

Klassendiagramme

Ein Sachverhalt ist denkbar, heißt:
Wir können uns ein Bild von ihm machen.
Ludwig Wittgenstein

Klassendiagramme bilden das architekturelle Rückgrat vieler Systemmodellierungen. Deshalb werden in diesem Kapitel die in der UML/P definierten Klassendiagramme mit den Kernelementen *Klasse*, *Attribut*, *Methode*, *Assoziation* und *Komposition* eingeführt. Im Abschnitt über *Sichten* und *Repräsentationen* werden Einsatzvarianten von Klassendiagrammen diskutiert. Es wird außerdem gezeigt, wie mit *Stereotypen* und *Merkmalen* Modellierungskonzepte für projektspezifische Problemstellungen angepasst werden.

2.1	Bedeutung der Klassendiagramme	16
2.2	Klassen und Vererbung	19
2.3	Assoziationen	25
2.4	Sicht und Repräsentation	31
2.5	Stereotypen und Merkmale	34

Klassendiagramme bilden nach wie vor die mit Abstand wichtigste und am meisten genutzte Modellierungstechnik der UML. Historisch entstanden sind Klassendiagramme aus Anleihen von der Entity/Relationship-Modellierung [Che76] und der graphischen Darstellung von Modulen, die ihrerseits von Datenfluss-Diagrammen [DeM79] beeinflusst wurden. Klassendiagramme beschreiben die Struktur eines Softwaresystems und bilden daher die erste behandelte Kernnotation für die objektorientierte Modellierung.

Im Anhang C.2 wird ergänzend die hier präsentierte Form der Klassendiagramme mit dem UML-Standard verglichen und die Syntax der Klassendiagramme präzisiert.

2.1 Bedeutung der Klassendiagramme

Objektorientierte Systeme beinhalten eine hohe Dynamik. Dadurch wird die Modellierung der Strukturen eines Systems zu einer komplexen Aufgabe in der objektorientierten Softwareentwicklung. Klassendiagramme beschreiben diese Struktur beziehungsweise Architektur eines Systems, auf der nahezu alle anderen Beschreibungstechniken basieren. Klassendiagramme und die darin modellierten Klassen haben jedoch eine Vielfalt von Aufgaben.

Modellierung von Struktur

In einer objektorientierten Implementierung wird der Code in Form von Klassen organisiert. Ein Klassendiagramm stellt daher eine Übersicht über die Code-Struktur und seine inneren Zusammenhänge dar. Weil Programmierern das Konzept *Klasse* aus der Programmierung bekannt ist, sind die in der Modellierung genutzten Klassendiagramme auch leicht verständlich und kommunizierbar. Klassendiagramme werden zur Darstellung der strukturellen Zusammenhänge eines Systems eingesetzt und bilden so das Skelett für fast alle weiteren Notationen und Diagrammarten, da diese sich jeweils auf die in Klassendiagrammen definierten Klassen und Methoden abstützen. Auch deshalb bilden Klassendiagramme ein essentielles – wenn auch nicht einziges – Beschreibungsmittel zur Modellierung von Softwarearchitekturen und Frameworks.

Klassen in Analyse, Design und Implementierung

In der Analyse werden Klassendiagramme genutzt, um Konzepte der realen Welt zu strukturieren. Demgegenüber werden Klassendiagramme bei der Erstellung von Entwurfsdokumenten und in der Implementierung vor allem zur Darstellung einer strukturellen Sicht des Softwaresystems genutzt. Die in der Implementierungssicht dargestellten Klassen sind tatsächlich im implementierten System wieder zu finden. Klassen der Analyse werden dafür

oft signifikant modifiziert, durch technische Aspekte ergänzt oder ganz weggelassen, weil sie z.B. nur zum Systemkontext gehören.

Eines der Defizite der UML entsteht aus der nicht optimalen Möglichkeit, den Diagrammen explizit einen Verwendungszweck zuzuordnen. Wird der Standpunkt eingenommen, dass ein Klassendiagramm eine Implementierung widerspiegelt, so kann die Semantik eines Klassendiagramms relativ einfach und verständlich erklärt werden. Diesen Standpunkt nehmen eine Reihe von Einführungsbüchern in die Modellierung mit Klassen beziehungsweise der UML ein [Mey97, Fow00]. Außerdem wird dieser Standpunkt oft auch durch Werkzeuge impliziert. Fusion [CAB⁺94] stellt demgegenüber eine explizite Abgrenzung zwischen zum System gehörigen und externen Klassen zur Verfügung und demonstriert so, dass die Modellierung von nicht-softwaretechnischen Konzepten mit Klassendiagrammen möglich und sinnvoll ist.

Das Sprachprofil UML/P ist implementierungsorientiert. Deshalb ist die nachfolgende Bedeutungserklärung von Klassendiagrammen auf Basis des dadurch modellierten Java-Codes für diesen Einsatz ideal.

Aufgabenvielfalt einer Klasse

In der objektorientierten Programmierung und stärker noch der Modellierung haben Klassen eine Vielzahl von Aufgaben. Primär dienen sie zur *Gruppierung* und *Kapselung* von Attributen und dazugehörigen Methoden zu einer konzeptuellen Einheit. Durch Vergabe eines *Klassennamens* können *Instanzen* der Klasse an beliebigen Stellen im Code erzeugt, gespeichert und weitergereicht werden. Klassendefinitionen dienen daher gleichzeitig als *Typsystem* und als *Implementierungsbeschreibung*. Sie können (im Allgemeinen) beliebig oft in Form von *Objekten* instanziiert werden.

In der Modellierung wird eine Klasse auch als *Extension*, also als die Menge aller zu einem bestimmten Zeitpunkt existierenden Objekte, verstanden. Durch die explizite Verfügbarkeit der Extension in der Modellierung kann zum Beispiel die Einhaltung einer Invariante für jedes existierende Objekt einer Klasse beschrieben werden.

Weil die Anzahl der Objekte in einem System potentiell unbeschränkt ist, ist die Katalogisierung der Objekte in endlich viele Klassen notwendig. Dadurch wird eine endliche Aufschreibung eines objektorientierten Systems erst ermöglicht. Klassen stellen damit eine *Charakterisierung der möglichen Strukturen* eines Systems dar. Diese Charakterisierung beschreibt gleichzeitig auch notwendige Strukturformen, ohne jedoch eine konkrete Objektstruktur festzulegen. Deshalb gibt es normalerweise unbeschränkt viele unterschiedliche Objektstrukturen, die einem Klassendiagramm genügen. In der Tat entspricht jedes korrekt laufende System einer sich weiterentwickelnden Sequenz von Objektstrukturen, bei der zu jedem Zeitpunkt die aktuelle Objektstruktur dem Klassendiagramm genügt.

Im Gegensatz zu den Objekten haben Klassen jedoch während der Laufzeit eines Systems in vielen Programmiersprachen keine direkt manipulierbare Repräsentation. Ausnahmen hierzu bilden etwa Smalltalk, das Klassen ebenfalls als Objekte repräsentiert und dadurch uneingeschränkte reflektive Programmierung erlaubt.¹ Java ist demgegenüber restriktiver, denn es erlaubt nur lesenden Zugriff auf den Klassencode. Generell sollte reflektive Programmierung nur sehr sparsam eingesetzt werden, weil eine Wartung des Systems aufgrund der reduzierten Verständlichkeit sehr viel komplexer wird. Deshalb wird im weiteren Verlauf auf reflektive Programmierung nicht weiter eingegangen.

Die Aufgaben einer Klasse sind:

- Kapselung von Attributen und Methoden zu einer konzeptuellen Einheit
- Ausprägung von Instanzen als Objekte
- Typisierung von Objekten
- Implementierungsbeschreibung
- Klassencode
(die übersetzte, ausführbare Form der Implementierungsbeschreibung)
- Extension (Menge aller zu einem Zeitpunkt existierenden Objekte)
- Charakterisierung der möglichen Strukturen eines Systems

Abbildung 2.1. Aufgabenvielfalt einer Klasse

Metamodellierung

Für die Beschreibung einer diagrammatischen Sprache hat sich aufgrund ihrer zweidimensionalen Darstellungsform die *Metamodellierung* [CEK⁺00, RA01, CEK01, Béz05, GPHS08, JJM09, AK03] als Präsentationsform durchgesetzt und damit die für Text üblichen Grammatiken abgelöst. Ein *Metamodell* definiert die abstrakte Syntax einer graphischen Notation. Spätestens seit der UML-Standardisierung ist es üblich, als Metamodell-Sprache selbst eine einfache Form von Klassendiagrammen einzusetzen. Dieser Ansatz hat den Vorteil, dass nur eine Sprache erlernt werden muss. Wir diskutieren Metamodellierung im Anhang A und nutzen eine Variante der Klassendiagramme um die graphischen Anteile der UML/P darzustellen.

¹ In Smalltalk manifestiert sich eine Klasse zur Laufzeit als normales Objekt, das wie andere Objekte manipuliert werden kann. Der Inhalt eines solchen Objekts ist allerdings eine Beschreibung von Struktur und Verhalten der diesem Klassen-Objekt zugeordneten Instanzen. Siehe [Gol84].

