Software-Entwurf

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Objektorientierter Entwurf

In diesem Abschnitt werden wir untersuchen, wie die Komponenten eines Systems in Hinblick auf Gemeinsamkeiten und spätere Änderungen ausgesucht werden.

Hierzu dient uns der *objektorientierte Entwurf*.

Objektorientierte Modellierung

Die *Objektorientierte Modellierung mit UML* umfaßt u.a. folgende Aspekte des Entwurfs:

Objekt-Modell: Welche Objekte benötigen wir?

- Welche Merkmale besitzen diese Objekte? (Attribute, Methoden)
- Wie lassen sich diese Objekte klassifizieren? (Klassenhierarchie)
- Welche Assoziationen bestehen zwischen den Klassen?

Sequenzdiagramm: Wie wirken die Objekte global zusammen?

Zustandsdiagramm: In welchen Zuständen befinden sich die Objekte?

Objekt-Modell: Klassendiagramm

Darstellung der Klassen als Rechtecke, unterteilt in

- Klassenname,
- Attribute – möglichst mit Typ (meistens ein Klassenname)
- Methoden – möglichst mit Signatur

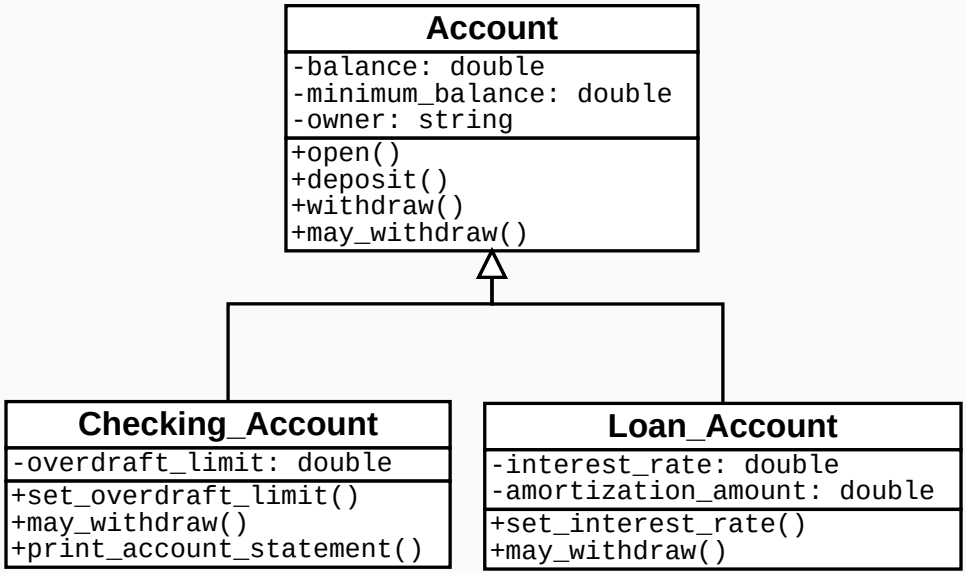
Darstellung der Vererbungshierarchie – ein Dreieck (Symbol: $\triangle$) verbindet Oberklasse und Unterklassen.







Beispiel für Vererbung: Konten



Ererbte Methoden (wie hier: `open()`, `deposit()`) werden in der Unterklasse nicht mehr gesondert aufgeführt.

Definitionen in der Unterklasse *überschreiben* die Definition in der Oberklasse (hier: `may_withdraw()`)

Abstrakte Klassen und Methoden

Klassen, von denen keine konkreten Objekte erzeugt werden können, heißen *abstrakte Klassen*. Sie verfügen meist über eine oder mehrere *abstrakte Methoden*, die erst in Unterklassen realisiert werden.

Eine *konkrete Klasse* ist eine Klasse, von der konkreten Objekte erzeugt werden können.

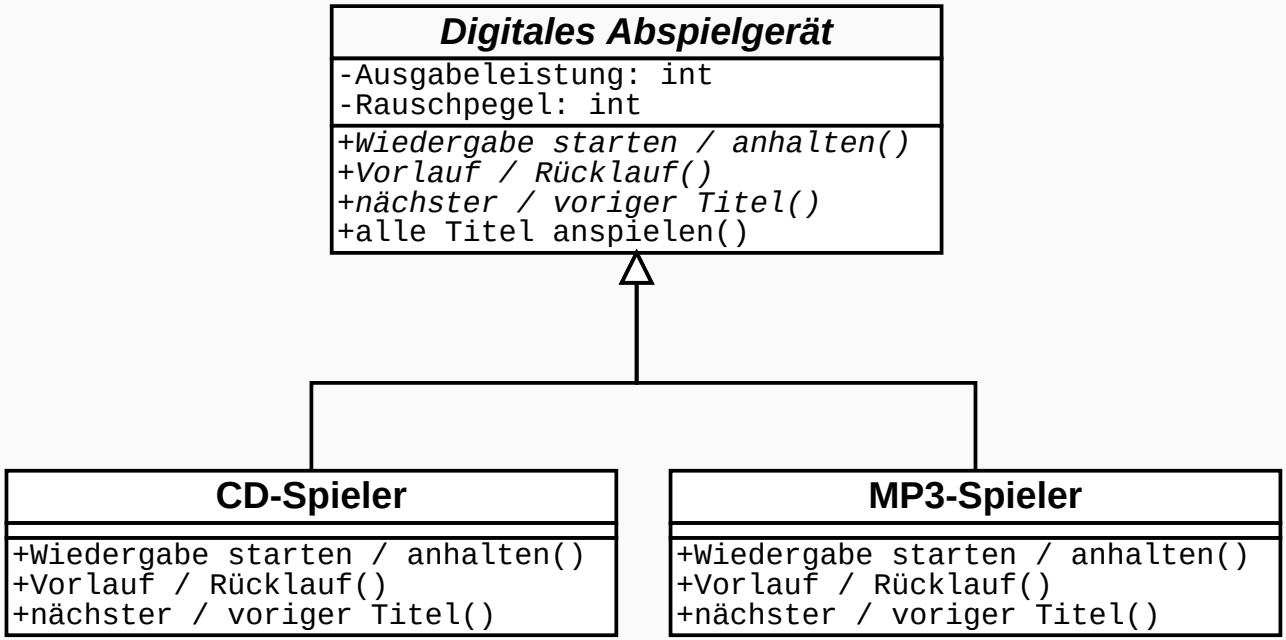






Beispiel für abstrakte Klassen

Ein „Digitales Abspielgerät“ ist ein abstrakter Oberbegriff für *konkrete* Realisierungen – z.B. ein CD- oder ein MP3-Spieler.



Kursiver Klassen-/Methodenname: abstrakte Klasse/Methode

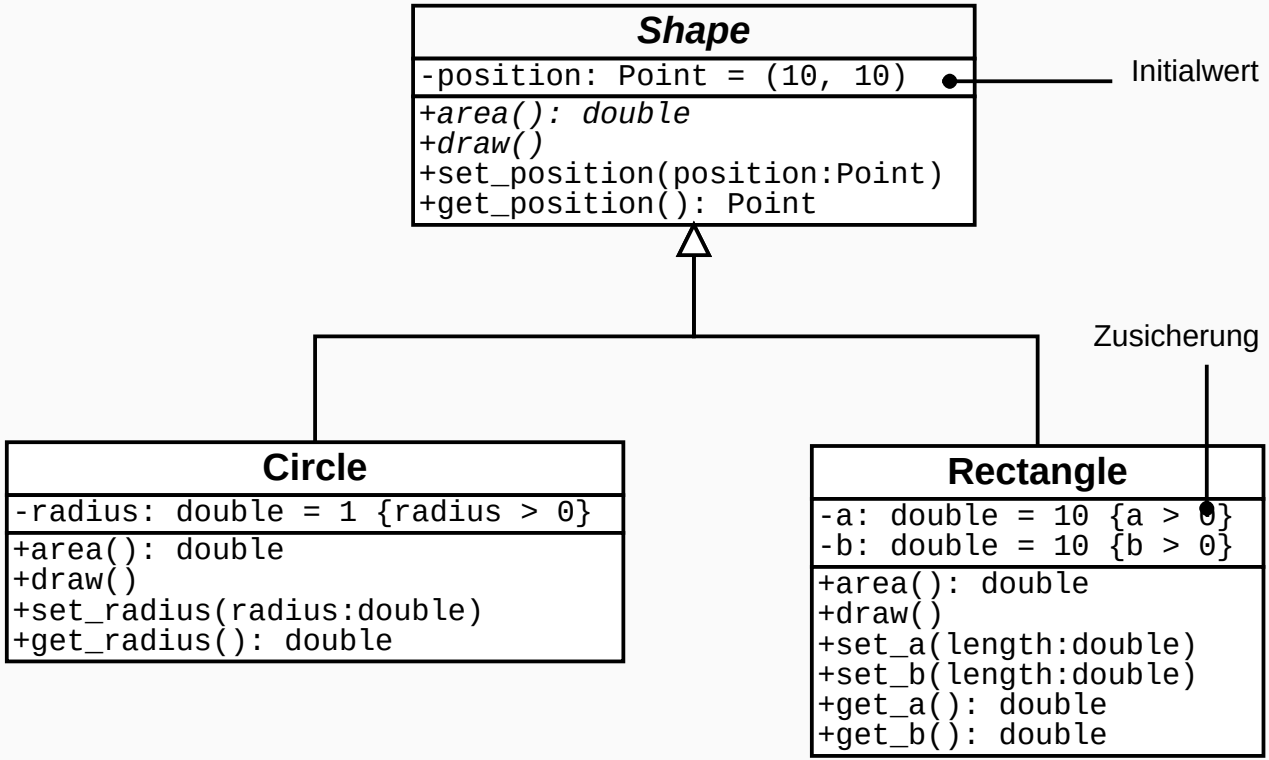
Initialwerte und Zusicherungen

Die Attribute eines Objekts können mit *Initialwerten* versehen werden. Diese gelten als *Vorgabe*, wenn bei der Konstruktion des Objekts nichts anderes angegeben wird.

Mit *Zusicherungen* werden *Bedingungen* an die Attribute spezifiziert. Hiermit lassen sich *Invarianten* ausdrücken – Objekt-Eigenschaften, die stets erfüllt sein müssen.

Beispiel für Zusicherungen

Die Zusicherungen garantieren, daß Kreise immer einen positiven Radius haben und Rechtecke positive Kantenlängen.



Das Problem der Spezialisierung

Eine Unterklasse ist eine *Spezialisierung* einer Oberklasse, wenn sie

- weitere Zusicherungen über die Attribute der Oberklasse enthält
- weitere Vorbedingungen über die Methoden der Oberklasse enthält

Klassenhierarchien sollten Spezialisierung vermeiden.

