketen, die `restaurantsystem` importieren, verboten.

Auch das Paket `gaesteverwaltung` importiert die `datenverwaltung` als privat und verbietet damit allen weiteren Paketen den direkten Zugriff.





Die Klasse `Gast` ist sowohl im Paket `gaesteverwaltung` als auch in `vipGaesteverwaltung` enthalten. Da beide Pakete unterschiedliche Namensräume definieren, entsteht dadurch kein Namenskonflikt.

Das Paket `vipGaesteverwaltung` mergt jedoch die `gaesteverwaltung`, wodurch eine Generalisierung zwischen den beiden `Gast`-Klassen definiert wird und die Klasse `vipGaesteverwaltung::Gast` alle Attribute der Klasse `gaesteverwaltung::Gast` erbt.



## 7.5 Irrungen und Wirrungen

Abbildung 7.22 zeigt eine Auswahl der häufigsten Fehler bei der Modellierung mit Paketdiagrammen.

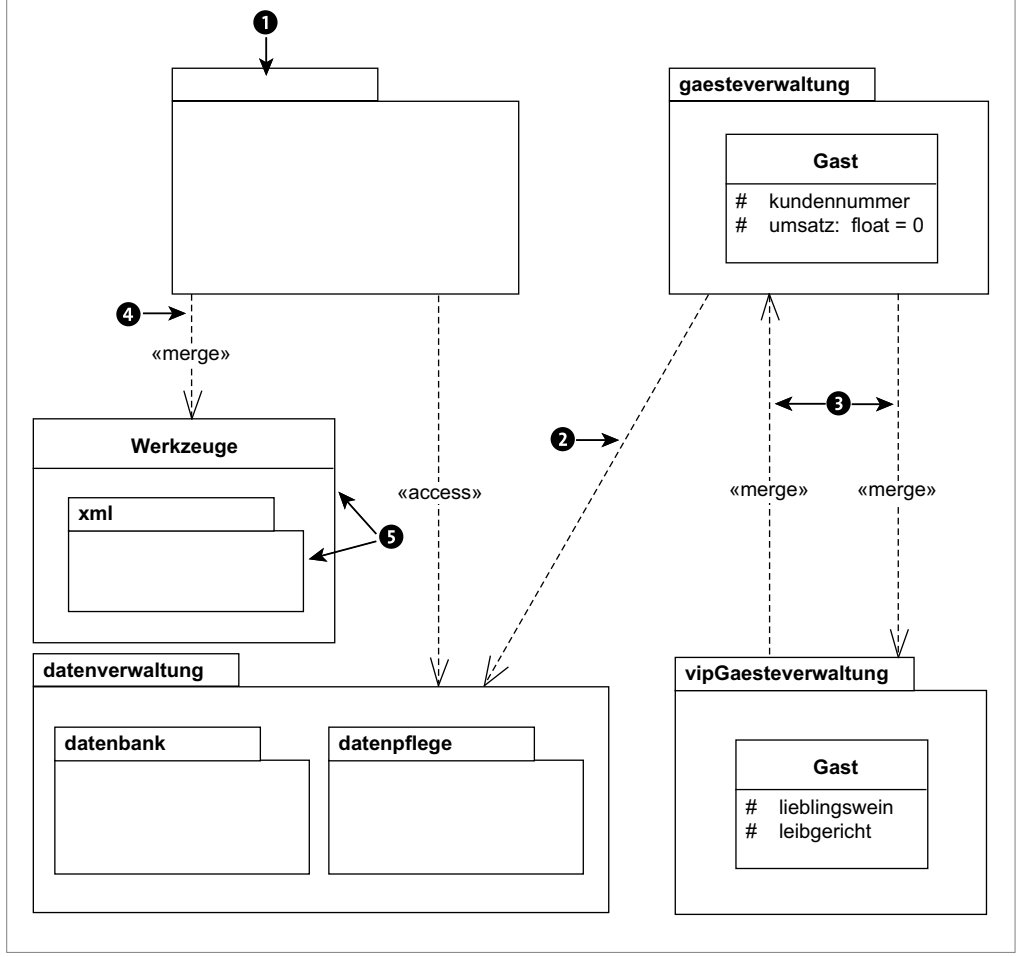


**Abbildung 7.22** Ein fehlerhaftes Paketdiagramm

❶ **Paketname fehlt**
Pakete müssen mit Namen eindeutig definiert werden.