Weiterführende Konzepte für Klassendiagramme

In der UML werden weiterführende Konzepte angeboten, die hier der Vollständigkeit halber erwähnt sein sollen. Assoziationsklassen sind zum Beispiel Klassen, die an die nachfolgend noch eingeführten Assoziationen angefügt werden, um Information abzulegen, die keiner der an der Assoziation beteiligten Klassen, sondern nur der Beziehung selbst zugeordnet werden können. Es gibt aber Standardverfahren, solche Daten ohne Assoziationsklassen darzustellen.

Moderne Programmiersprachen wie C++ und Java [GJSB05] sowie auch die UML ab Version 2.3 [OMG10a] bieten mittlerweile die zunächst von funktionalen Sprachen wie Haskell [Hut07] eingeführten generischen Typen an. In Java ist diese Einführung gut gelungen [Bra04]. Dies muss mit Sorgfalt erfolgen, da Typisierungen in fast allen Diagrammtypen vorkommen. Da Generics bei der Modellierung keine ganz so wichtige Rolle einnehmen, sondern vor allem in der Implementierung für die Wiederverwendung generischer Komponenten genutzt werden, wird auf die volle Allgemeinheit generischer Klassen mit Wildcards, gebundener Typisierungen etc. in der UML/P verzichtet und nur die wichtigen Container-Klassen als generisch also mit Typparametern realisiert angeboten. Die Klassendiagramme der UML/P bieten daher keine Mechanismen zur Definition von Generizität, die OCL/P sowie die Codegenerierung gehen aber darauf ein.

2.2 Klassen und Vererbung

Bei der Einführung von Klassen, Attributen, Methoden und von Vererbung wird in diesem Abschnitt, wie bereits diskutiert, eine Implementierungssicht zugrunde gelegt. Die Abbildung 2.2 enthält eine Einordnung der wichtigsten Begriffe für Klassendiagramme.

In Abbildung 2.3 ist ein einfaches Klassendiagramm bestehend aus einer Klasse und einem angehängten Kommentar zu sehen. Die kursiven Erläuterungen und die geschwungenen Pfeile gehören nicht zum Diagramm selbst. Sie dienen zur Beschreibung von Diagrammelementen. Die Darstellung einer Klasse wird typischerweise in drei Felder unterteilt. Im ersten Feld wird der Klassenname angegeben.

2.2.1 Attribute

Das mittlere Feld einer Klassendefinition beschreibt die Liste von Attributen, die in dieser Klasse definiert werden. Die dargestellte Information über Attribute kann in mehrerer Hinsicht unvollständig sein. So kann ein Attribut mit oder ohne seinen Typ angegeben werden. Im Beispiel in Abbildung 2.3 sind bei allen vier Attributen die Datentypen angegeben. Im Hinblick auf die Zielsprache Java wurde die in der UML standardmäßig übliche Form

<p>Klasse. Eine Klasse besteht aus einer Sammlung von Attributen und Methoden, die den Zustand und das Verhalten ihrer <i>Instanzen (Objekte)</i> festlegt. Klassen sind durch Assoziationen und Vererbungsbeziehungen miteinander verknüpft. Ein <i>Klassenname</i> erlaubt es, die Klasse zu identifizieren.</p> <p>Attribut. Die Zustandskomponenten einer Klasse werden als Attribute bezeichnet. Sie beinhalten grundsätzlich <i>Name</i> und <i>Typ</i>.</p> <p>Methode. Die Funktionalität einer Klasse ist in Methoden abgelegt. Eine Methode besteht aus einer <i>Signatur</i> und einem <i>Rumpf</i>, der die Implementierung beschreibt. Bei einer <i>abstrakten</i> Methode fehlt der Rumpf.</p> <p>Modifikator. Zur Festlegung von Sichtbarkeit, Instanzierbarkeit und Veränderbarkeit des modifizierten Elements können die Modifikatoren <code>public</code>, <code>protected</code>, <code>private</code>, <code>readonly</code>, <code>abstract</code>, <code>static</code> und <code>final</code> auf Klassen, Methoden, Rollen und Attribute angewandt werden. Für die ersten vier genannten Modifikatoren gibt es in UML/P die graphischen Varianten „+“, „#“ und „-“ und „?“.</p> <p>Konstanten sind als spezielle Attribute mit den Modifikatoren <code>static</code> und <code>final</code> definiert.</p> <p>Vererbung. Stehen zwei Klassen in Vererbungsbeziehung, so vererbt die <i>Oberklasse</i> ihre Attribute und Methoden an die <i>Unterklasse</i>. Die Unterklasse kann weitere Attribute und Methoden hinzufügen und Methoden <i>redefinieren</i> – soweit die Modifikatoren dies erlauben. Die Unterklasse bildet einen <i>Subtyp</i> der Oberklasse, der es nach dem <i>Substitutionsprinzip</i> erlaubt, Instanzen der Unterklasse dort einzusetzen, wo Instanzen der Oberklasse erforderlich sind.</p> <p>Interface. Ein Interface (Schnittstelle) beschreibt die Signaturen einer Sammlung von Methoden. Im Gegensatz zur Klasse werden keine Attribute (nur Konstanten) und keine Methodenrumpfe angegeben. Interfaces sind verwandt zu abstrakten Klassen und können untereinander ebenfalls in einer Vererbungsbeziehung stehen.</p> <p>Typ ist ein Basisdatentyp wie <code>int</code>, eine Klasse oder ein Interface.</p> <p>Interface-Implementierung ist eine der Vererbung ähnliche Beziehung zwischen einem Interface und einer Klasse. Eine Klasse kann beliebig viele Interfaces implementieren.</p> <p>Assoziation ist eine binäre Beziehung zwischen Klassen, die zur Realisierung struktureller Information verwendet wird. Eine Assoziation durch einen <i>Assoziationsnamen</i>, für jedes Ende einen <i>Rollenamen</i>, eine <i>Kardinalität</i> und eine Angabe über die <i>Navigationsrichtungen</i> beschrieben.</p> <p>Kardinalität. Die Kardinalität (Multiplicity, auch: Multiplizität) wird für jedes Assoziationsende angegeben. Sie ist von der Form „0..1“, „1“ oder „*“ und beschreibt, ob eine Assoziation in dieser Richtung optional oder eindeutig ist beziehungsweise mehrfache Bindung erlaubt.</p>

Abbildung 2.2. Begriffsdefinition für Klassendiagramme

„attribut: Typ“ durch die Java-konforme Fassung „Typ attribut“ ersetzt.

Für Attribute stehen mehrere *Modifikatoren* zur Verfügung, die die Attributeigenschaften genauer festlegen. UML stellt als kompakte Formen „+“ für

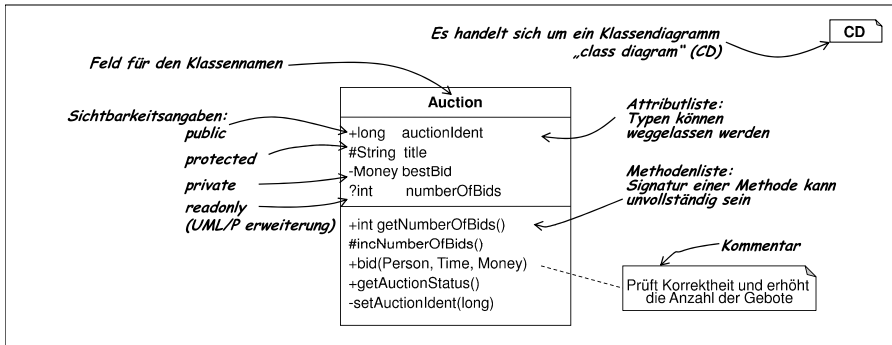


Abbildung 2.3. Klasse Auction im Klassendiagramm

`public`, `#` für `protected` und `-` für `private` zur Verfügung, um die Sichtbarkeit des Attributs für fremde Klassen zu beschreiben. `+` ermöglicht einen generellen Zugriff, `#` für Unterklassen und `-` erlaubt Zugriff nur innerhalb der definierenden Klasse. Nicht im UML-Standard enthalten ist eine vierte, nur von UML/P angebotene Sichtbarkeitsangabe `?`, die ein Attribut als *nur-lesbar* (*readonly*) markiert. Ein so markiertes Attribut ist frei lesbar, darf aber nur in Unterklassen und der Klasse selbst modifiziert werden. Diese Sichtbarkeitsangabe wirkt also beim Lesen wie `public` und bei der Modifikation wie `protected`. Sie erweist sich bei der Modellierung als hilfreich, um die Zugriffsrechte noch feiner zu beschreiben.

Weitere aus der Programmiersprache Java zur Verfügung stehende Modifikatoren, wie beispielsweise `static` und `final` zur Beschreibung statischer und nicht-modifizierbarer Attribute können im Klassendiagramm ebenfalls genutzt werden. In Kombination dienen diese Modifikatoren zur Definition von Konstanten, jedoch werden Konstanten in Klassendiagrammen häufig weggelassen. Ein mit `static` markiertes Attribut wird auch als *Klassenattribut* bezeichnet und kann wie in Abbildung 2.4 gezeigt alternativ durch einen Unterstrich gekennzeichnet werden.

