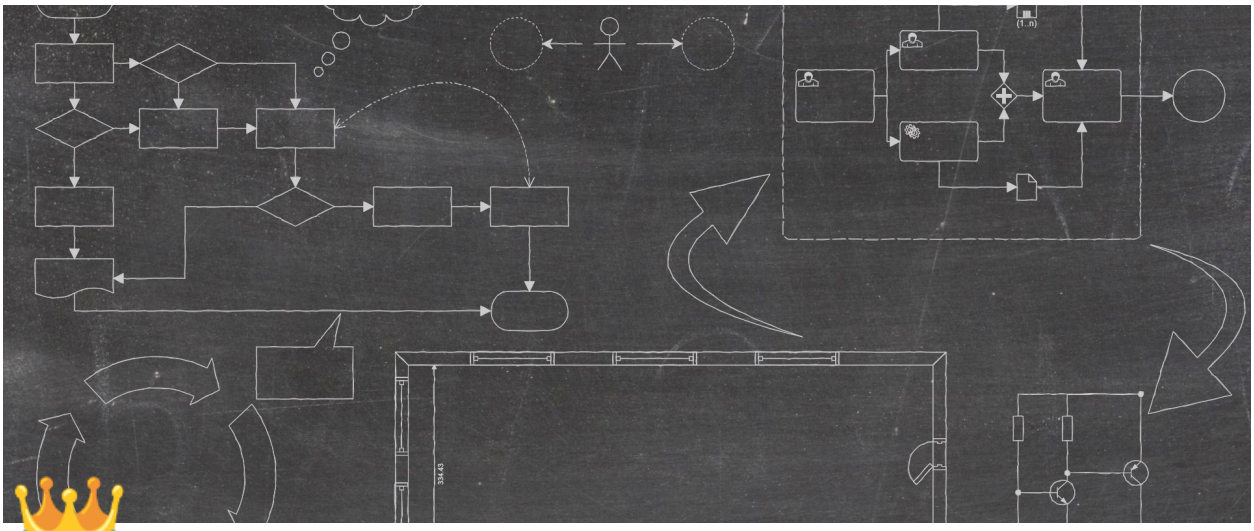


Klassendiagramme

Bedeutung der Klassendiagramme

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.

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. Außerdem wird dieser Standpunkt oft auch durch Werkzeuge impliziert. Fusion 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.

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.

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

Metamodellierung

Für die Beschreibung einer diagrammatischen Sprache hat sich aufgrund ihrer zweidimensionalen Darstellungsform die *Metamodellierung* 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 und nutzen eine Variante der Klassendiagramme um die graphischen Anteile der UML/P darzustellen.

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.

Klasse

Eine Klasse besteht aus einer Sammlung von Attributen und Methoden, die den Zustand und das Verhalten ihrer *Instanzen* (*Objekte*) festlegt. Klassen sind durch Assoziationen und Vererbungsbeziehungen miteinander verknüpft. Ein *Klassenname* erlaubt es, die Klasse zu identifizieren.

Attribut

Die Zustandskomponenten einer Klasse werden als Attribute bezeichnet. Sie beinhalten grundsätzlich *Name* und *Typ*.

Methode

Die Funktionalität einer Klasse ist in Methoden abgelegt. Eine Methode besteht aus einer *Signatur* und einem *Rumpf*, der die Implementierung beschreibt. Bei einer *abstrakten* Methode fehlt der Rumpf.

Modifikator

Zur Festlegung von Sichtbarkeit, Instanziierbarkeit und Veränderbarkeit des modifizierten Elements können die
Modifikatoren public, protected, private, readonly, abstract, static und final auf Klassen, Methoden, Rollen und Attribute angewandt werden. Für die ersten vier genannten Modifikatoren gibt es in UML/P die graphischen Varianten „+“, „#“ und „-“ und „?“.

Konstanten

sind als spezielle Attribute mit den Modifikatoren static und final definiert.