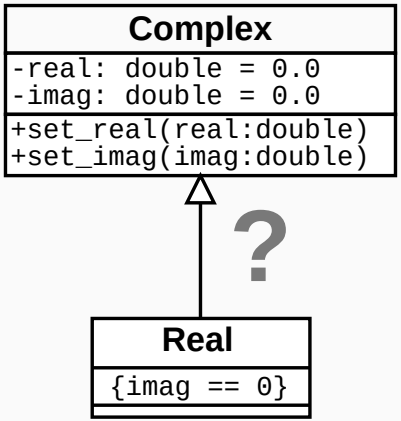






Reelle und komplexe Zahlen

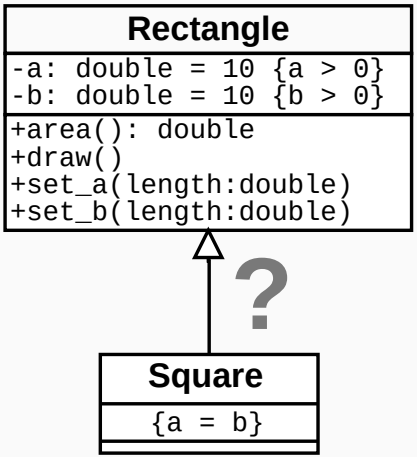
Jede reelle Zahl ist auch eine komplexe Zahl (mit Imaginärteil 0). Sollten deshalb reelle Zahlen Unterklassen der komplexen Zahlen werden?



Problem: `set_imag`, von `Complex` geerbt, würde die Zusicherung von `Real` verletzen.

Analog: Quadrat

Analog: Ein Quadrat „ist ein“ Rechteck. Ist `Square` eine Unterklasse von `Rectangle`, verletzt `set_a` die Zusicherung von `Square`.



Ziel: Konformanz

Konformanz bedeutet, daß ein Objekt einer Unterklasse *in jeder Beziehung* als Objekt der Oberklasse einsetzbar ist.

Klassenhierarchien sollten so konstruiert werden, daß Konformanz stets sichergestellt ist.

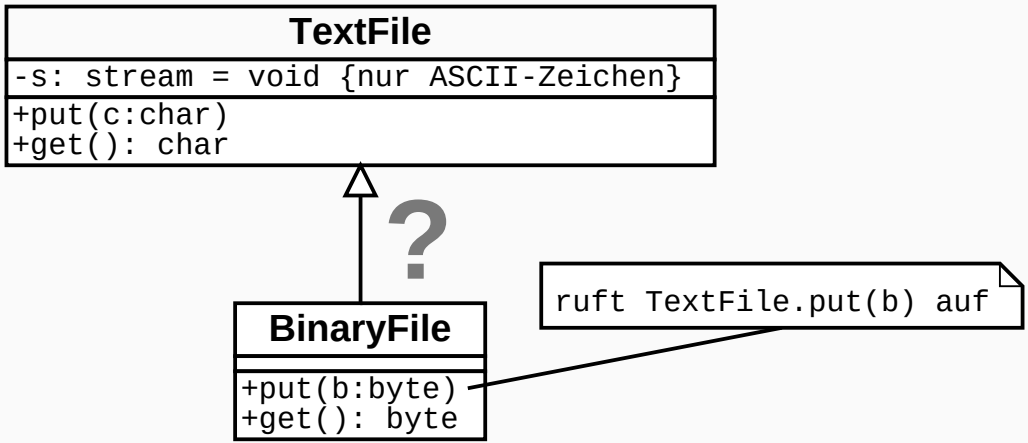






Verletzte Konformanz

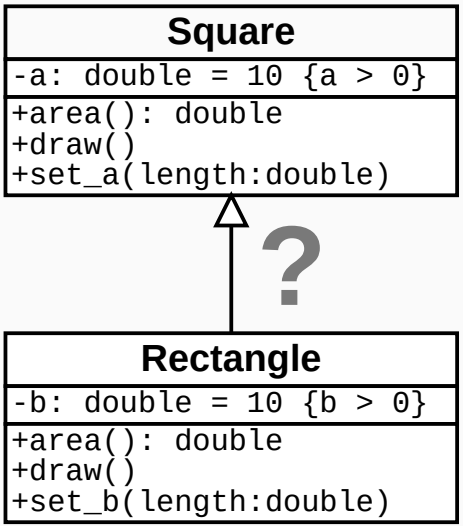
In folgendem Beispiel ist die Konformanz verletzt. Von einer existierenden Klasse `TextFile` hat ein Programmierer eine Klasse für binäre Dateien abgeleitet, um deren Methoden zu erben (sogenanntes *Erben von Implementierungen*).



Die Methoden des `BinaryFile` verletzen jedoch die Zusicherung des `TextFile`.

Verletzte Konformanz (2)

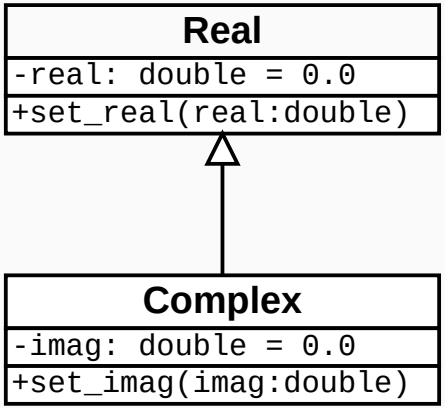
Auch ohne explizite Zusicherungen kann die Konformanz verletzt sein:



Auch hier ist die Konformanz verletzt: Wird irgendwo ein Quadrat bearbeitet, jedoch ein Rechteck übergeben, wird nur die Kantenlänge a berücksichtigt.

Verletzte Konformanz (3)

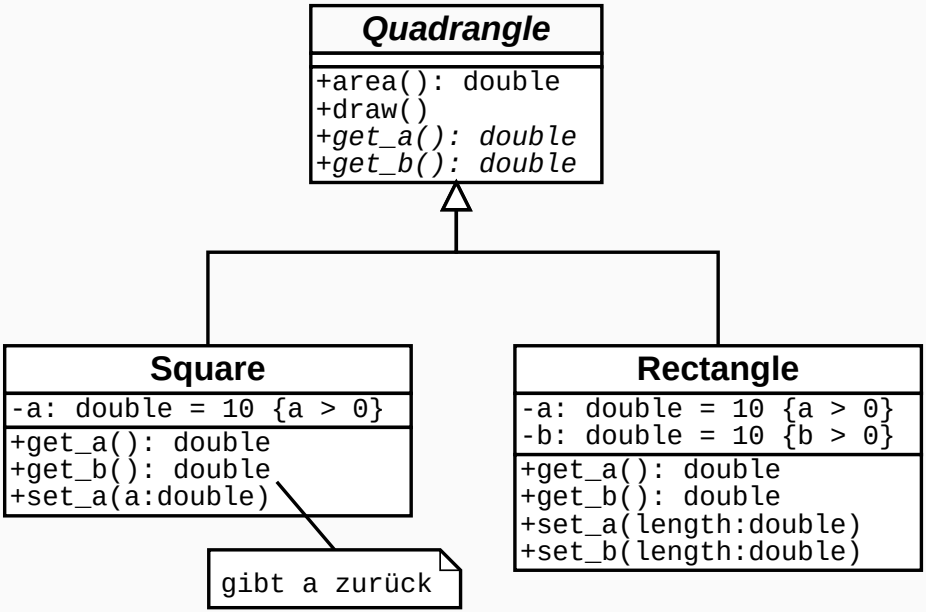
Im Fall reelle/komplexe Zahlen könnte eine solche Anordnung jedoch Sinn machen: Jede komplexe Zahl kann als reelle verwendet werden (wobei die Funktionalität auf den Realteil beschränkt ist).



Frage ist, ob eine solche implizite Umwandlung von komplexen in reelle Zahlen nicht mehr Risiken als Nutzen bringt.

Implizite Umwandlung

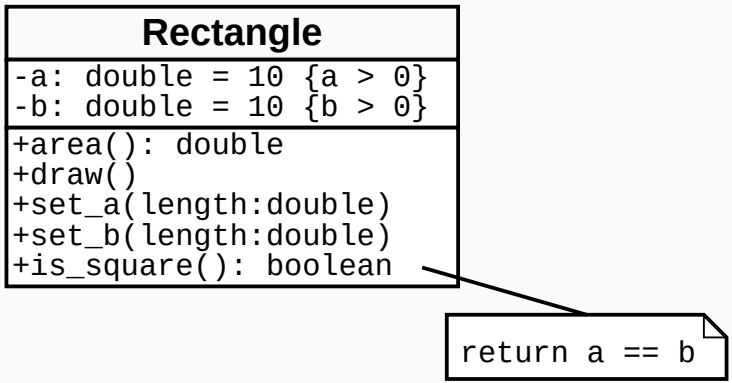
Um tatsächlich Quadrate und Rechtecke zu modellieren, bieten sich zwei Wege an. Man kann Quadrate und Rechtecke als separate Klassen modellieren, die durch eine neue gemeinsame Oberklasse verbunden sind:



Die bequeme Lösung

Alternativ kann man sich auf den Standpunkt stellen, daß jedes Rechteck ein Quadrat sein kann – nämlich dann, wenn die beiden Seiten gleich lang sind.

Idee: verzichte auf eigene Klasse für Quadrate; erweitere Rechtecke um `is_square`:



Nach dem gleichen Muster können auch Text-/Binär-Dateien und reelle/komplexe Zahlen modelliert werden.

Schnittstellen

Schnittstellen sind mit dem Schlüsselwort `interface` gekennzeichnet.

Die *implementierungs*-Beziehung ist gestrichelt dargestellt; sie wird mit dem Schlüsselwort `implements` versehen.