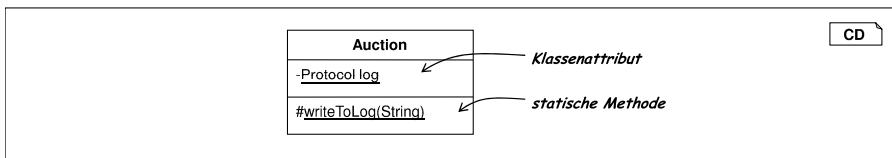


Abbildung 2.4. Klassenattribut und statische Methode

Die UML erlaubt *abgeleitete Attribute* die mit `..` markiert werden (siehe Abbildung 2.5). Bei einem abgeleiteten Attribut lässt sich sein Wert aus anderen Attributen desselben oder anderer Objekte sowie Assoziationen be-

rechnen („ableiten“). Die Berechnungsformel wird typischerweise in Form einer Bedingung `attr==...` definiert. UML/P sieht dafür die Verwendung der in Kapitel 3 eingeführten OCL vor.

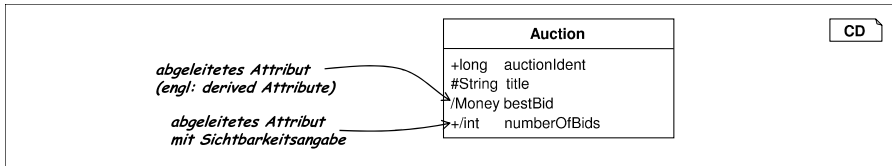


Abbildung 2.5. Abgeleitete Attribute

2.2.2 Methoden

Im dritten Feld einer Klassenrepräsentation werden Methoden mit Namen, Signaturen und ggf. Modifikatoren für Methoden dargestellt. Auch hier wird die Java-konforme Schreibweise `Typ methode (Parameter) : Typ` statt der offiziellen UML-Schreibweise `methode (Parameter) : Typ` verwendet. Während Attribute den Zustand eines Objekts speichern, dienen Methoden dazu, bestimmte Aufgaben zu erledigen und Daten zu berechnen. Sie nutzen dazu die in Attributen gespeicherten Daten und rufen andere Methoden des eigenen oder anderer Objekte auf. Wie Java bietet auch die UML/P Methoden mit variabler Stelligkeit, die zum Beispiel in der Form `Typ methode(Typ variable ...)` angegeben werden. Die Zugriffsrechte für Methoden können analog zu den Sichtbarkeiten für Attribute mit „+“, „#“ und „-“ gesteuert werden.

Weitere Modifikatoren für Methoden sind

- `static`, um die Methode auch ohne instanziiertes Objekt zugänglich zu machen,
- `final`, um die Methode für Unterklassen unveränderlich zu machen und
- `abstract`, um anzuzeigen, dass die Methode in dieser Klasse nicht implementiert ist.

Genau wie bei Klassenattributen wird es in der UML bevorzugt, statische Methoden alternativ durch Unterstreichung darzustellen. Konstruktoren werden wie statische Methoden in der Form `Klasse (Argumente)` dargestellt und unterstrichen. Beinhaltet eine Klasse eine abstrakte Methode, so ist die Klasse selbst als abstrakt zu definieren. Die Klasse kann dann keine Objekte als Instanzen ausprägen. In Unterklassen können jedoch die abstrakten Methoden einer Klasse geeignet implementiert werden.

2.2.3 Vererbung

Zur Strukturierung von Klassen in überschaubare Hierarchien kann die Vererbungsbeziehung eingesetzt werden. Existieren mehrere Klassen mit teilweise übereinstimmenden Attributen oder Methoden, so können diese in eine gemeinsame Oberklasse faktorisiert werden. Abbildung 2.6 demonstriert dies anhand der Gemeinsamkeiten mehrerer im Auktionssystem vorkommender Nachrichtenarten.

Stehen zwei Klassen in Vererbungsbeziehung, so vererbt die *Oberklasse* ihre Attribute und Methoden an die *Unterklasse*. Die Unterklasse kann die Liste der Attribute und Methoden erweitern sowie Methoden *umdefinieren* – soweit die Modifikatoren der Oberklasse dies erlauben. Gleichzeitig bildet die Unterklasse einen *Subtyp* der Oberklasse, der es nach dem *Substitutionsprinzip* erlaubt, Instanzen der Unterklasse dort einzusetzen, wo Instanzen der Oberklasse erforderlich sind.

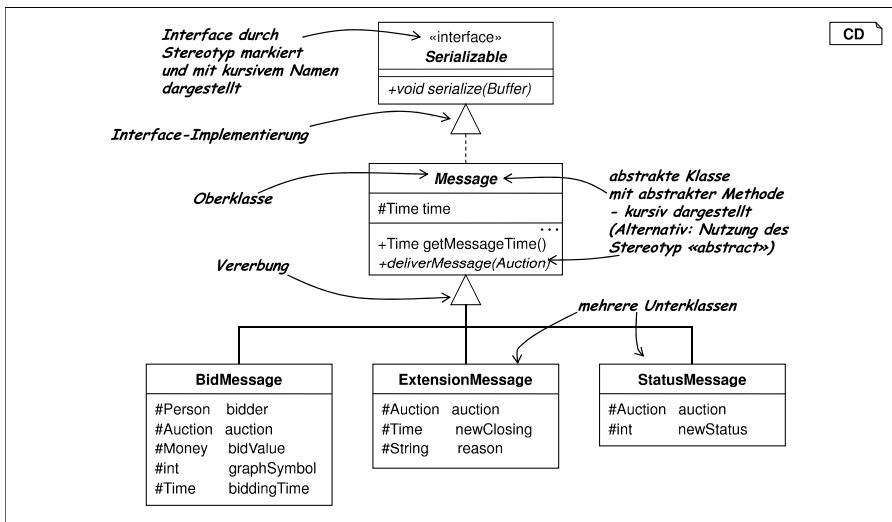


Abbildung 2.6. Vererbung und Interface-Implementierung

In Java erbt jede Klasse (bis auf `Object`) von genau einer Oberklasse. Jedoch kann eine Oberklasse viele Unterklassen besitzen, die ihrerseits weitere Unterklassen haben können. Durch die Nutzung der Vererbung als Strukturierungsmittel entsteht eine *Vererbungshierarchie*. Eine Oberklasse kann dabei als Verallgemeinerung (auch Generalisierung) ihrer Unterklassen angesehen werden, da ihre Attribute und Methodensignaturen die Gemeinsamkeiten aller Unterklassen festlegen. Steht in einer Vererbungshierarchie weniger die Codevererbung im Vordergrund, sondern die Strukturierung, so wird auch von *Generalisierungshierarchie* gesprochen. Insbesondere bei der

Erhebung von Anforderungen und beim Grobdesign spielt der Aspekt der Generalisierung bei der Systemstrukturierung eine wesentliche Rolle.

Vererbung ist ein wesentliches Strukturierungsmittel objektorientierter Modellierung. Dennoch sollten tiefe Vererbungshierarchien vermieden werden, da sie die in der Vererbungsbeziehung stehenden Klassen und damit den darin enthaltenen Code stark koppeln. Zum Verständnis einer Unterklasse müssen sowohl die direkte als auch alle darüber liegenden Oberklassen verstanden werden.

2.2.4 Interfaces

Java bietet eine Spezialform der Klasse an, das *Interface*. Ein Interface besteht aus einer Menge von Methodensignaturen und Konstanten und wird vor allem zur Definition einer Schnittstelle zwischen Systemteilen (Komponenten) eingesetzt. In Abbildung 2.6 wird das Interface `Serializable` benutzt, um eine bestimmte Funktionalität von allen Klassen zu fordern, die dieses Interface implementieren.

Ein Interface wird wie eine Klasse durch ein Rechteck dargestellt und mit dem Stereotyp `«interface»` markiert. Genauso wie von einer abstrakten Klasse können von einem Interface nicht direkt Objekte instanziiert werden. Stattdessen müssen die angegebenen Methodensignaturen in Klassen realisiert werden, die das Interface implementieren. Auch können Interfaces außer Konstanten keine Attribute beinhalten.

Während in Java eine Klasse nur von einer Oberklasse erben darf, kann sie beliebig viele Interfaces *implementieren*. Ein Interface kann auch andere Interfaces erweitern und so in einer *Subtyp-Beziehung* zu den erweiterten Interfaces stehen. Dabei bindet das *Subinterface* die vom *Superinterface* definierten Methodensignaturen in die eigene Definition und erweitert diese, wie in Abbildung 2.7 an einem Ausschnitt aus der Java-Klassenbibliothek gezeigt, um zusätzliche Methoden.