Vererbung

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

Interface

Ein Interface (Schnittstelle) beschreibt die Signaturen einer Sammlung von Methoden. Im Gegensatz zur Klasse werden keine Attribute (nur Konstanten)

und keine Methodenrümpfe angegeben. Interfaces sind verwandt zu abstrakten Klassen und können untereinander ebenfalls in einer Vererbungsbeziehung stehen.

Typ

ist ein Basisdatentyp wie int, eine Klasse oder ein Interface.

Interface-Implementierung

ist eine der Vererbung ähnliche Beziehung zwischen einem Interface und einer Klasse. Eine Klasse kann beliebig viele Interfaces implementieren.

Assoziation

ist eine binäre Beziehung zwischen Klassen, die zur Realisierung struktureller Information verwendet wird. Eine Assoziation durch
einen *Assoziationsnamen*, für jedes Ende einen *Rollennamen*,
eine *Kardinalität* und eine Angabe über
die *Navigationsrichtungen* beschrieben.

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.

Abbildung 2.2

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.

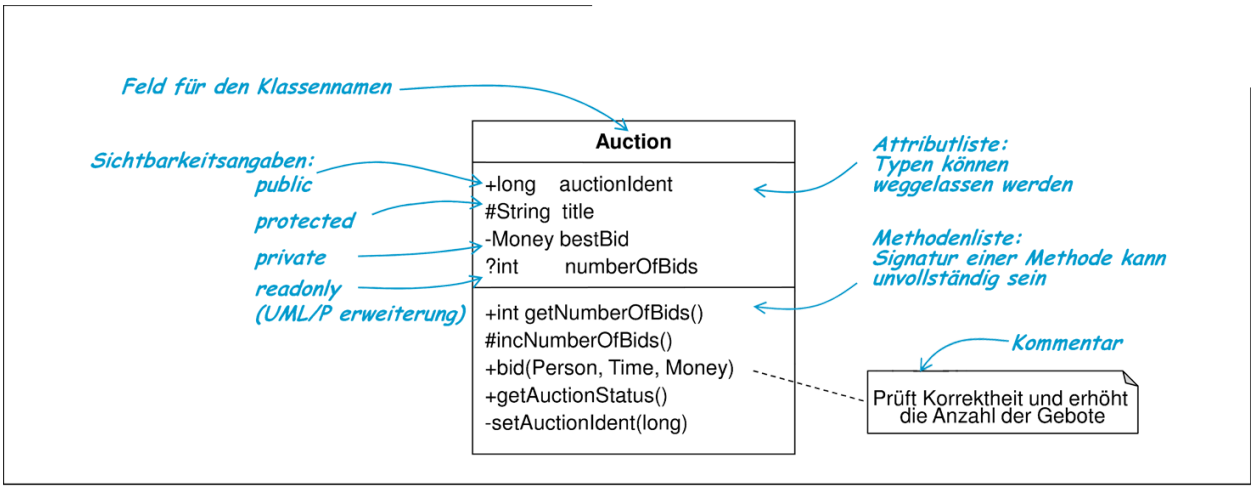


Abbildung 2.3: Klasse Auction im Klassendiagramm

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 „attribut: Typ“ durch die Java-konforme Fassung „Typ attribut“ ersetzt. In unserem Falle wird aber standardmäßig übliche Form verwendet.

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

- “#” für protected
- “-” 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.

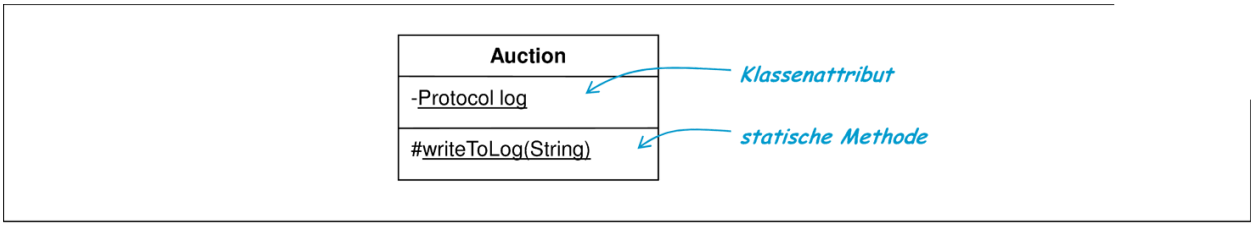


Abbildung 2.4: Klassenattribut und statische Methode

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) 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.