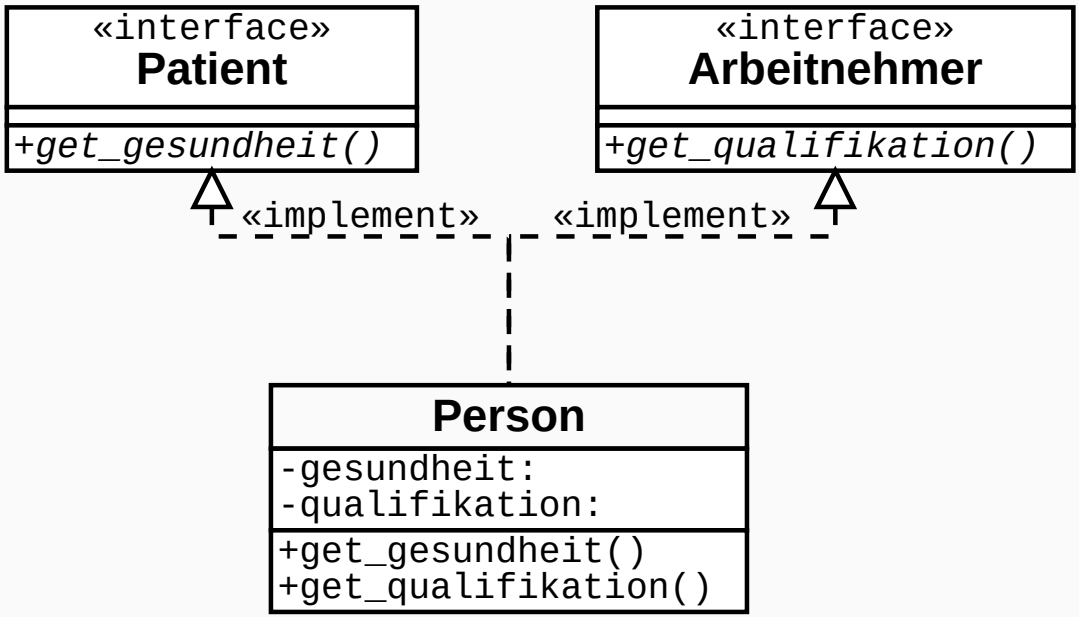






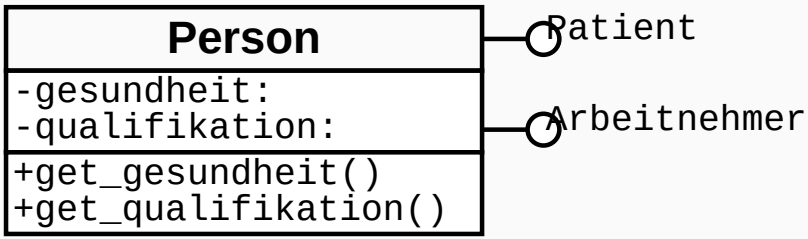
Schnittstellen (2)

Beispiel: Eine Person realisiert sowohl eine Patienten- als auch eine Arbeitnehmerschnittstelle



Schnittstellen (3)

Kurzschreibweise mit „Lolli“



Objekt-Modell: Assoziationen

Allgemeine Assoziationen

- Verbindungen zwischen nicht-verwandten Klassen stellen Assoziationen (Relationen) zwischen Klassen dar
- Diese beschreiben den *inhaltlichen Zusammenhang* zwischen Objekten (vgl. Datenbanktheorie!)
- Durch *Multiplizitäten* wird die Anzahl der assoziierten Objekte eingeschränkt.

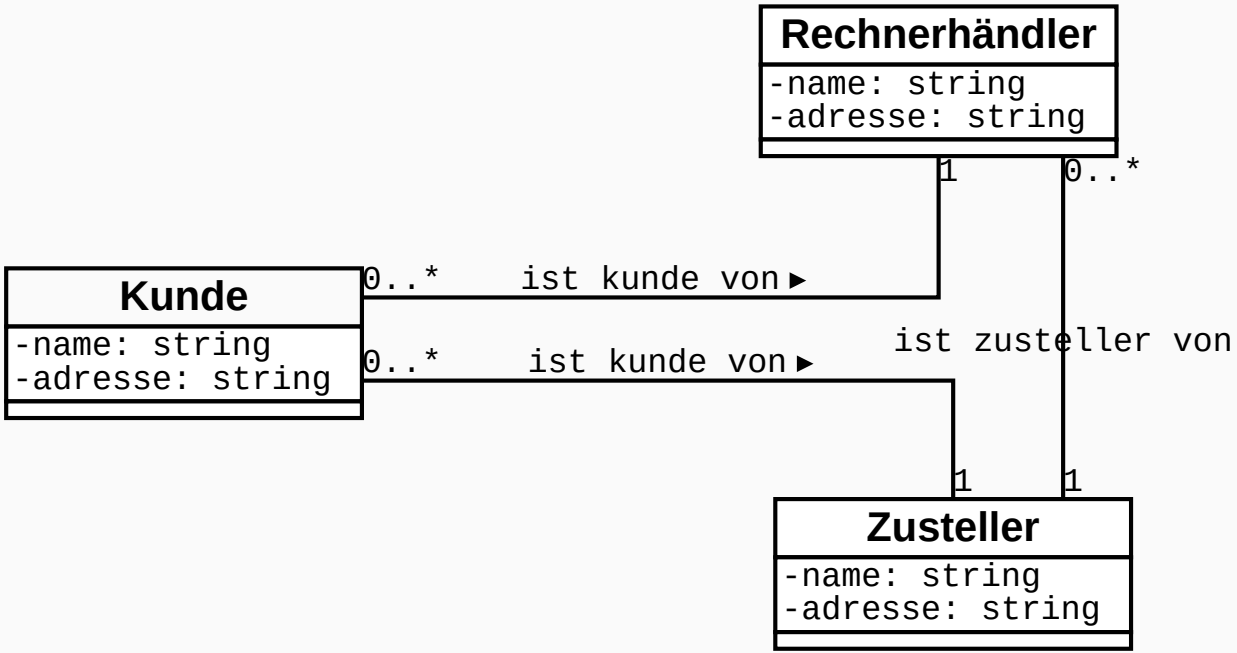






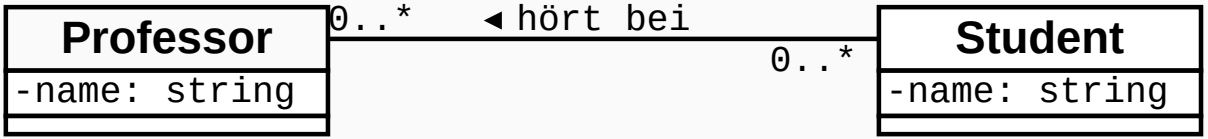
Multiplizitätsangaben

Ein *Rechnerhändler* hat mehrere *Kunden*, ein *Zusteller* hat mehrere Kunden, aber ein Rechnerhändler hat nur einen Zusteller

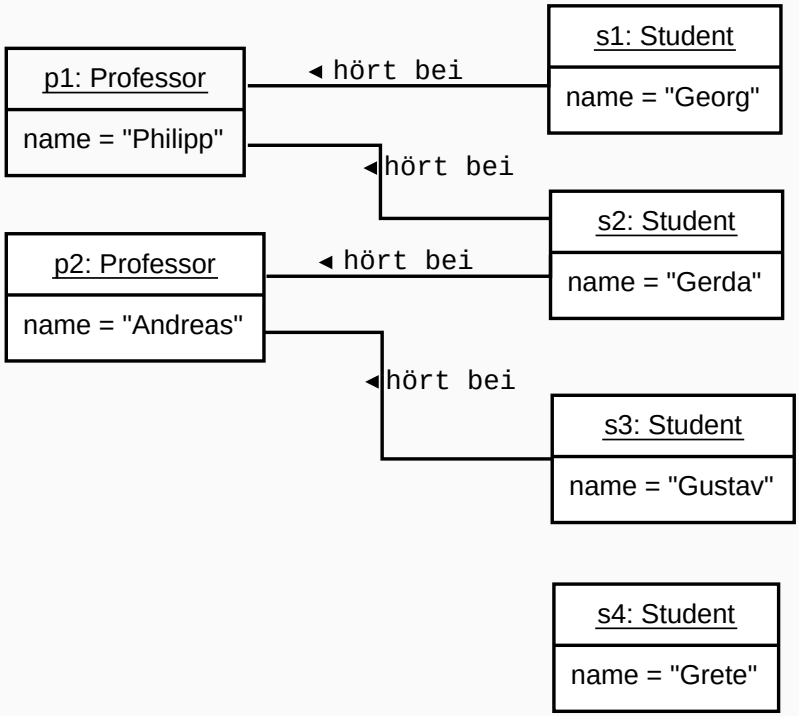


Multiplizitätsangaben (2)

Beispiel: *Professoren* haben mehrere Studenten, *Studenten* haben mehrere Professoren



Beispiel für Objektbeziehungen

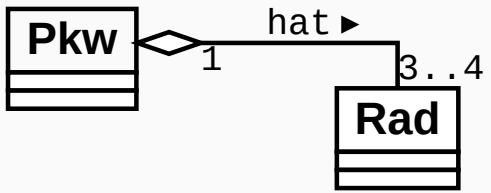


Darstellung von Exemplaren (Objekten) mit unterstrichenem Objektnamen; konkrete Werte für die Attribute

Aggregation

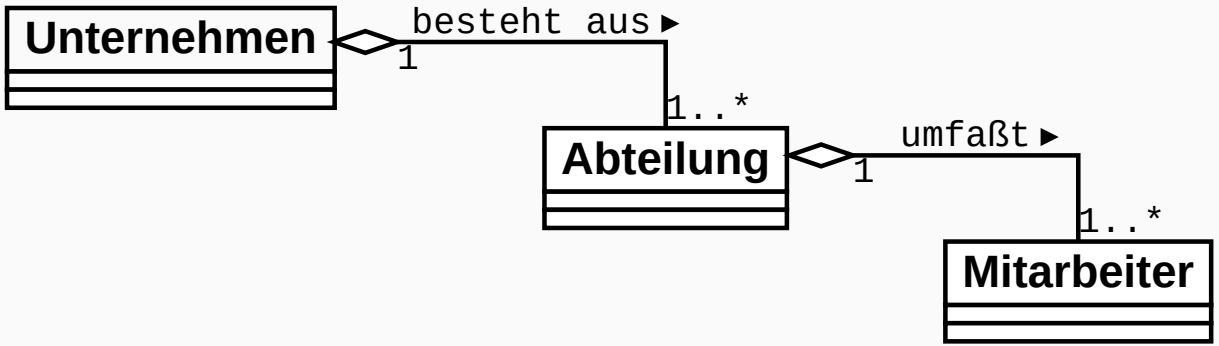
Eine häufig auftretende und deshalb mit ◊ besonders markierte Assoziation ist die *hat*-Beziehung, die die Hierarchie zwischen einem Ganzen und seiner Teile ausdrückt.

Beispiel: Ein Pkw hat 3–4 Räder



Aggregation (2)

Beispiel: Ein Unternehmen hat 1..* Abteilungen mit jeweils 1..* Mitarbeitern



Aggregation (3)

Ein Aggregat kann (meistens zu Anfang) auch ohne Teile sein: die Multiplizität 0 ist zulässig. Gewöhnlich ist es jedoch Sinn eines Aggregats, Teile zu sammeln.

Bei einem Aggregat *handelt das Ganze stellvertretend für seine Teile*, d.h. es übernimmt Operationen, die dann an die Einzelteile weiterpropagiert werden.

Beispiel: Methode `berechneUmsatz()` in der Klasse `Unternehmen`, die den Umsatz der Abteilungen aufsummiert.

Komposition

Ein Sonderfall der Aggregation ist die *Komposition*, markiert mit ⧫.

Eine Komposition liegt dann vor, wenn das Einzelteil vom Aggregat *existenzabhängig* ist – also nicht ohne das Aggregat existieren kann.