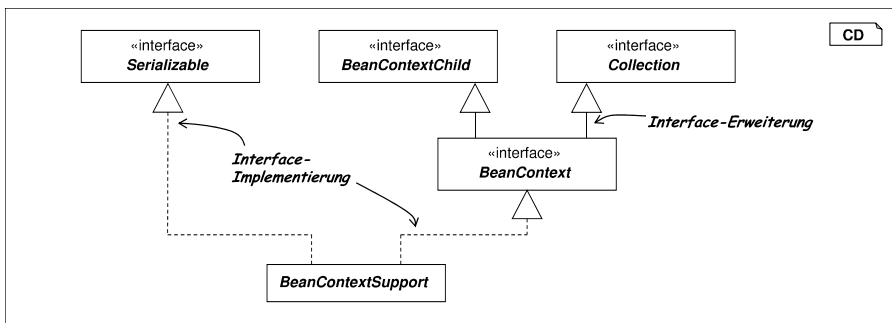


Abbildung 2.7. Interface-Implementierung und -Erweiterung

Eine Assoziation ist genauso wie eine Klasse ein Modellierungskonzept im Klassendiagramm. Zur Laufzeit eines Systems manifestiert sich eine Assoziation durch *Links* zwischen den damit verbundenen Objekten. Die Anzahl der Links wird durch die Kardinalität der Assoziation eingeschränkt. Ist eine Assoziation in eine Richtung navigierbar, so werden in der Implementierung Vorkehrungen getroffen diese Navigierbarkeit effizient zu realisieren.

2.3.1 Rollen

Mithilfe des Rollennamens lassen sich die über eine Assoziation beziehungsweise deren Links verbundenen Objekte ansprechen. So kann aus einem Objekt der Klasse `Person` mithilfe des Rollennamens `auctions` auf die Auktionen zugegriffen werden, an der die Person teilnimmt. Ist kein expliziter Rollenname gegeben, so wird ersatzweise der Name der Assoziation oder der Zielklasse als Rollenname angenommen, wenn diese die intendierte Navigation eindeutig beschreiben. Im Beispiel 2.8 kann von einem Objekt der Klasse `Auction` mit den Namen `biddingPolicy` und `messages` auf die entsprechenden Objekte zugegriffen werden. Dabei werden gemäß der zur Implementierung genutzten Programmiersprache und der im Klassendiagramm angegebenen Kardinalitäten schematische Umsetzungen des ersten Buchstabens im Namen vorgenommen. So beginnen in UML/P Rollennamen grundsätzlich mit einem Kleinbuchstaben, während Klassennamen mit einem Großbuchstaben anfangen.

Sind beide Assoziationsenden mit derselben Klasse verbunden, so wird von einer *reflexiven Assoziation* gesprochen. Reflexive Assoziationen erlauben die Realisierung einer Reihe von Entwurfsmustern [GHJV94, BMR⁺96], wie zum Beispiel eine Teile-Ganzes-Beziehung. In einer reflexiven Assoziation ist es notwendig, wenigstens eines der Enden mit Rollennamen zu versehen. So kann eine Unterscheidung der teilnehmenden Objekte durch ihre Rollen vorgenommen werden.

Abbildung 2.10 zeigt eine reflexive Assoziation, in der einem Beobachter der Bieter zugeordnet wird, den er „beobachten“ darf. Obwohl damit eine reflexive Klassenstruktur geschaffen wurde, ist im Beispiel die Rekursionstiefe auf 1 beschränkt. Bieter sind selbst keine Beobachter und Beobachter haben eine direkte Verbindung zu Bieter. Dies kann durch geeignete OCL-Bedingungen ausgedrückt werden (siehe Kapitel 3).

2.3.2 Navigation

In einem entwurfsnahen oder für die Implementierung gedachten Klassendiagramm spielen die erlaubten Richtungen zur Navigation entlang einer Assoziation eine wesentliche Rolle. Im Beispiel 2.8 ist beschrieben, dass von einem Personen-Objekt der Zugriff auf die ihm zugeordneten Message-Objekte möglich ist. Umgekehrt ist es nicht vorgesehen, von einem Message-Objekt (direkt) auf die Personen zuzugreifen, denen es zugestellt wurde.

Das Modell erlaubt damit die Verteilung einer Nachricht, zum Beispiel im Broadcast-Verfahren, an mehrere Personen ohne Duplikation.

Assoziationen können grundsätzlich uni- oder bidirektional sein. Ist keine explizite Pfeilrichtung angegeben, so wird von einer bidirektionalen Assoziation ausgegangen. Formal werden die Navigationsmöglichkeiten in dieser Situation als unspezifiziert und damit nicht einschränkend betrachtet.

Ist die grundsätzliche Navigierbarkeit durch den Pfeil modelliert, so wird durch den Rollennamen festgelegt, wie die Assoziation bzw. die auf der anderen Seite liegenden Objekte angesprochen werden können. Die Modifikatoren `public`, `protected` und `private` können für Rollen eingesetzt werden, um die Sichtbarkeit dieser Navigation entsprechend einzuschränken.

2.3.3 Kardinalität

Für jedes Ende einer Assoziation kann eine Kardinalität angegeben werden. Die Assoziation `participants` lässt zum Beispiel zu, dass eine Person in mehreren Auktionen teilnimmt und in einer Auktion mehrere Personen bieten können. Einer Auktion ist jedoch nur genau eine `TimingPolicy` zugeordnet. Die drei Kardinalitätsangaben „*“, „1“ und „0..1“ erlauben wie in Abbildung 2.8 zu sehen die Zuordnung von *beliebig vielen*, *genau einem* beziehungsweise *maximal einem* Objekt. Allgemein sind Kardinalitäten von der Form $m..n$ oder $m..*$ und konnten in den früheren UML 1.x Varianten sogar kombiniert werden (Beispiel $3..7, 9, 11..*$). In einer Implementierung sind jedoch vor allem die drei zuerst genannten Kardinalitätsformen direkt umsetzbar und daher von Interesse, weshalb auf eine Behandlung der allgemeinen Kardinalitätsform hier verzichtet wird. In Kapitel 3 wird mit den OCL-Invarianten ein Mechanismus eingeführt, der es erlaubt allgemeine Kardinalitäten zu beschreiben und methodisch einzusetzen.

In der UML Literatur wird manchmal zwischen *Kardinalität* und *Multiplizität* unterschieden. Dann bezeichnet die Kardinalität die Anzahl der tatsächlichen Links einer Assoziation, während die Multiplizität den Bereich möglicher Kardinalitäten angibt. Die ER-Modelle unterscheiden nicht und verwenden einheitlich den Begriff Kardinalität.

2.3.4 Komposition

Eine besondere Form der Assoziation ist die *Komposition*. Sie wird durch eine ausgefüllte Raute an einem Assoziationsende dargestellt. In einer Komposition sind die Teilobjekte stark abhängig vom *Ganzen*. Im Beispiel 2.8 sind `BiddingPolicy` und `TimingPolicy` in ihrem Lebenszyklus von dem Auktionsobjekt abhängig. Das heißt, Objekte dieser Typen werden gemeinsam mit dem Auktionsobjekt erzeugt und an dessen Lebensende obsolet. Da es sich bei `BiddingPolicy` und `TimingPolicy` um Interfaces handelt, werden stattdessen geeignete Objekte verwendet, die diese Interfaces implementieren.

Eine alternative Darstellungsform stellt den Kompositionscharakter einer Kompositionsassoziation stärker in den Vordergrund, indem sie statt einer Raute graphisches Enthaltensein nutzt. Abbildung 2.9 zeigt zwei Alternativen, die sich nur in Details unterscheiden. Im Klassendiagramm (a) ist der Assoziationscharakter der Komposition herausgestellt. Er beschreibt auch Navigationsmöglichkeiten. Im Klassendiagramm (b) sind Navigationsrichtungen nicht direkt angegeben, jedoch beide Klassen mit je einem *Rollenname* versehen, der beschreibt, wie ein Zugriff vom enthaltenden Auktions-Objekt auf seine Komponenten möglich ist. Die Kardinalität ist in der Klasse rechts oben angegeben. Die Darstellung (b) wirkt einerseits intuitiver, ist andererseits aber weniger aussagekräftig. So ist es darin weder möglich die Rückrichtung der Navigation zu klären, noch der darin vorhandenen Kompositionsassoziation weitere Merkmale hinzuzufügen. Die Kardinalität auf Kompositionsseite ist standardmäßig „1“ kann aber durch „0..1“ angepasst werden. Das heißt, ein Objekt ist maximal einem Kompositum zugeordnet.

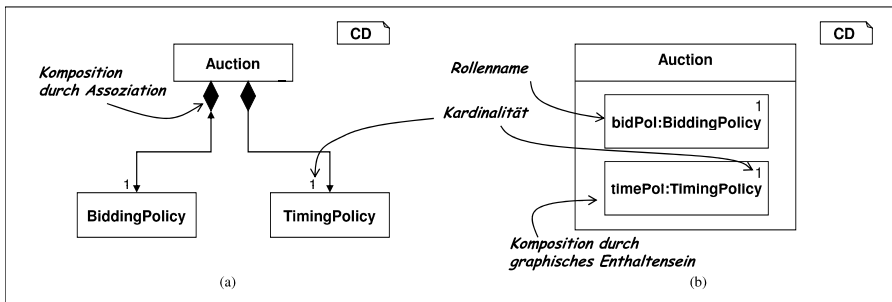


Abbildung 2.9. Alternative Darstellungen von Komposition

Bezüglich der Austauschbarkeit und des Lebenszyklus von abhängigen Objekten in einem Kompositum gibt es erhebliche Interpretationsunterschiede². Eine präzise Definition der Bedeutung einer Komposition sollte daher jeweils projektspezifisch festgelegt werden. Dies kann zum Beispiel durch die in Abschnitt 2.5 eingeführten Stereotypen erfolgen, durch ergänzende projekt- oder unternehmensspezifische, informelle Erläuterungen präzisiert oder durch selbst definierte Stereotypen festgelegt werden.