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.

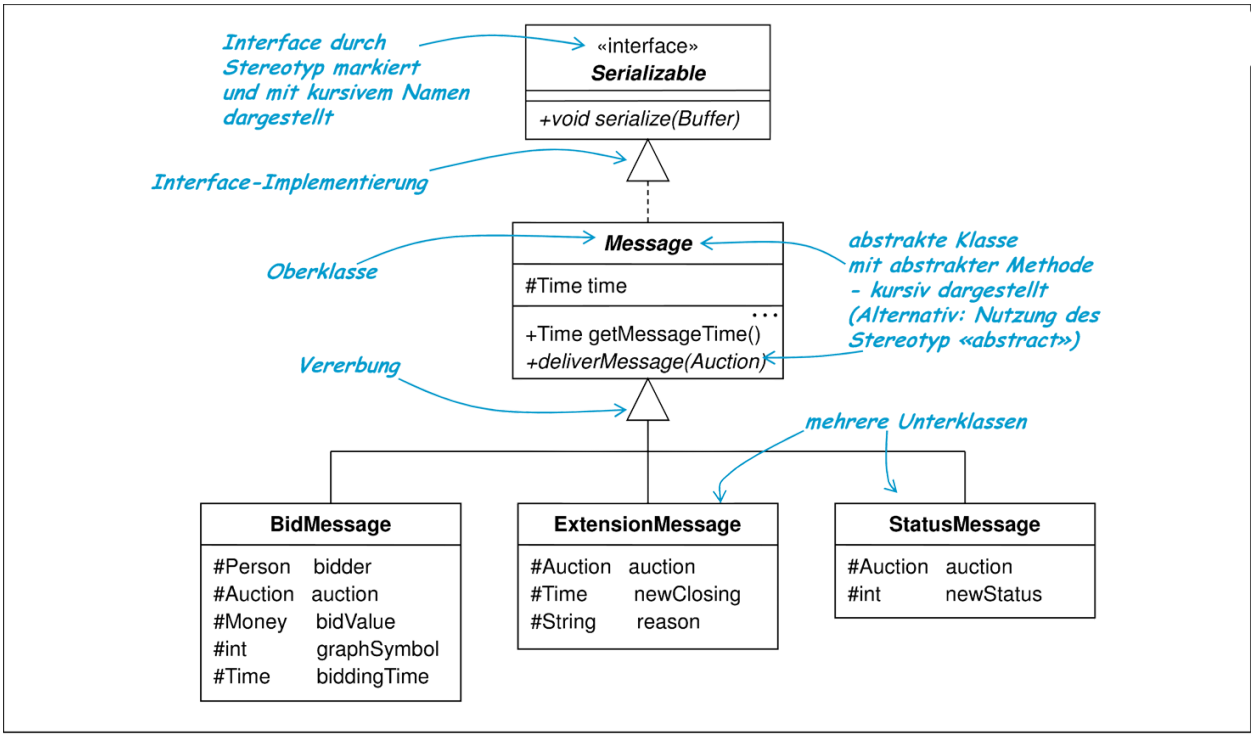


Abbildung 2.6: Vererbung und Interface-Implementierung

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.

Interfaces

Viele Programmiersprachen 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.