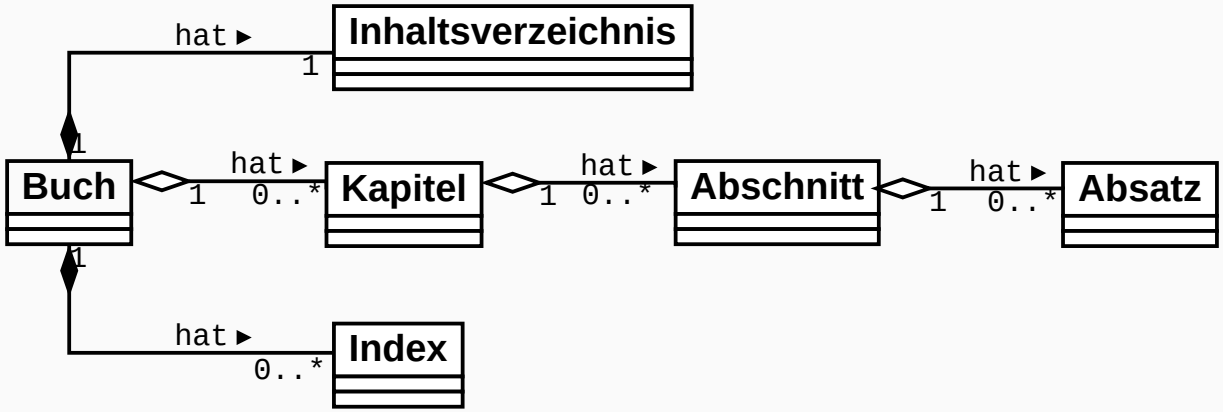


Beispiel: Rechnungsposition

Eine Rechnungsposition gehört immer zu einer Rechnung.

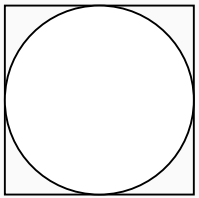
Beispiel: Buch

Ein *Buch* besteht aus *Inhaltsverzeichnis*, mehreren *Kapiteln*, *Index*; ein Kapitel besteht aus mehreren *Abschnitten* usw.



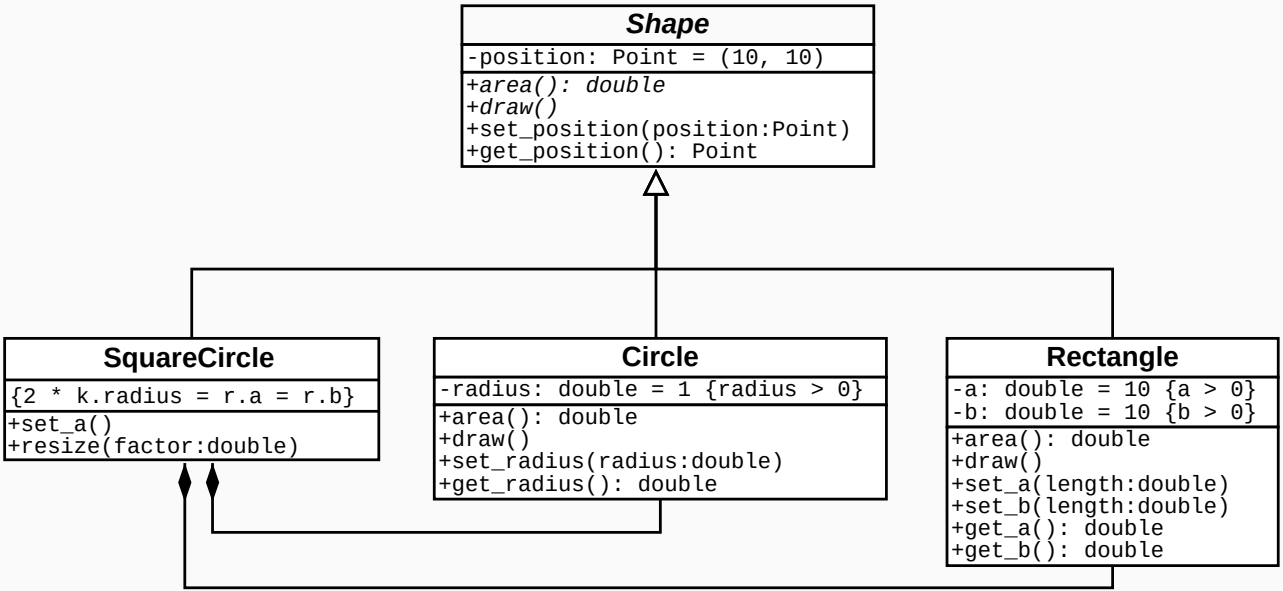
Beispiel: Kreiseck

Ein „Kreiseck“ besteht aus einem übereinander gelegten Quadrat und einem Kreis:



Beispiel: Kreiseck (2)

Modellierung als Klasse `SquareCircle`, die sowohl einen Kreis als auch ein Quadrat enthält:



Ergänzungen

Ein Einzelteil kann immer nur von *einem* Aggregat existenzabhängig sein.

Eine Klasse kann auch als Komposition aller ihrer Attribute aufgefaßt werden.

In vielen Programmiersprachen werden *Aggregationen* durch Referenzen (Zeiger auf Objekte) realisiert, *Kompositionen* jedoch durch die Werte selbst.







Rollen

An jedem Ende einer Assoziation können *Rollennamen* vergeben werden. Ein Rollenname beschreibt, wie das Objekt durch das in der Assoziation gegenüberliegende Objekt gesehen wird.

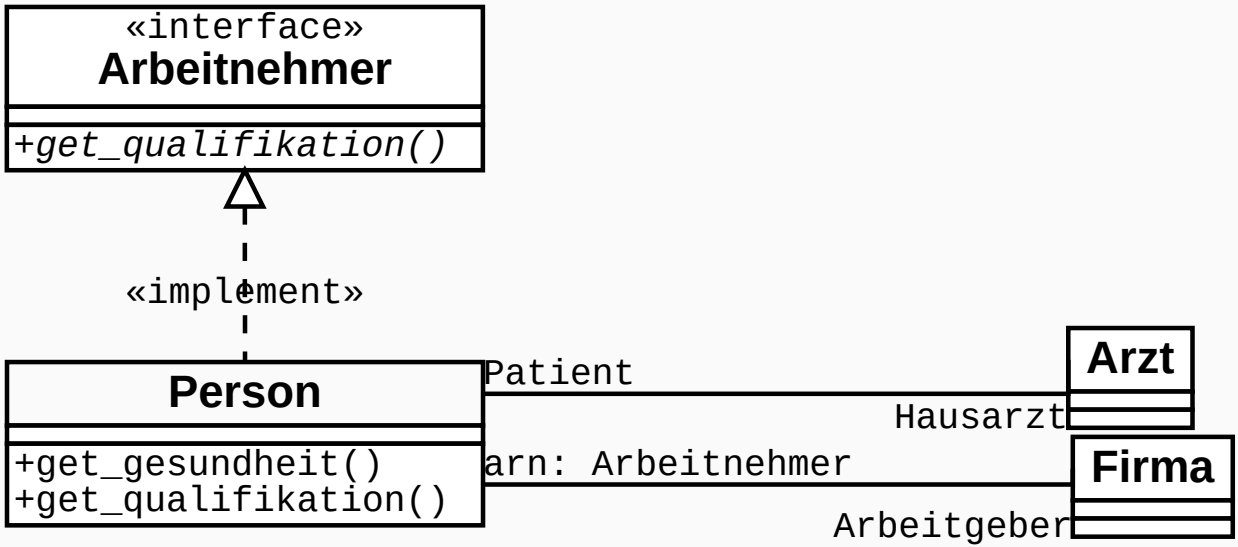






Rollen (2)

Beispiel: Eine Person wird vom Arbeitgeber als *Arbeitnehmer*, vom Hausarzt als *Patient* gesehen.



Rollen (3)

Zusätzlich zum Rollennamen kann der Name einer Schnittstelle angegeben werden, die von der jeweiligen Klasse implementiert wird (hier: die Schnittstelle `Arbeitnehmer` zum Rollennamen `arn`).

In diesem Fall wird der *Umfang der Assoziation* eingeschränkt – es sind nur noch die Attribute/Methoden der Schnittstelle sichtbar.

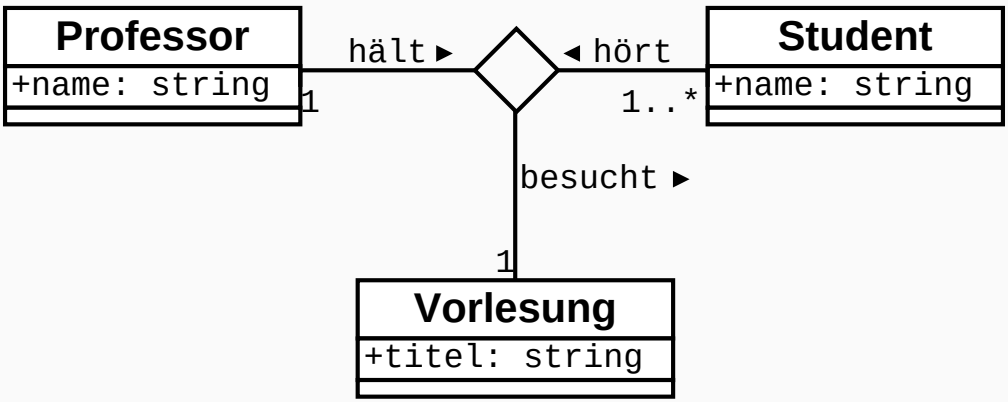
In obigem Beispiel kann etwa die Firma nicht auf die Patientendaten zugreifen.

Mehrgliedrige Assoziationen

Neben Zweierbeziehungen gibt es noch *mehrgliedrige Assoziationen* – Assoziationen, an denen drei oder mehr Klassen beteiligt sind.