2.3.5 Abgeleitete Assoziationen

Neben abgeleiteten Attributen gibt es in UML/P auch abgeleitete Assoziationen. Deren Namen wird ebenfalls die Marke „/“ vorangestellt. Eine Assoziation gilt als abgeleitet, wenn sich die Menge ihrer Links aus anderen

² Eine ausführlichere Diskussion dieser Thematik ist zum Beispiel in [HSB99] und [Bre01] enthalten.

Zustandselementen berechnen („ableiten“) lässt. Zur Berechnung können andere Attribute und Assoziationen herangezogen werden. Im Beispiel in Abbildung 2.10 sind zwei Assoziationen gegeben, die beschreiben, welche Personen in einer Auktion bieten und welche Personen das Verhalten eines bietenden Kollegen („fellow“) beobachten dürfen. Die abgeleitete Assoziation `/observers` berechnet sich aus diesen beiden Assoziationen. Dazu kann zum Beispiel die ebenfalls in Abbildung 2.10 angegebene Charakterisierung mittels einer OCL-Bedingung (siehe Kap. 3) verwendet werden.

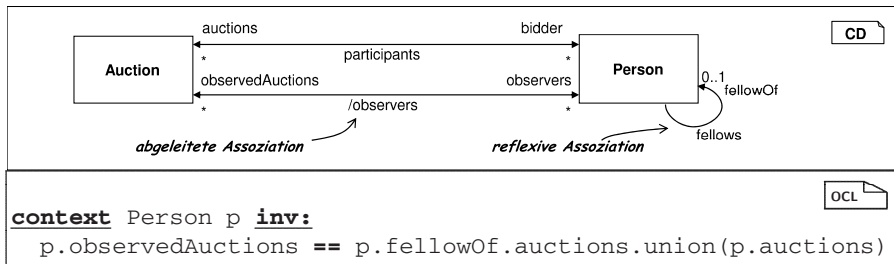


Abbildung 2.10. Abgeleitete Assoziation

2.3.6 Assoziationsmerkmale

Der UML-Standard bietet eine Reihe zusätzlicher Merkmale für Assoziationen an, die Assoziationseigenschaften genauer regeln. Abbildung 2.11 beinhaltet drei in diesem Kontext interessante Merkmale. Mit `{ordered}` wird angezeigt, dass eine mit der Kardinalität „*“ versehene Assoziation einen geordneten Zugriff erlaubt. In diesem Fall ist die Reihenfolge der Nachrichten innerhalb einer Auktion relevant. Mit `{frozen}` wird angezeigt, dass nach Ende der Initialisierung eines Auktionsobjekts die beiden Assoziationen zu den Policy-Objekten nicht mehr verändert werden. Dadurch stehen während der gesamten Lebenszeit eines Auktionsobjekts dieselben Policies zur Verfügung. Mit `{addOnly}` wird modelliert, dass in der Assoziation nur Objekte hinzugefügt werden dürfen, das Entfernen jedoch verboten ist. Im Modell in Abbildung 2.11 wird damit ausgedrückt, dass Nachrichten, die in einer Auktion versandt worden sind, nicht mehr zurückgenommen werden können.

Eine Assoziation, bei der ein Ende mit dem Merkmal `{ordered}` gekennzeichnet ist, muss natürlich einen Mechanismus anbieten, der einen Zugriff gemäß dieser Ordnung erlaubt. Assoziationen mit dem Merkmal `{ordered}` stellen einen Spezialfall einer qualifizierten Assoziation dar.

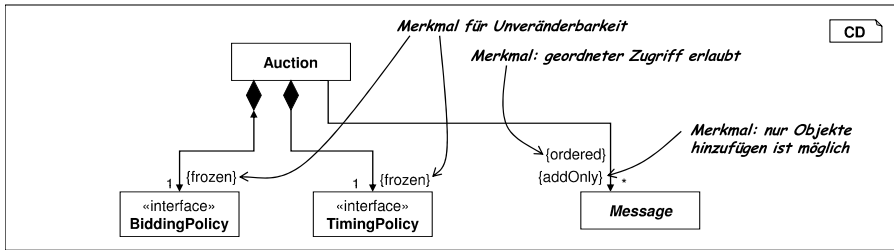


Abbildung 2.11. Merkmale für Assoziationen

2.3.7 Qualifizierte Assoziation

In der allgemeinsten Form bietet die *qualifizierte Assoziation* die Möglichkeit, aus einer Menge von zugeordneten Objekten mithilfe des *Qualifikators* ein einzelnes Objekt zu selektieren. Abbildung 2.12 zeigt mehrere auf unterschiedliche Weise qualifizierte Assoziationen. Zusätzlich ist es möglich, eine Komposition zu qualifizieren und die Qualifikation einer Assoziation bei beiden Enden vorzunehmen.

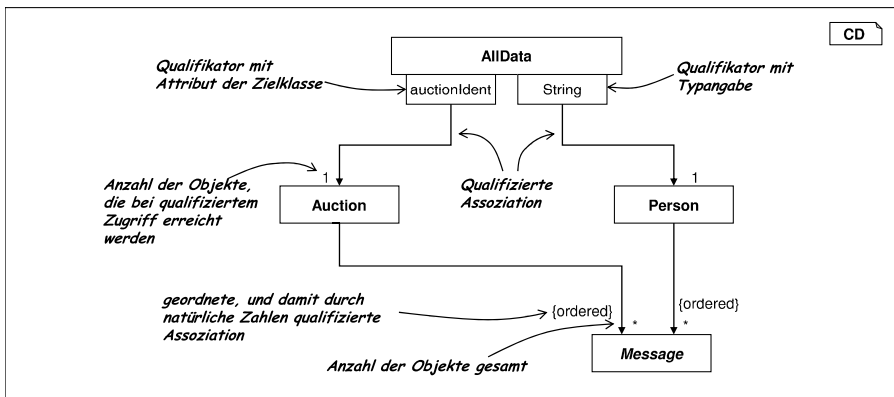


Abbildung 2.12. Qualifizierte Assoziationen

Im Auktionssystem wird ein Objekt der Klasse `AllData` genutzt, um die Sammlung aller aktuell geladenen Auktionen und der daran teilnehmenden Personen zu speichern. Der qualifizierte Zugriff über den Auktionsidentifikator `auctionIdent` wird als Abbildung verstanden und deshalb die Funktionalität des Interface `Map<long, Auction>` zur Verfügung gestellt. In welcher Form die qualifizierte Assoziation implementiert wird, ist damit allerdings noch nicht festgelegt. Bei einer Codegenerierung kann eine Transformation in eine alternative Datenstruktur stattfinden oder weitere Funktionalität beispielsweise der `NavigableMap` hinzugefügt werden.

Als Schlüssel für die Abbildung werden im Beispiel die bereits in der Auktion vorhandenen Auktionsidentifikatoren verwendet. In analoger Form können Personen über den als Typ `String` gegebenen Namen selektiert werden. Es ist jedoch ein signifikanter Unterschied, ob als Qualifikator ein Typ (im Beispiel: `String`) oder ein Attributname der gegenüberliegenden Klasse angegeben ist. Während im ersten Fall als Schlüssel beliebige Objekte beziehungsweise Werte des angegebenen Typs verwendet werden können, ist im zweiten Fall als Qualifikator nur der tatsächliche Attributinhalt zulässig. Mit der in Kapitel 3 eingeführten OCL kann diese Eigenschaft wie folgt formuliert werden:

```
context AllData ad inv:
  forall k in long:
    auction.containsKey(k) implies
      auction.get(k).auctionIdent == k
```



Während in explizit qualifizierten Assoziationen die Art des Qualifikators wählbar ist, stehen in geordneten Assoziationen ganzzahlige Intervalle beginnend mit der 0 zur Verfügung.³

Wird ein expliziter Qualifikator verwendet, so wird durch den qualifizierten Zugriff nur ein Objekt erreicht, auch wenn eine andere Kardinalität angegeben ist. Das Zielobjekt wird normalerweise nur in Bezug auf das Ausgangsobjekt und den Qualifikator eindeutig identifiziert. Beispielsweise können bei Index 0 für jede Auktion unterschiedliche Nachrichten gespeichert sein. Der Qualifikator `allIdent` ist nur deshalb Systemweit eindeutig, weil er gleichzeitig einen eindeutigen Schlüssel des Zielobjekts darstellt. Sollte der Qualifikator nicht besetzt sein, so muss geeignet reagiert werden, zum Beispiel durch einen `null`-Pointer wie bei den Java Maps oder eine Exception. In einer geordneten Assoziation (Merkmal `{ordered}`) bleibt der Qualifikator implizit, denn er wird nicht in einer entsprechenden Box am Assoziationsende angegeben.