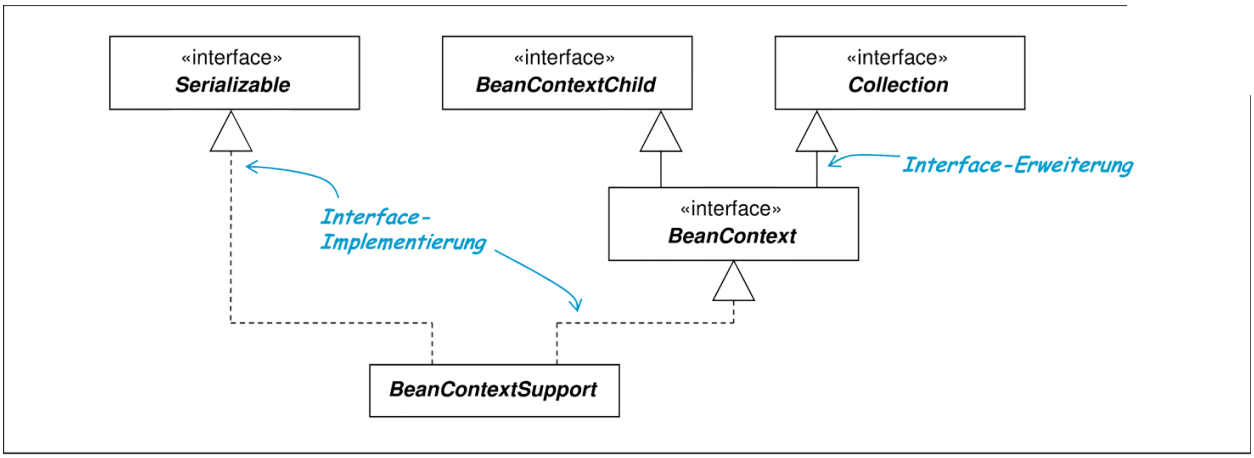
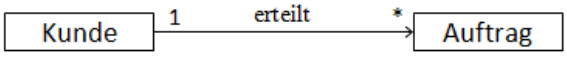


Abbildung 2.7: Interface-Implementierung und -Erweiterung

Technisch gesehen sind Interfaces und Klassen sowie Vererbung und Interface-Implementierung jeweils ähnliche Konzepte. Vereinfachend wird deshalb in Zukunft häufig der Begriff *Klasse* als Oberbegriff für Klassen und Interfaces sowie *Vererbung* für die Vererbung zwischen Klassen, die Implementierungsbeziehung zwischen Interfaces und Klassen und für die Subtyp-Beziehung zwischen Interfaces verwendet. Diese Vereinfachung ist insbesondere in der Analyse und dem Grobdesign sinnvoll, wenn eine Entscheidung, ob eine Klasse instanziierbar, abstrakt oder ein Interface wird, noch nicht getroffen ist.

Assoziationen

Definition: Assoziation (kennt-Beziehung; benutzt-Beziehung)



Bestehen zwischen Objekten von Klassen Beziehungen, dann spricht man von **Assoziationen**. Dabei kennen sich die Objekte, existieren aber unabhängig voneinander. Ein Objekt, das ein anderes Objekt kennt, verwaltet dieses nicht.

In der Regel zeichnet man Assoziationen in ein Klassendiagramm ein. Zwischen zwei Klassen wird eine Linie eingezeichnet. Am Ende der Linie kennzeichnet eine (offene) Pfeilspitze die Art der Beziehung (Assoziation).

Eine Assoziation dient dazu, Objekte zweier Klassen in Beziehung zu setzen. Mithilfe von Assoziationen können komplexe Datenstrukturen gebildet werden und Methoden benachbarter Objekte aufgerufen werden. Abbildung 2.8 beschreibt einen Ausschnitt aus dem Auktionssystem mit drei Klassen, zwei Interfaces und fünf Assoziationen in unterschiedlichen Formen.