❷ **Bezeichnung der Beziehung fehlt**
Definieren Sie die Art der modellierten Beziehung. Handelt es sich um eine `<<import>>`-, `<<access>>`- oder eine `<<merge>>`-Beziehung?

❸ **Zirkulare <<merge>>-Beziehung**
Zwei Pakete dürfen sich nicht gegenseitig mergen. Achten Sie auch darauf, dass keine zirkulare Beziehung zwischen mehreren Paketen entsteht, die möglicherweise nicht so einfach zu erkennen ist wie in diesem Beispiel.

❹ **Ungültige <<merge>>-Beziehung**
Die `<<merge>>`-Beziehung darf nur zwischen Paketen modelliert werden. Ein Merge einer Klasse mit einem Paket ist nicht definiert.

❺ **Falsche Hierarchie**
Klassen können weitere Klassen, jedoch keine Pakete enthalten. Andersherum ist es Paketen durchaus gestattet, Klassen in sich zu gruppieren.

## 7.6 Zusammenfassung

Abschließend werden die wichtigsten Notationselemente kurz aufgeführt:

- **Pakete** gruppieren Elemente wie Klassen oder weitere Pakete und definieren Namensräume.

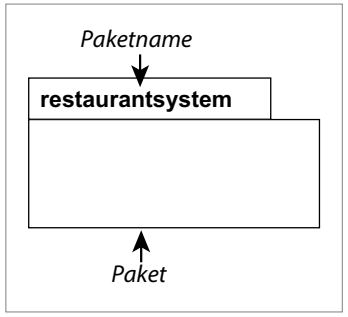


**Abbildung 7.23** Paket





- Ein **Paket-Import** importiert alle Namen des importierten Pakets als öffentlich, ein **Paket-Access** als privat.

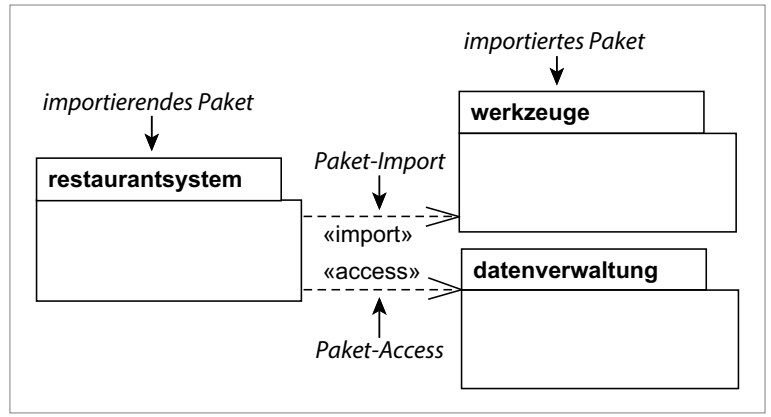


**Abbildung 7.24** Paket-Import und -Access

- Mithilfe eines **Paket-Merge** können die Inhalte von Paketen verschmolzen werden.

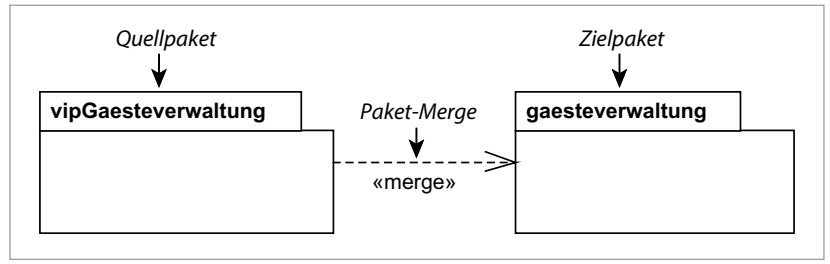


**Abbildung 7.25** Paket-Merge