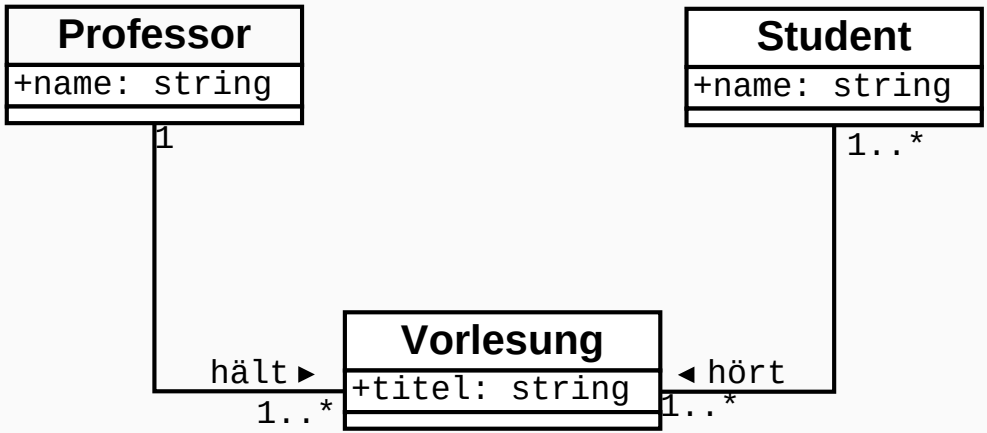
Ternäre Assoziationen

Beispiel für eine ternäre (dreigliedrige) Assoziation: *Studenten* hören *Vorlesungen* bei *Professoren*



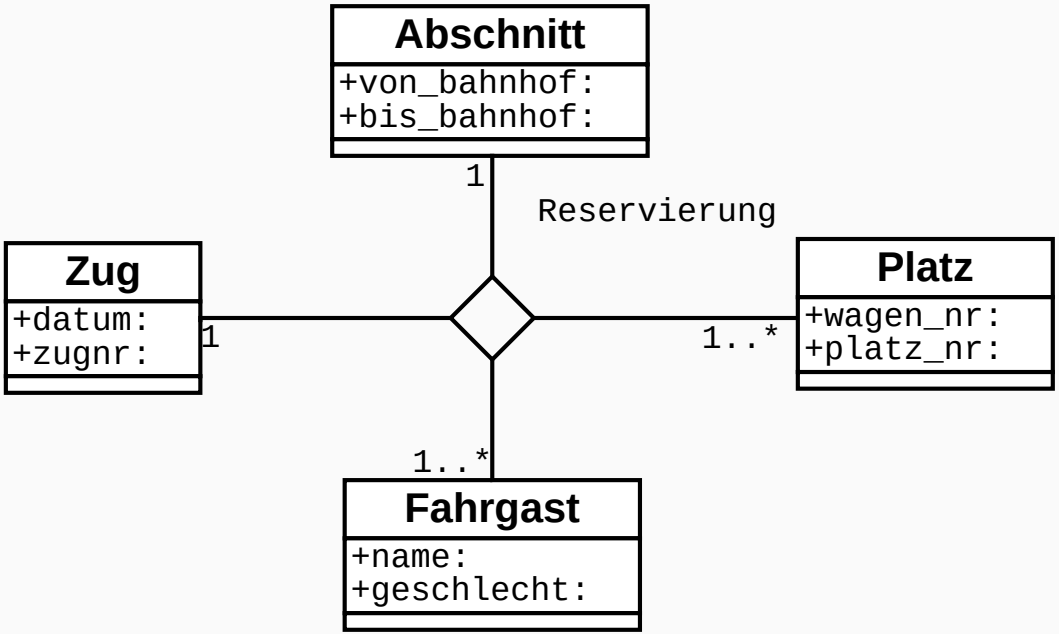
Ternäre Assoziationen (2)

Mehrgliedrige Assoziationen lassen sich gewöhnlich in normale Assoziationen umformen:



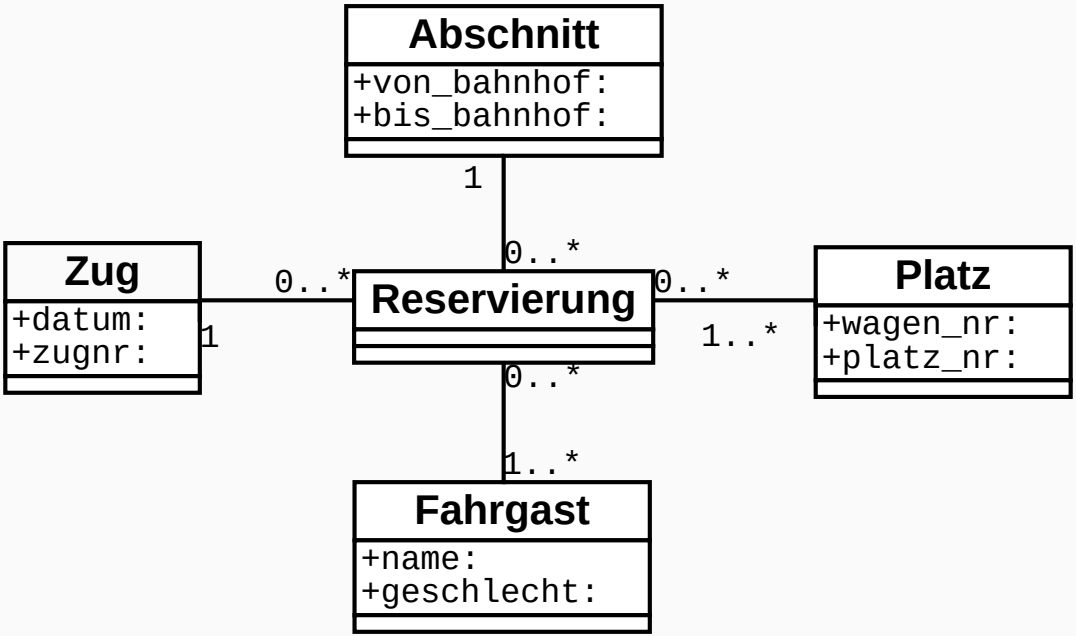
Mehrgliedrige Assoziationen

Beispiel für eine viergliedrige Assoziation: *Fahrgäste* reservieren *Plätze* für *Abschnitte* von *Zügen*.



Mehrgliedrige Assoziationen (2)

Auch hier gibt es eine Umformung in normale Assoziationen:

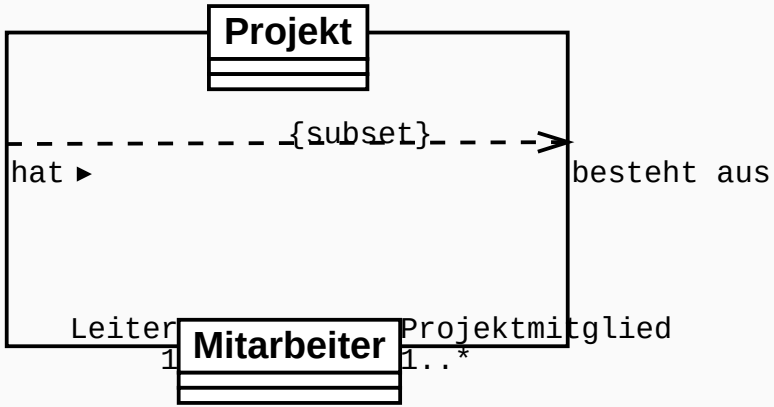


Formale Zusicherungen

Neben Attributen können beliebige UML-Elemente mit *Zusicherungen* versehen werden. Dies geschieht entweder *frei formuliert* oder mit Hilfe der UML-eigenen *Object Constraint Language*, kurz OCL.

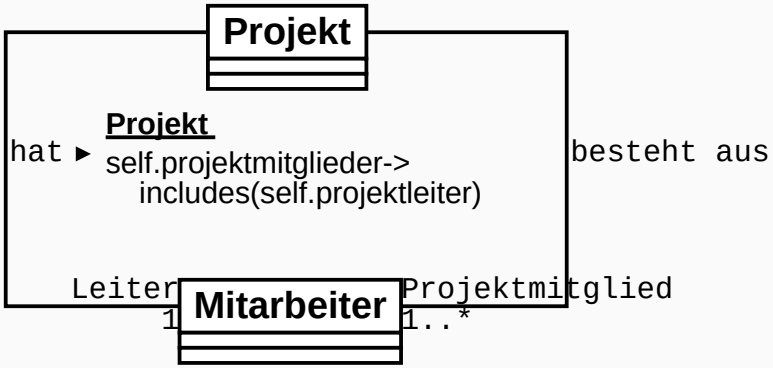
Beispiel: Der *Leiter* eines Projekts rekrutiert sich aus den Reihen der *Projektmitglieder.*

Informale Zusicherung



Die Zusicherung ist als $\{subset\}$ frei formuliert – mit der Bedeutung $projektleiter \in projektmitglieder$.

Formale Zusicherung



Der OCL-Ausdruck

$$self.projektmitglieder \rightarrow includes(self.projektleiter)$$

bedeutet, daß die Mengenoperation *includes*, die auf die Menge der Projektmitglieder angewandt wird, und der der Projektleiter als Argument übergeben wird, den Wert *true* liefern muß.

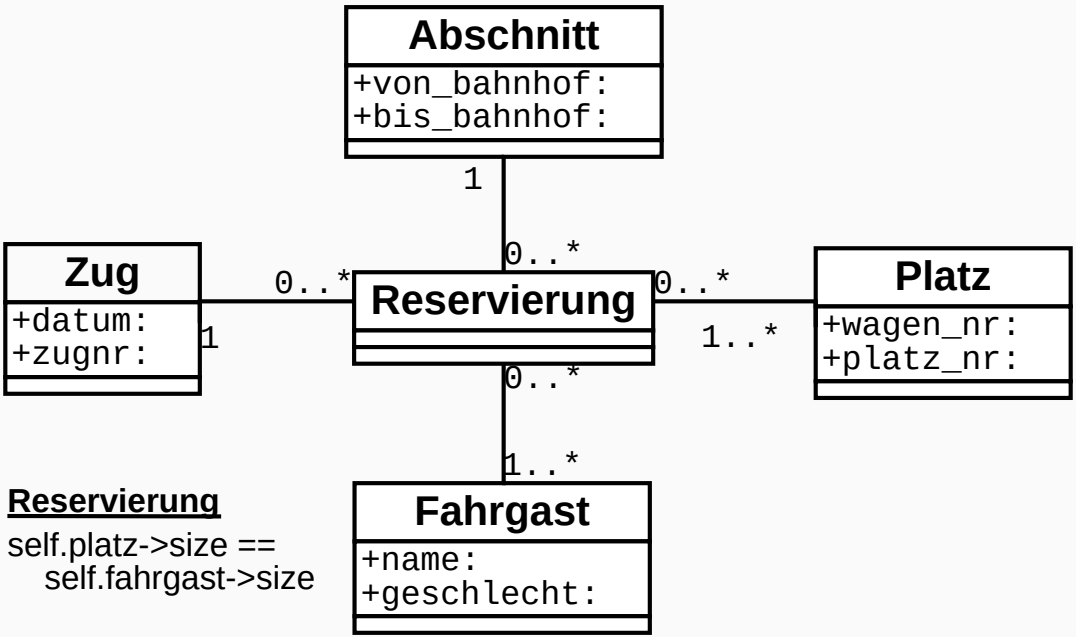






Beispiel: Reservierungen

Die Anzahl der reservierten Plätze muß mit der Anzahl der Fahrgäste übereinstimmen



Die Operation *size* gibt die Anzahl der Elemente zurück.