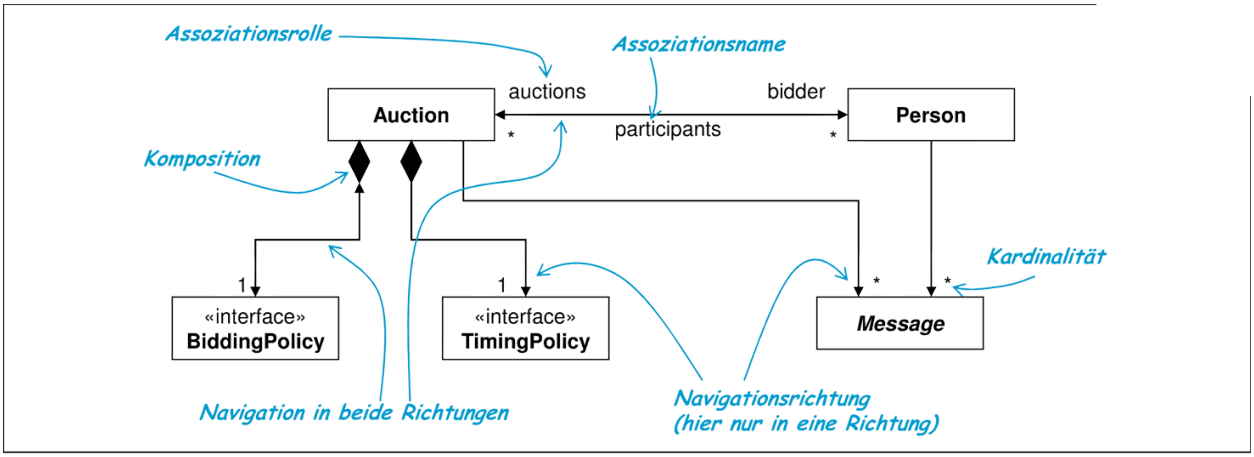


Abbildung 2.8: Klassendiagramm mit Assoziationen

Eine Assoziation besitzt im Normalfall einen *Assoziationsnamen* und für jedes der beiden Enden je eine *Assoziationsrolle*, eine Angabe der *Kardinalität* und eine Beschreibung der möglichen *Navigationsrichtungen*. Einzelne Angaben können im Modell auch weggelassen werden, wenn sie zur Darstellung des gewünschten Sachverhalts keine Rolle spielen und die Eindeutigkeit nicht verloren geht.

Zum Beispiel dienen Assoziationsnamen häufig nur zur Unterscheidung von Assoziationen, insbesondere dann, wenn sie die gleichen Klassen verbinden.

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.

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 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.

Aggregation

Definition: Aggregation (hat-Beziehung; besitzt-Beziehung)

Die **Aggregation** ist eine Sonderform der Assoziation zwischen zwei Klassen. Sie liegt dann vor, wenn zwischen

Aggregationen sind festere Beziehungen zwischen Objekten als die oben beschriebenen Assoziationen, z. B. wenn

liegt dann vor, wenn zwischen den Objekten der beteiligten Klassen eine Beziehung vorliegt, die sich als „ist Teil von“, „besteht aus“ oder einfach „hat“ beschreiben lässt.

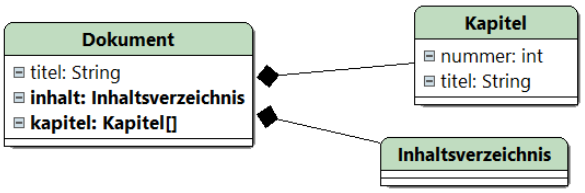
eine Rangordnung zwischen den Objekten besteht. Eine Assoziation wird in einem UML-Diagramm durch eine offene Raute an der **besitzenden** Seite gekennzeichnet.

Eine Aggregation beschreibt, wie sich ein Ganzes, d.h. ein Objektverbund, aus den verschiedenen Teilen, d.h. den Einzelobjekten, zusammensetzt; es lassen sich hierarchischen Teil, d.h. den Einzelobjekten, zusammensetzen; es lassen sich hierarchische Strukturen einer Objektmenge darstellen.

Komposition

Definition: Komposition

Die **Komposition** ist eine Sonderform der Aggregation. Sie drückt aus, dass die Teile von der Existenz des Ganzen abhängig sind.



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 Bidding Policy und Timing Policy 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 Bidding Policy und Timing Policy 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 Enthalten sein nutzt. Abbildung 2.9 zeigt zwei Alternativen, die sich nur in

Details unterscheiden. Im Klassendiagramm ist der Assoziationscharakter der Komposition herausgestellt.