2.4 Sicht und Repräsentation

Klassendiagramme haben im Allgemeinen das Ziel, die für eine bestimmte Aufgabe notwendige Struktur einschließlich ihrer Zusammenhänge zu beschreiben. Eine vollständige Liste aller Methoden und Attribute ist in diesem Fall meist hinderlich. Stattdessen sollten die Methoden und Attribute dargestellt werden, die für die Darstellung der „Story“ hilfreich sind. „Story“ ist eine Metapher, die gerne verwendet wird, um anzuzeigen, dass ein Diagramm einen Fokus hat, der wesentliche Information hervorhebt und unwesentliche Information weglässt. Diagramme können beispielsweise unterschiedliche Teilsysteme oder einzelne Systemfunktionen modellieren.

³ Die Indizierung beginnt, wie in Java üblich, mit 0.

Ein Klassendiagramm stellt daher meist eine unvollständige *Sicht* des Gesamtsystems dar. So können einzelne Klassen oder Assoziationen fehlen. Innerhalb der Klassen können Attribute und Methoden weggelassen oder unvollständig dargestellt werden. Bei Methoden können beispielsweise die Argumentliste und der Ergebnistyp fehlen.

Leider ist einem UML-Diagramm im Allgemeinen nicht anzusehen, ob die darin enthaltene Information vollständig ist. Deshalb wurde neben dem von der UML bereits angebotenen Repräsentationsindikator „...“ zur Markierung unvollständiger Information auch „©“ zur Darstellung vollständiger Information aus [FPR01] übernommen.

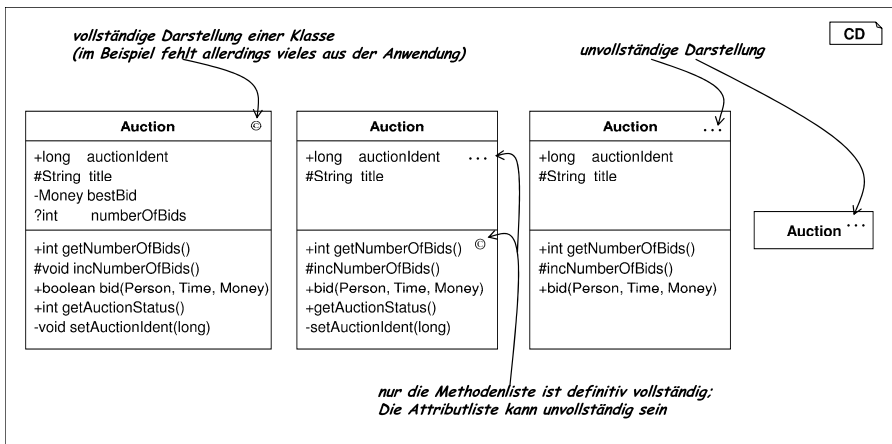


Abbildung 2.13. Vollständige Klassendarstellung

Abbildung 2.13 zeigt, wie die beiden Indikatoren „...“ und „©“ eingesetzt werden können. Die Indikatoren „©“ und „...“ wirken nicht auf die Klasse selbst, sondern auf ihre Darstellung innerhalb des Klassendiagramms. Ein „©“ beim Klassennamen besagt, dass sowohl die Attribut- als auch die Methodenliste vollständig ist. Demgegenüber bedeutet der Unvollständigkeits-Indikator „...“, dass die Darstellung unvollständig sein *kann*. Aufgrund des nachfolgend diskutierten Dualismus zwischen Assoziationen und Attributen wird bei einer Vollständigkeit der Attributliste impliziert, dass gleichzeitig alle Assoziationen, die von dieser Klasse aus navigiert werden können, angezeigt sind.

Beide Indikatoren können auch einzeln auf die Liste der Attribute und die Liste der Methoden angewandt werden. Der Unvollständigkeits-Indikator „...“ wird bei Fehlen eines expliziten Indikators standardmäßig angenommen. Dies entspricht der oft üblichen Sichtweise, dass ein Diagramm eine Abstraktion des Systems darstellt.

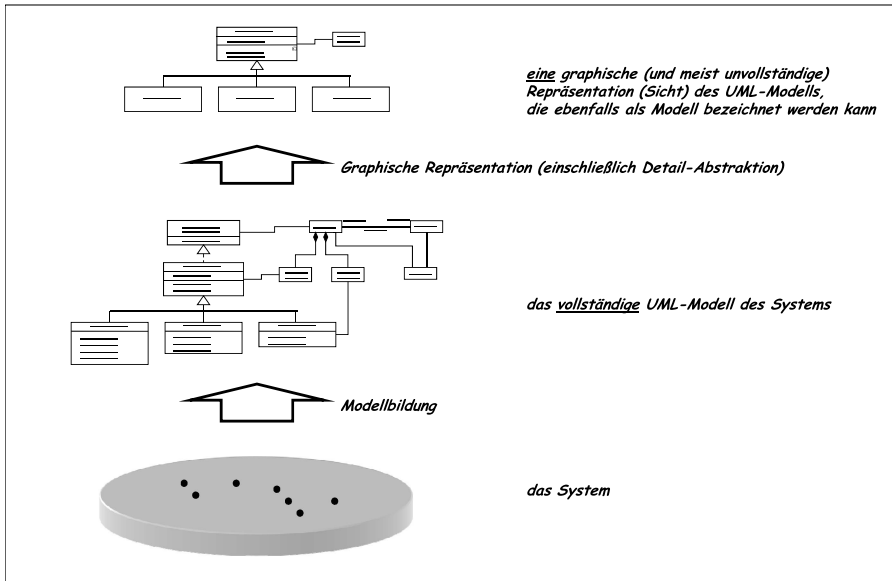


Abbildung 2.14. Illustration der drei Modellebenen

Um die Verwendung dieser Indikatoren präzise zu erklären, sind die in Abbildung 2.14 illustrierten drei Modellebenen zu unterscheiden: das System selbst, das vollständige UML-Modell des Systems und eine graphische, unvollständige *Repräsentation* des Modells beispielsweise auf dem Bildschirm. Diese Repräsentation wird meist auch *Sicht* genannt, ist aber selbst ein Modell. Die beiden Indikatoren „©“ und „...“ beschreiben eine Beziehung zwischen der Sicht und dem vollständigen UML-Modell, auf dem die Sicht beruht. Wie Abbildung 2.13 zeigt, gibt es eine Reihe verschiedener Sichten für dieselbe Klasse. Die beiden Markierungen werden deshalb *Repräsentationsindikatoren* genannt.

Erreicht ein Softwaresystem eine bestimmte Größe, so kann ein vollständiges Klassendiagramm sehr überladen wirken. Aufgrund der vielen Details wird dann die Story eher verborgen als präsentiert. Deshalb ist es sinnvoll, bei der Softwareentwicklung mit mehreren kleineren Klassendiagrammen zu arbeiten. Notwendigerweise haben Klassendiagramme, die jeweils einen Ausschnitt des Systems zeigen, Überlappungen. Durch solche Überlappungen wird der Zusammenhang zwischen den einzelnen Modellen hergestellt. Abbildung 2.15 stellt zwei Ausschnitte des Auktionssystems dar, in der die Klasse *Auction* in unterschiedlicher Form repräsentiert ist. Der Indikator „...“ erlaubt hierbei explizit zu beschreiben, dass nur die für die jeweilige Story notwendige Information dargestellt wird.

Aus einer Sammlung einzelner Klassendiagramme kann ein vollständiges Klassendiagramm durch *Verschmelzung* gewonnen werden. Wie in Ab-

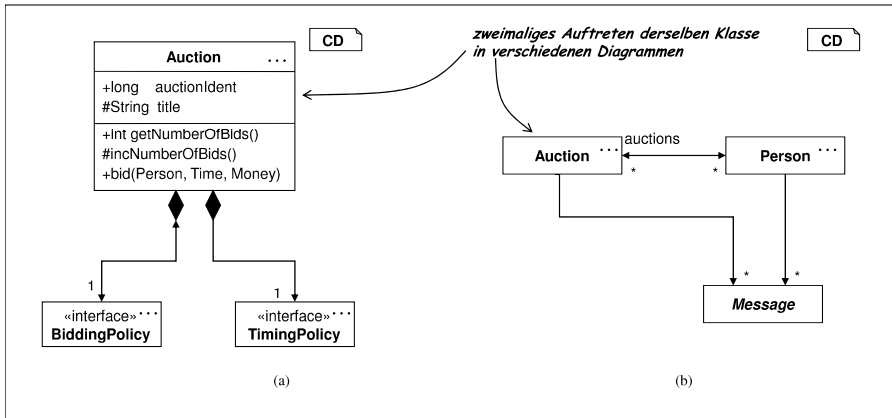


Abbildung 2.15. Überlappung von Klassendiagrammen

schnitt 1.4.4 diskutiert, wird in einigen Werkzeugen intern immer ein vollständiges Klassendiagramm als Modell eingesetzt und dem Nutzer Ausschnitte in Form von Sichten dargestellt. Eine Verschmelzung von Klassendiagrammen ist im Wesentlichen eine Vereinigung aller Klassen-, Assoziations- und Vererbungsbeziehungen, wobei bei einer mehrfach auftretenden Klasse Attribut- und Methodenlisten ebenfalls vereinigt werden. Natürlich gibt es eine Reihe von Konsistenzbedingungen, die zu beachten sind. Dazu gehören übereinstimmende Typisierungsangaben bei Attributen und Methoden, kompatible Navigationsrollen und Kapazitätsangaben bei Assoziationen sowie die Vermeidung zyklischer Vererbungsbeziehungen.