Weitere OCL-Beispiele

Das Gehalt des Mitarbeiters muß kleiner sein als das Gehalt des Chefs:

```
mitarbeiter.gehalt < chef.gehalt
```

Eine Person muß mindestens eine Anschrift haben

```
not person.anschriften->isEmpty   oder
person.anschriften->size >= 1
```

Alle Fahrer eines Mietwagens müssen älter als 21 sein

```
self.fahrer->forAll(f | f.alter >= 21)
```

Solche Zusicherungen können ggf. in *Zusicherungen der Programmiersprache* übersetzt werden, die dann zur Laufzeit Verletzungen aufdecken.

Sequenzdiagramme

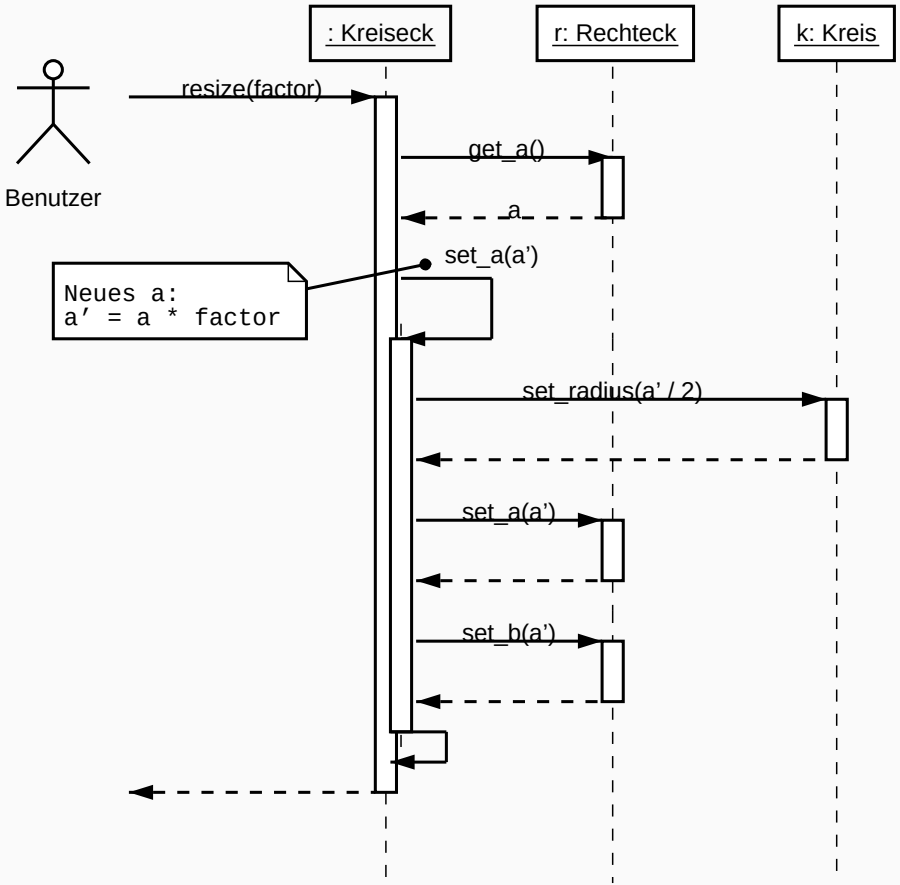
Ein *Sequenzdiagramm* gibt den Informationsfluß zwischen einzelnen Objekten wieder mit Betonung des *zeitlichen Ablaufs.*

Objekte werden mit senkrechten *Lebenslinien* dargestellt; die Zeit verläuft von oben nach unten.

Der *Steuerungsfokus* (breiter Balken) gibt an, welches Objekt gerade aktiv ist.

Pfeile („Nachrichten“) kennzeichnen Informationsfluß – z.B. Methodenaufrufe (durchgehende Pfeile) und Rückkehr (gestrichelte Pfeile).

Beispiel: Vergrößern eines Kreisecks



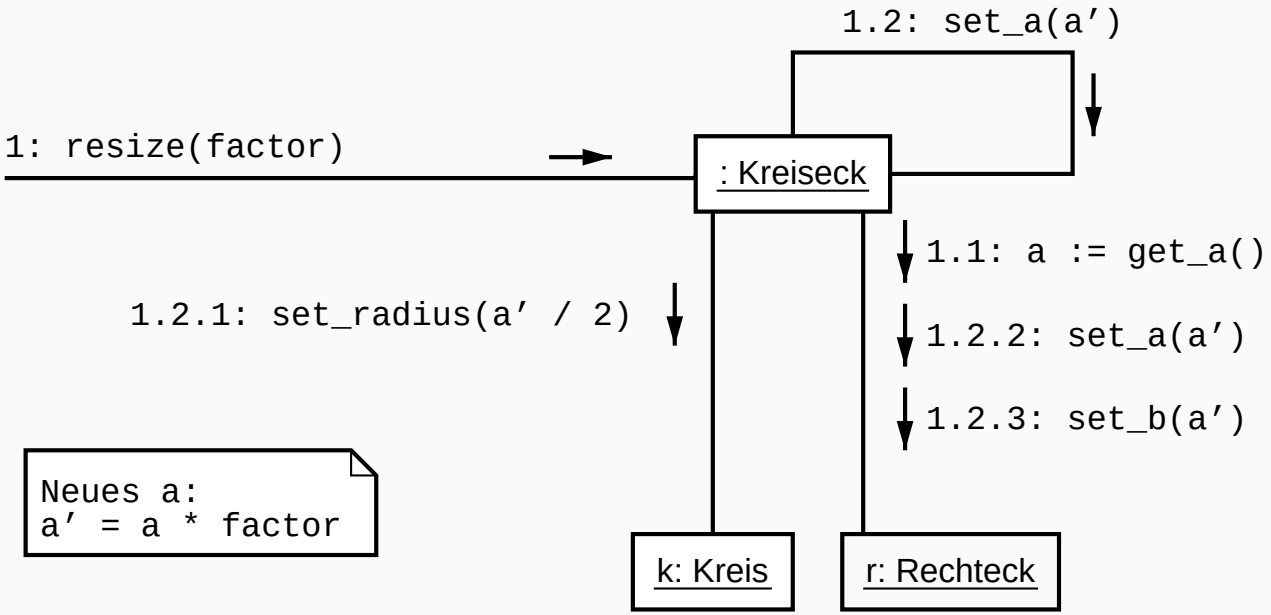
Kollaborationsdiagramme

Kollaborationsdiagramme beschreiben ebenfalls den Informationsfluß zwischen einzelnen Objekten.

Im Gegensatz zu Sequenzdiagrammen betonen sie aber die *Beziehungsstruktur* der Objekte.

Beispiel: Vergrößern eines Kreisecks

Das bekannte Szenario – diesmal als Kollaborationsdiagramm:



Im Kollaborationsdiagramm werden die Nachrichten *durchnumeriert.*

Die Gliederung gibt dabei an, welche Objekte noch aktiv sind.

Zustandsdiagramme

Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Ereignisse Zustandsänderungen stattfinden.

Ein Zustandsdiagramm zeigt einen *endlichen Automaten*.







Zustandsübergänge

Zustandsübergänge werden in der Form

$$\mathit{Ereignisname}[\mathit{Bedingung}]/\mathit{Aktion}$$

notiert. Hierbei ist

- *Ereignisname* der Name eines Ereignisses (typischerweise ein Methodenaufruf);
- *Bedingung* die Bedingung, unter der der Zustandsübergang stattfindet (optional)
- *Aktion* eine Aktion, die beim Übergang ausgeführt wird (optional)

Auch Zustände können mit Aktionen versehen werden: Das Ereignis `entry` kennzeichnet das Erreichen eines Zustands; `exit` steht für das Verlassen eines Zustands.

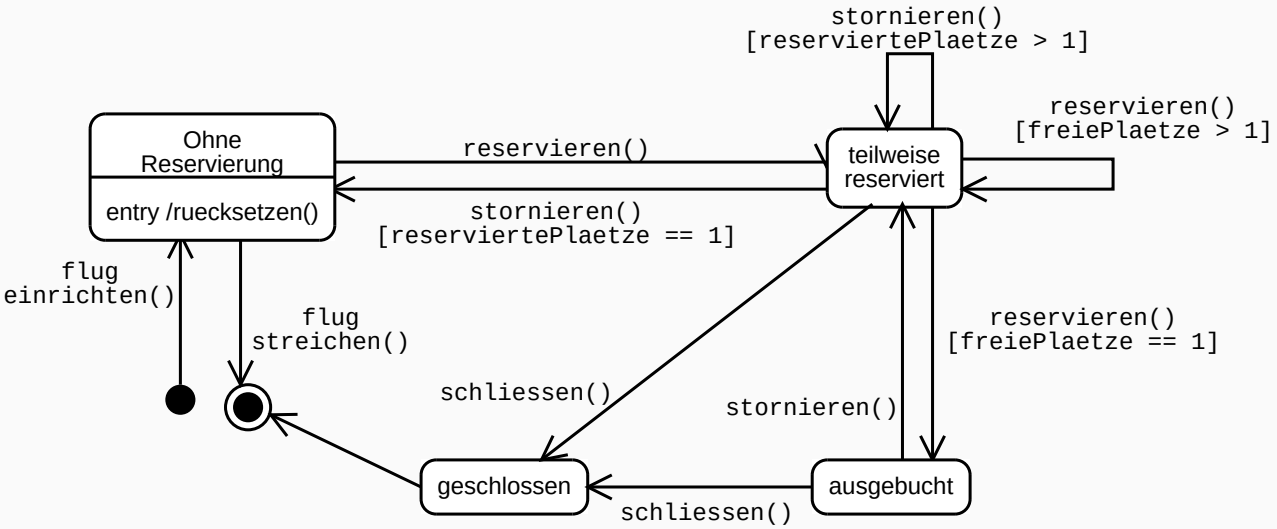






Beispiel: Flugreservierung

Wird ein Flug eingerichtet, ist noch nichts reserviert. Die Aktion `ruecksetzen()` sorgt dafür, daß die Anzahl der freien und reservierten Plätze zurückgesetzt wird.