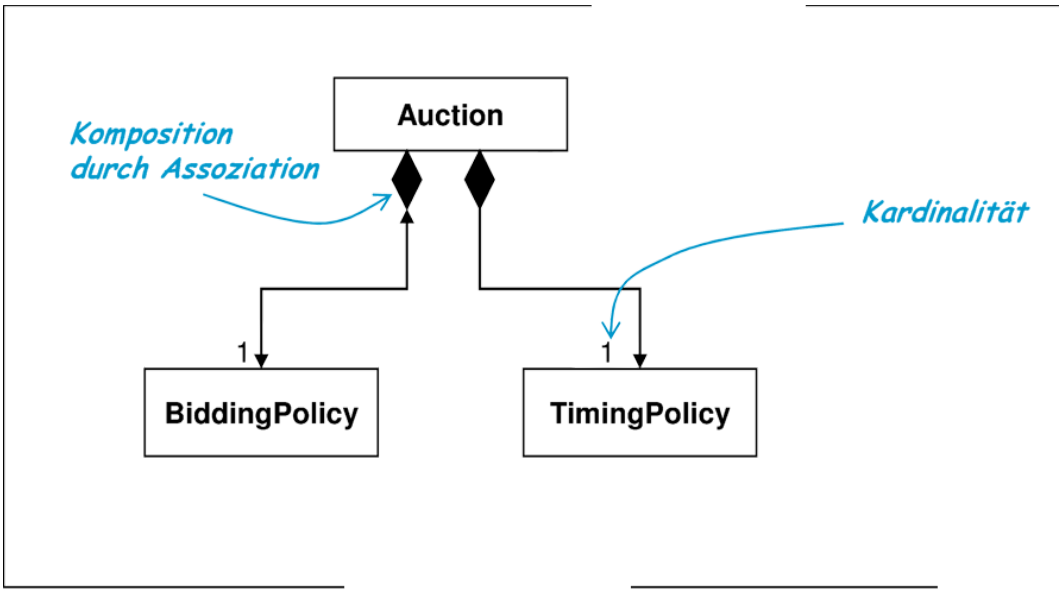


Abbildung 2.9: Alternative Darstellungen von Komposition

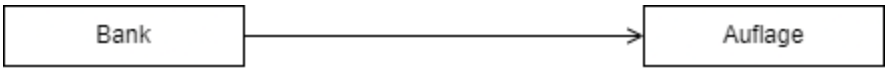
Bezüglich der Austauschbarkeit und des Lebenszyklus von abhängigen Objekten in einem Kompositum gibt es erhebliche Interpretationsunterschiede. Eine präzise Definition der Bedeutung einer Komposition sollte daher jeweils projektspezifisch festgelegt werden. Dies kann zum Beispiel durch Stereotypen erfolgen,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

Beispiele

Assoziation – Aggregation – Komposition

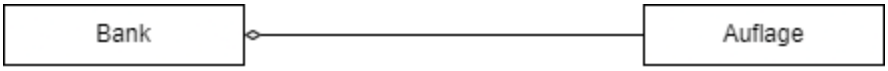
Treiben wir es in einem Beispiel mal auf die Spitze … wir haben die zwei Objekte Bank und Auflage für eine Bank. Unter welchen Rahmenbedingungen liegt nun eine Assoziation, eine Aggregation bzw. eine Komposition vor.

Assoziation



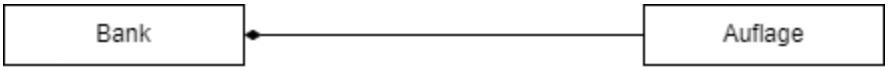
Es existiert zu der Bank eine passende Auflage, die zwar zur Bank gehört aber trotzdem auch selbständig gekauft werden kann (und auch auf andere Bänke gelegt werden kann).

Aggregation



Die Bank besitzt eine spezielle pass-genaue Auflage, die nur hier passt und auch direkt mitgeliefert wird. Diese könnte u. A. auch fest angeschraubt sein (und wieder abgeschraubt werden können. Hier könnten auch mehrere Auflagen existieren, die man je nach Wunsch wechselt.

Komposition



Da hier eine existentielle Abhängigkeit bestehen muss, müsste die Auflage also unwiderruflich fest auf der Bank angebracht sein, so dass eine Abnahme nicht möglich ist. Eine Zerstörung der Bank würde auch die Auflage zerstören.

Quellen

- https://mbse.se-rwth.de/book1/index.php?c=chapter2-2