2.5 Stereotypen und Merkmale

Obwohl die UML als graphische Sprache für den Einsatz bei der Softwareentwicklung gedacht ist, hat sie eine Reihe von Eigenschaften mit natürlichen Sprachen gemeinsam. Die UML besitzt ein Grundgerüst, das einer Grammatik entspricht und dessen Aussagen in Form von Diagrammen gebildet werden. Ähnlich den natürlichen Sprachen gibt es Mechanismen, um das Sprachvokabular entsprechend den jeweils erforderlichen Bedürfnissen zu erweitern und anzupassen. Diese Mechanismen bilden die Grundlage für projekt- und unternehmensspezifische Dialekte bzw. Profile der UML.

Bereits die Einführung einer neuen Klasse erweitert das zur Verfügung stehende Vokabular, denn es erlaubt, diese Klasse an anderen Stellen zu nutzen. Von diesem Blickwinkel aus gesehen ist die Tätigkeit des Programmierens eine stetige Erweiterung des im System zur Verfügung stehenden Vokabulars. Während jedoch in einer Programmiersprache die Einführung einer neuen Kontrollstruktur nicht möglich ist, erlaubt die UML in restringierter

Form die Einführung neuer Arten von Modellelementen, indem sie *Stereotypen* und *Merkmale* anbietet, mit denen vorhandene Modellelemente spezialisiert und angepasst werden können (siehe Abbildung 2.16).

Ohne einen expliziten Mechanismus dafür anzugeben, erlaubt die UML einerseits das syntaktische Aussehen durch Einschränkung oder Erweiterung der Sprache zu modifizieren andererseits aber auch die Bedeutung der Sprache zu verändern. Da die UML als „universelle“ Sprache konzipiert ist, ist je nach Einsatzform eine gewisse Bandbreite ihrer Bedeutung sinnvoll. Sogenannte „*semantische Variabilität*“ [Grö10] erlaubt die projektspezifische Anpassung der Bedeutung und der Werkzeuge. „Semantische Variationspunkte“ sind in der Standard-UML selbst nicht beschreibbar, weshalb in [GRR10, CGR09] dafür ein eigenständiger Mechanismus auf Basis von Featurediagrammen definiert ist und zur Profilbildung eingesetzt werden kann. Die allgemein möglichen Veränderungen gehen weit über das hier vorgestellte Konzept der Stereotypen und Merkmale hinaus.

<p>Stereotyp. Ein Stereotyp klassifiziert Modellelemente wie beispielsweise Klassen oder Attribute. Durch einen Stereotyp wird die Bedeutung des Modellelements spezialisiert und kann so beispielsweise bei der Codegenerierung spezifischer behandelt werden. Ein Stereotyp kann eine Menge von Merkmalen besitzen.</p> <p>Merkmal. Ein Merkmal beschreibt eine Eigenschaft eines Modellelements. Ein Merkmal wird notiert als Paar bestehend aus <i>Schlüsselwort</i> und <i>Wert</i>. Mehrere solche Paare können zu einer Komma-separierten Liste zusammengefasst werden.</p> <p>Modellelemente sind die (wesentlichen) Bausteine der UML-Diagramme. Beispielsweise hat das Klassendiagramm als Modellelemente Klassen, Interfaces, Attribute, Methoden, Vererbungsbeziehungen und Assoziationen. Merkmale und Stereotypen können auf Modellelemente angewandt werden, sind aber selbst keine Modellelemente.</p>

Abbildung 2.16. Begriffsdefinition Merkmal und Stereotyp

In den vorangegangenen Beispielen dieses Kapitels wurden bereits vereinzelt *Stereotypen*, *Merkmale*⁴ und verwandte Mechanismen verwendet. In der in Abbildung 2.3 dargestellten Klasse wurden die Sichtbarkeitsangaben „+“, „#“, „?“ und „-“ eingeführt. Abbildung 2.13 zeigt die beiden Repräsentationsindikatoren „©“ und „. .“, die sich auf die Repräsentation einer Sicht des Modells beziehen. Abbildung 2.6 zeigt den Stereotyp «interface», der eine „besondere“ Klasse, nämlich ein Interface kennzeichnet. Die Merkmale {ordered}, {frozen} und {addOnly} dienen schließlich zur Kennzeichnung von Assoziationsenden, wie das Beispiel 2.11 zeigt.

⁴ In der englischen UML-Definition wird von *tagged values* und *properties* gesprochen, die unter anderem in [Bal99] mit *Merkmal* übersetzt werden.

2.5.1 Stereotypen

Abbildung 2.17 zeigt drei Arten von Stereotypen. Während der Stereotyp «interface» standardmäßig von der UML zur Verfügung gestellt wird, sind die beiden rechten Stereotypen «JavaBean» und «Message» in einem Projekt, Werkzeug oder Framework selbst zu definieren.

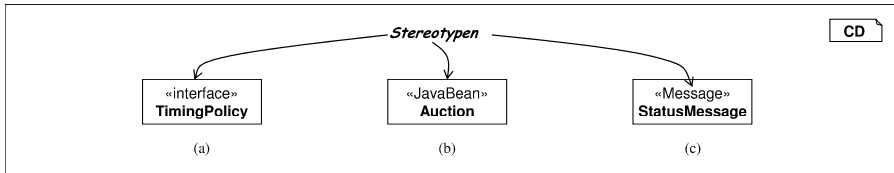


Abbildung 2.17. Arten von Stereotypen für Klassen

Stereotypen werden standardmäßig in französischen Anführungszeichen (Guillemots) mit umgekehrten Spitzen angegeben. Im Prinzip kann jedes UML-Modellelement mit einem oder mehreren Stereotypen versehen werden. Meistens jedoch werden Stereotypen für Klassen verwendet, um ihnen spezialisierte Eigenschaften zuzuordnen.

Der Stereotyp «interface» markiert ein Interface, das als Sonderform der Klasse gilt. Der Stereotyp «JavaBean» wirkt als Indikator dafür, dass die markierte Klasse die von JavaBeans geforderte Funktionalität zur Verfügung stellt. Der Stereotyp «Message» wird im Auktionsprojekt verwendet, um in kompakter Form festzuhalten, dass die markierte Klasse eine Unterklasse von Message ist und damit der Nachrichtenübermittlung dient.

Stereotypen haben also eine Reihe verschiedener Anwendungsmöglichkeiten. Sie können Modellelemente klassifizieren, um ihnen beispielsweise zusätzliche Eigenschaften oder Funktionalitäten zuzuordnen oder Einschränkungen aufzuerlegen. Der UML-Standard bietet einen Metamodellbasierten und einen tabellarischen Ansatz zur informellen Definition von Stereotypen. Je nach Intention eines Stereotyps können einschränkende Bedingungen jedoch auch präziser formuliert oder nur Mechanismen für eine spezifische Umsetzung in Code angegeben werden. Folgende Liste zeigt einige der Verwendungsmöglichkeiten für Stereotypen:

- Ein Stereotyp beschreibt die syntaktischen Eigenschaften eines Modellelements, indem es zusätzliche Eigenschaften fordert oder vorhandene Eigenschaften spezialisiert.
- Ein Stereotyp kann die Form der Repräsentation eines Modells in der dem Benutzer zur Verfügung gestellten Sicht beschreiben. Der Indikator „©“ kann als eine solche spezielle Form angesehen werden.
- Ein Stereotyp kann anwendungsspezifische Anforderungen beschreiben. Zum Beispiel kann «persistent» charakterisieren, dass die Objekte dieser

Klasse speicherbar sind. Jedoch wird nicht beschrieben, wie diese Speicherung vorzunehmen ist.

- Ein Stereotyp kann eine methodische Beziehung zwischen Modellelementen beschreiben. Dazu ist beispielsweise der im UML-Standard definierte Stereotyp «refine» gedacht.
- Ein Stereotyp kann die Intention des Modellierers widerspiegeln, die beschreibt, wie ein Programmierer ein Modellelement einsetzen sollte. Zum Beispiel kann eine Klasse als «adaptiv» markiert sein, um anzuzeigen, dass es sich um einen guten Kandidaten für eine Erweiterung handelt. Solche Stereotypen sind speziell für Frameworks geeignet (siehe [FPR01]). Mit Stereotypen der Form «Wrapper» kann beispielsweise die Rolle einer Klasse in einem Entwurfsmuster dokumentiert werden.