Fallstudie: Tabellenkalkulation

Eine *Tabelle* besteht aus $m \times n$ *Zellen*

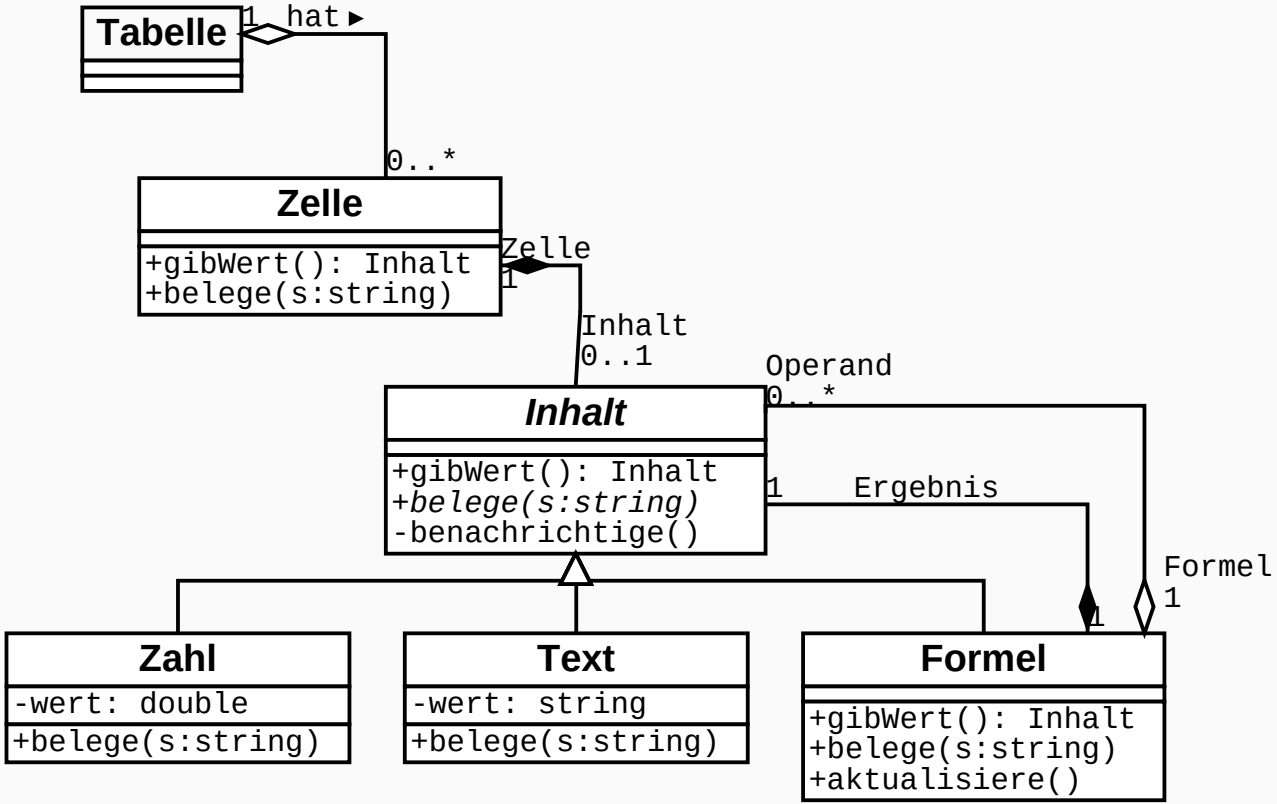
Zellen sind entweder leer oder haben einen *Inhalt*

Inhalte sind *Zahlen*, *Texte* oder *Formeln*

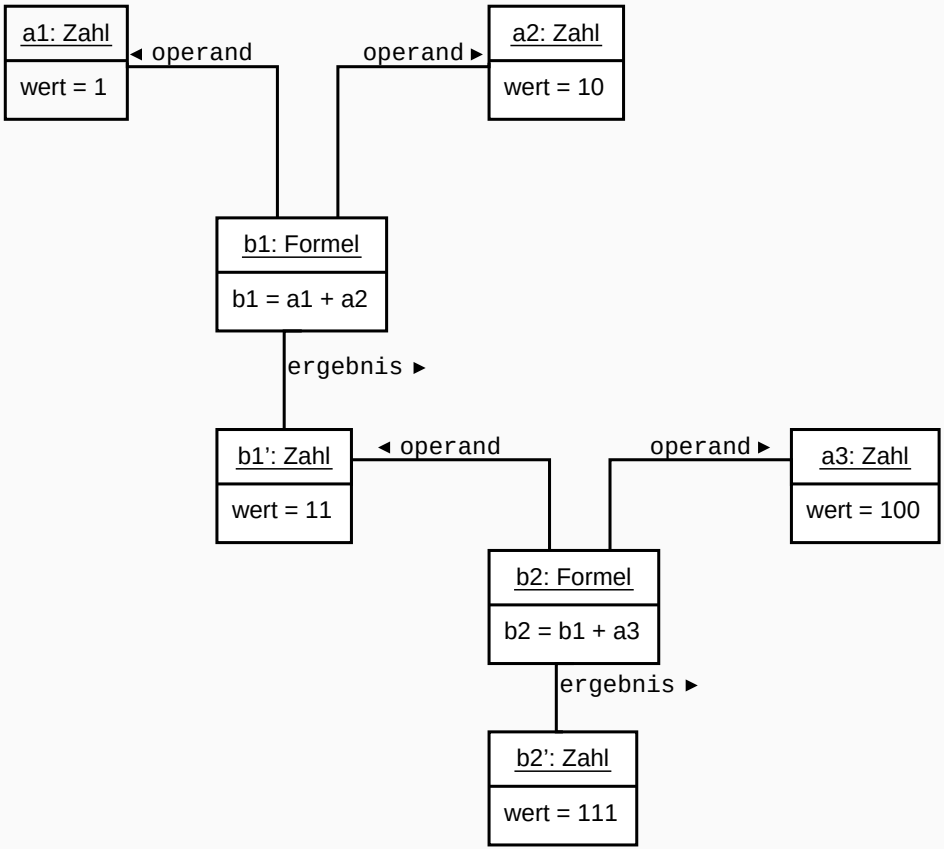
Zu einem Inhalt gibt es mehrere Formeln (die die Inhalte ansprechen)

Zu einer Formel gibt es mehrere Inhalte (die als Operanden in der Formel auftreten)

Objektmodell

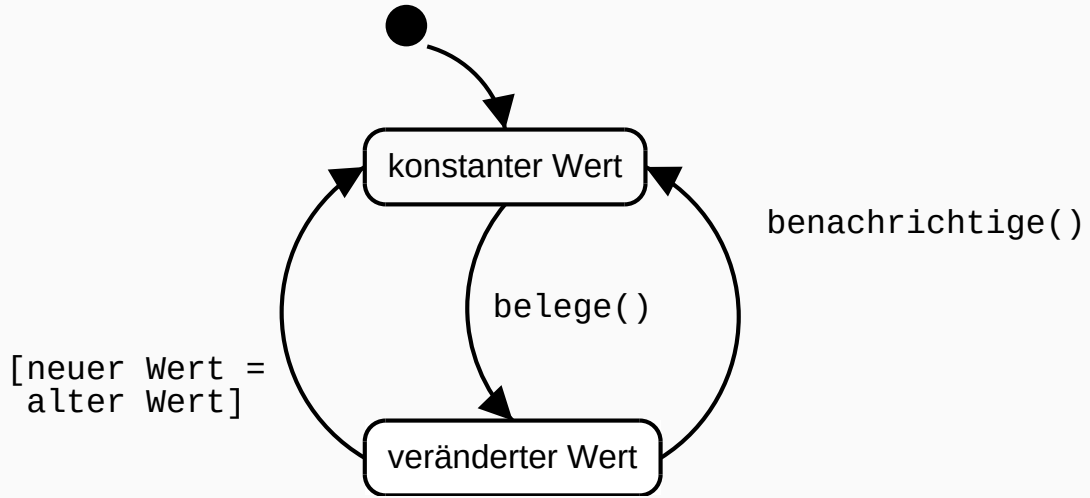


Beispiel für Objektbeziehungen



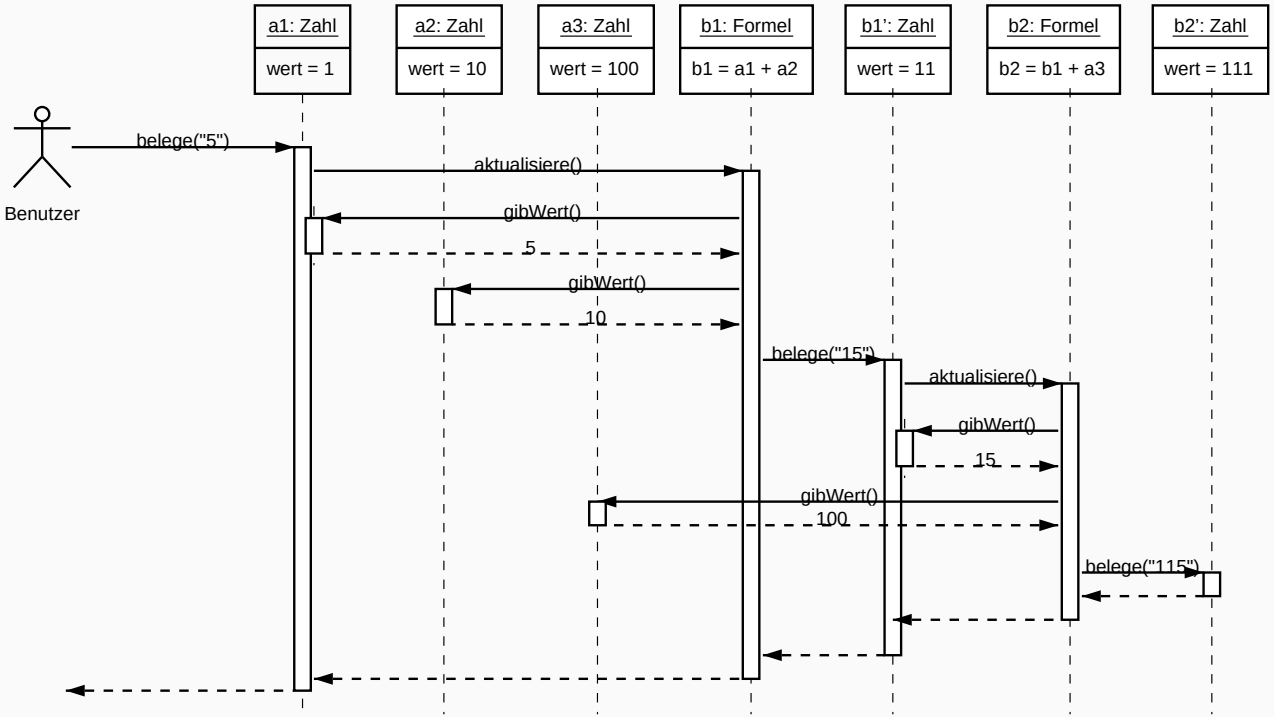
Zustandsdiagramm

Die Methode `belege` der Klasse `Inhalt` prüft, ob sich der Wert geändert hat. Wenn ja, benachrichtigt sie mit `benachrichtige()` alle Formelobjekte, in denen der Inhalt als Operand vorkommt.



Sequenzdiagramm

Beispiel: Die Tabelle sei belegt wie beschrieben; nun ändern wir den Wert der Zelle A1 von 1 auf 5.