Natürlich gibt es eine Reihe von Überlappungen zwischen den genannten und weiteren Anwendungsmöglichkeiten für Stereotypen.

Für eine weitere Detaillierung von Eigenschaften kann ein Stereotyp mit einer Menge von Merkmalen versehen sein. Die Anwendung eines Stereotyps auf ein Modellelement impliziert dann, dass die ihm zugeordneten Merkmale auf dem Modellelement ebenfalls definiert sind.

2.5.2 Merkmale

Abbildung 2.18 zeigt eine durch einen entsprechenden Stereotypen markierte Testklasse des Auktionssystems, bei der Informationen über den dargestellten Test und seine Ausführung in Form von Merkmalen gespeichert werden.

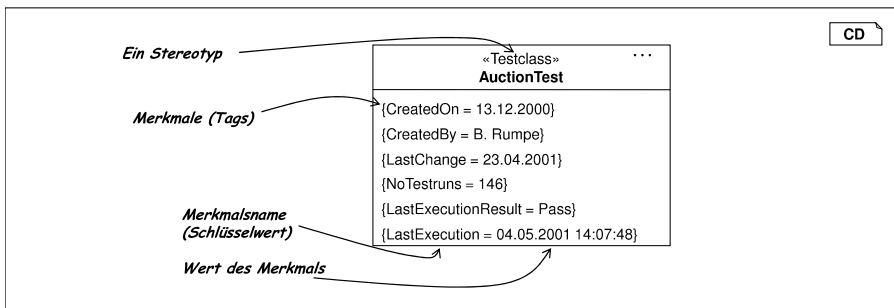


Abbildung 2.18. Merkmale für eine Testklasse

Merkmale können grundsätzlich an jedes Modellelement angefügt werden. Zusätzlich können Merkmale, wie in Abbildung 2.18 gezeigt, an einen Stereotyp gebunden und dadurch gemeinsam mit dem Stereotyp bei Modellelementen angewandt werden. Im Anwendungsbeispiel wird durch die

Verwendung des Stereotyps «Testclass» die Existenz der letzten drei Merkmale gefordert, da im Auktionsprojekt diese drei Merkmale dem Stereotyp zugeordnet sind.

Ein Merkmal wird normalerweise in der Form `{name = wert}` notiert. Als Wertebereiche stehen grundsätzlich Zeichenreihen und Zahlen zur Verfügung. Eine explizite Typisierung der Merkmalswerte wäre wünschenswert, wird aber heute weder durch den UML-Sprachstandard noch durch Werkzeuge unterstützt. Spielt der Wert keine Rolle, oder handelt es sich um den booleschen Wert **true**, so kann dieser auch weggelassen werden. Zum Beispiel sind `{sind.Tests_OK = true }` und `{sind.Tests_OK}` zwei alternative Darstellungen. Der UML-Standard [OMG10a] bietet Merkmale wie `{ordered}` für Assoziationen standardmäßig an, erlaubt aber auch situationspezifisch neue Merkmale zu definieren.

Auch wenn ein Merkmal einer Klasse zugeordnet wurde, unterscheidet es sich grundlegend von einem Attribut. Ein Merkmal ordnet dem Modell-element eine Eigenschaft zu, während ein Attribut bei jeder Instanz einer Klasse einen eigenständigen Wert hat. Attribute treten zur „Laufzeit“ eines Systems auf, während Merkmale dort nicht existieren. Dennoch können Merkmale Auswirkungen auf das System haben, wenn sie Eigenschaften des Modellelements in Bezug auf dessen Implementierung beeinflussen. Merkmale eignen sich unter anderem zur Darstellung folgender Eigenschaften:

- Der Initialwert eines Attributs kann so festgelegt werden.
- Abbildung 2.18 zeigt wie Projektinformationen in Form von Merkmalen dargestellt werden. Dazu gehören der Name des Erzeugers einer Klasse, das Erstellungsdatum, das Datum der letzten Änderung, die aktuelle Versionsnummer und ähnliches mehr.
- Informelle Kommentare können durch Merkmale dargestellt werden.
- Vorgesehene Techniken für Speicherung oder den Transfer der Daten über ein Netzwerk können so beispielsweise im Modell abgelegt werden.
- Für die graphische Darstellung in einem Werkzeug kann einem Modellelement ein geeignetes graphisches Symbol zugeordnet werden, dessen Dateiname im Merkmal abgelegt wird.

2.5.3 Einführung neuer Elemente

Die Vielfalt der Verwendungsmöglichkeiten für Stereotypen und Merkmale macht es nahezu unmöglich, eine Beschreibung der Bedeutung eines solchen Elements direkt mit der UML vorzunehmen. Bei der Definition eines Merkmals oder Stereotyps wird daher normalerweise eine informelle Beschreibung angegeben. Dadurch werden nicht nur das konkrete Aussehen, sondern vor allem auch die Intention und Einsatzgebiete beschrieben.

Der wichtigste Grund für die Einführung eines Stereotyps ist der methodische, werkzeugunterstützte Umgang während der Softwareentwicklung. Da für Stereotypen viele unterschiedliche Einsatzmöglichkeiten bestehen,

die von der Steuerung der Codegenerierung bis hin zur Dokumentation von unfertigen Baustellen im Modell reichen, kann im Allgemeinen wenig über die Bedeutung von Stereotypen gesagt werden. Deshalb wird in der Tabelle 2.19 nur ein allgemeiner Notationsvorschlag für Stereotyp-Definitionen angegeben, der für konkrete Aufgabenstellungen und durch geeignete Werkzeuge jeweils angepasst und erweitert werden kann.

Stereotyp «Name»	
Modell- element	Worauf wird der Stereotyp angewandt? Gegebenenfalls können Bilder die konkrete Darstellungsform illustrieren.
Motivation	Wozu dient der Stereotyp? Weshalb ist er notwendig? Wie unterstützt er den Entwickler?
Glossar	Begriffsbildung – soweit notwendig.
Rahmen- bedingung	Wann kann der Stereotyp angewandt werden?
Wirkung	Welche Wirkung wird damit erzielt?
Beispiel(e)	Illustration durch Anwendungsbeispiele. Meist mit Diagrammen untermauert.
Fallstricke	Welche besonderen Probleme können auftreten?
Siehe auch	Welche anderen Modellelemente sind ähnlich oder ergänzen den hier definierten Stereotyp?
Merkmale	Welche Merkmale (Name und Typ) sind mit dem Stereotyp assoziiert? Welche sind optional? Gibt es Defaultwerte? Was bedeutet ein Merkmal (soweit nicht in einer eigenen Tabelle definiert)?
Erweiterbar auf	Oft kann der Stereotyp auf ein übergeordnetes Modellelement oder ein ganzes Diagramm angewandt werden, um dann elementweise auf dessen Teilelemente zu wirken.

Tabelle 2.19.: Stereotyp «Name»

Die Definition eines Stereotyps folgt der allgemeinen Form der Entwurfsmuster [GHJV94], der Rezepte [FPR01] und der Prozessmuster [Amb98], indem sie auf informeller Basis Motivation, Voraussetzungen, Anwendungsform und Auswirkungen diskutiert. Diese Schablone sollte aber nicht als starre Form verstanden werden, sondern bei Bedarf um geeignete Abschnitte erweitert oder unnötige Abschnitte gekürzt werden.

Für Merkmale kann im Prinzip dieselbe Schablone verwendet werden. Merkmale sind jedoch im Allgemeinen einfacher strukturiert und verständlich, so dass eine derart detaillierte Schablone oft unnötig erscheint.

Die UML bietet eine dritte Form von Anpassungen für Modellelemente an. *Bedingungen* (engl.: *constraints*) sind ein Instrument zur detaillierten Spezifikation von Eigenschaften. Als Bedingungssprachen werden die im Kapitel 3 eingeführte OCL oder informeller Text vorgeschlagen. Eine Bedingung wird grundsätzlich in der Form {Bedingung} dargestellt. Der UML-Standard bietet standardmäßig einige Bedingungen an. Darunter ist die bereits bekannte Bedingung {ordered} für Assoziationen, die jedoch auch als Merkmal mit booleschen Wertebereich definiert werden kann. Insbesondere an diesem Beispiel ist zu sehen, dass die Unterschiede zwischen Bedingungen, Merkmalen und Stereotypen nicht immer zweifelsfrei geklärt werden können. So ist es auch nur wenig sinnvoll, einen Stereotyp einzuführen, der genau aus einem Merkmal besteht, weil dieses Merkmal auch direkt an ein Modellelement angefügt werden kann. Bei der Einführung neuer Stereotypen, Merkmale oder Bedingungen sind daher gewisse gestalterische Freiheiten gegeben, die von einem Modellierer genutzt werden können, um ein geeignetes Modell zu entwerfen.

Modellierung mit UML

Sprache, Konzepte und Methodik

Rumpe, B.

2011, X, 294 S. 176 Abb., 103 Abb. in Farbe., Hardcover

ISBN: 978-3-642-22412-6