Finden von Klassen und Methoden

„Wie finde ich die Objekte?“ ist die schwierigste Frage der Analyse.

Es gibt keine eindeutige Antwort: Man kann ein Problem auf verschiedene Weisen objektorientiert modellieren.

Entwurf nach Zuständigkeit

Ein verbreitetes Grundprinzip ist der *Entwurf nach Zuständigkeit*: Jedes Objekt ist für bestimmte Aufgaben zuständig und

- besitzt entweder die Fähigkeiten, die Aufgabe zu lösen,
- oder es kooperiert dazu mit anderen Objekten.

Ziel ist damit *das Auffinden von Objekten anhand ihrer Aufgaben in der Kooperation.*

Entwurf nach Zuständigkeit (2)

Wir gehen zunächst von einer *informalen Beschreibung* aus und betrachten *zentrale Begriffe der Aufgabenstellung*:

Hauptworte in der Beschreibung werden zu *Klassen* und *konkreten Objekten*

Verben in der Beschreibung werden zu *Diensten* –

- entweder zu Diensten, die ein Objekt anbietet,
- oder zu Aufrufen von Diensten kooperierender Objekte.

Diese Dienste kennzeichnen die *Zuständigkeit* und *Kooperationen* der einzelnen Klassen.

Entwurf nach Zuständigkeit (3)

Die so gefundenen Klassen werden dann anhand ihrer Zuständigkeiten auf sogenannte *KZK-Karten* (Klasse – Zuständigkeit – Kooperation) notiert:

Klassenname	*Zusammenarbeit mit*
zuständig für	

Die KZK-Karte gibt die *Rolle* wieder, die Objekte im Gesamtsystem übernehmen sollen.

Beispiel: Rechnerversand

Student Fritz bestellt mit einem Brief bei der Firma Rechnerwelt einen Rechner. Dieser wird ihm nach mehreren Tagen als Paket durch die Firma Gelbe Post zugestellt.







Eine erste Näherung

Student	*Zusammenarbeit mit*
zuständig für	
bestellen	Rechnerhändler
Paket annehmen	Zusteller

Rechnerhändler	*Zusammenarbeit mit*
zuständig für	
Bestellung annehmen	Student
Paket absenden	Zusteller

Zusteller	*Zusammenarbeit mit*
zuständig für	
Paket annehmen	Rechnerhändler
Paket abgeben	Student

Verfeinerung

Diese erste Näherung ist jedoch noch nicht vollständig:

- Fritz tritt in seiner Rolle als Kunde auf; es kommt nicht darauf an, daß er auch Student ist (es sei denn, er bekäme Studentenrabatt). Besser wäre also der Klassenname **Kunde** statt **Student**.
- Brief und Paket fehlen – es handelt sich um reine Datenobjekte, die weder Zuständigkeit noch Kooperationspartner haben.
- Wir haben offengelassen, wie der Bestellbrief zum Rechnerhändler gelangt; ggf. ist hierfür ebenfalls ein Zusteller zuständig.
- *Datenfluß*, *Zustandsübergänge* und *Klassenhierarchien* sind nicht berücksichtigt.

Überarbeitung des Entwurfs

Ein erster Grobentwurf kann meistens deutlich verbessert werden:

Herausfaktorisieren gleicher Merkmale. *Können gemeinsame Merkmale (Attribute, Methoden) verschiedener Klassen miteinander in Zusammenhang gebracht werden?*

Diese gemeinsamen Merkmale können

- in eine *dritte Klasse* verlagert werden, die von den bestehenden Klassen aggregiert wird. Die bestehenden Klassen müssen die ausgelagerten Dienste aber weiterhin anbieten
- in eine *gemeinsame Oberklasse* verlagert werden. Dies ist bei gemeinsamen *ist-ein*-Beziehungen sinnvoll.







Überarbeitung des Entwurfs (2)

Verallgemeinerung von Verhaltensweisen. *Können Methoden mit einheitlicher Schnittstelle bereits auf einer abstrakten Ebene angegeben werden?*

Abstrakte Klassen können etwa allgemeine Methoden bereitstellen, deren Details (abgeleitete Kernmethoden) in den konkreten Unterklassen realisiert werden.

Ersetzen einer umfangreichen Klasse durch ein Teilsystem. *Können Klassen mit vielen Merkmalen weiter unterteilt werden?*

Evtl. Einführung eines Teilsystems aus mehreren Objekten und zugehörigen Klassen

Überarbeitung des Entwurfs (3)

Minimierung von Objektbeziehungen. *Kann man durch Umgruppierung der Klassen oder neue Schnittstellen die Zahl der „Benutzt“-Beziehungen verringern?*

Auch hier unterhält nur noch ein neu einzuführendes Teilsystem Außenbeziehungen.

Wiederverwendung, Bibliotheken. *Kann man bestehende Klassen wiederverwenden?*

Ggf. können geeignete *Anpassungsklassen* (Adapter) eingeführt werden







Überarbeitung des Entwurfs (4)

Generizität. *Können generische Klassen und Methoden eingesetzt werden?*

Oder auch: können Klassen und Methoden des Entwurfs generisch gestaltet werden?

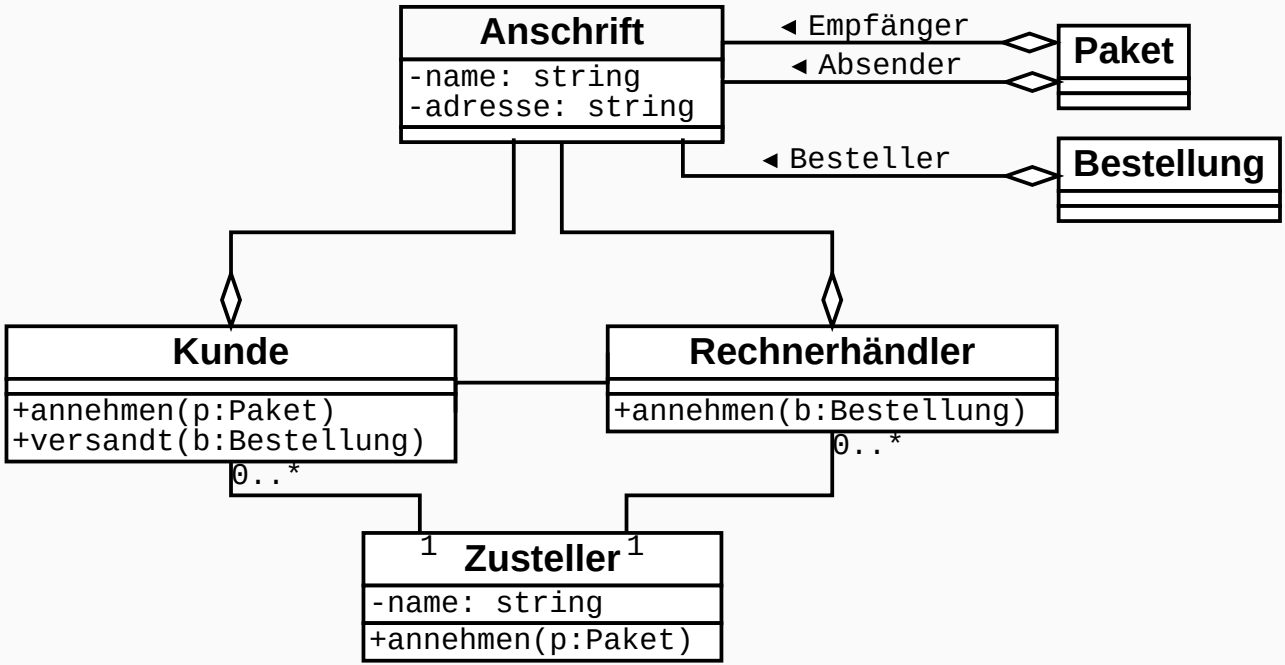
Entwurfsmuster. *Kann man Standard-Architekturbausteine einsetzen?*



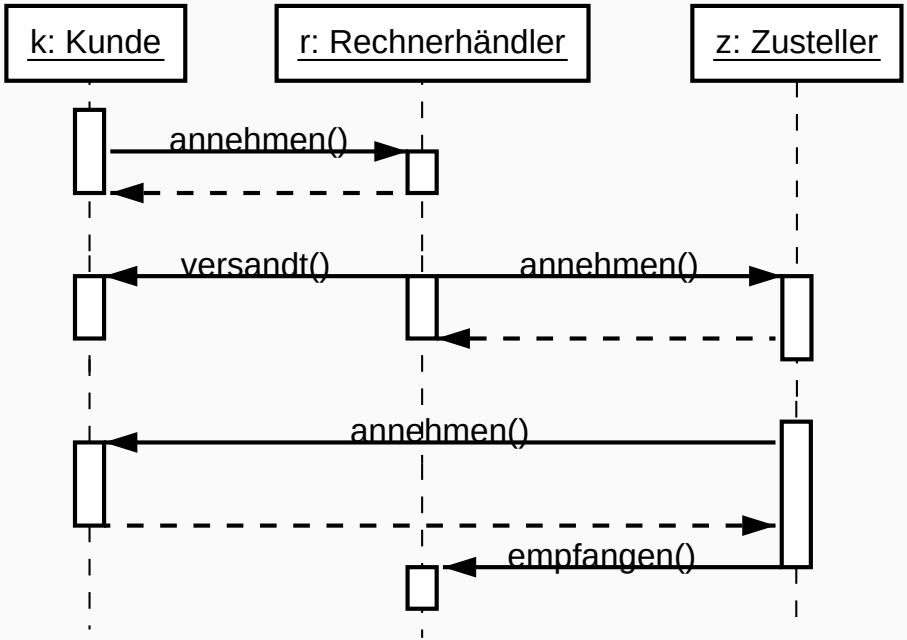




Objektmodell Rechnerversand



Sequenzdiagramm Rechnerversand



Vom Modell zum Programm

Wie verwandelt man den Entwurf in einen Quelltext?

- Klassen und Vererbung können direkt aus dem Klassendiagramm entnommen werden. Die Methoden(köpfe) ebenso. Zu jeder Methode muß eine vollständige Signatur angegeben werden.
- Assoziationen zwischen Klassen werden durch Attribute realisiert.
  - Eine $n:1$ oder $1:1$ Assoziation von P nach Q wird in P durch ein Attribut q vom Typ Q realisiert.
  - Eine $1:n$- bzw. $n:m$-Assoziation von P nach Q wird durch eine Menge qs vom Typ $set(Q)$ realisiert. (Implementierung als Feld, Liste...)

Vom Modell zum Programm (2)

- Falls Assoziationen eigene Methoden haben, muß man sie als eigene (Hilfs-) Klassen implementieren.
- Für jede Klasse sollte eine Invariante formuliert und dokumentiert werden; für jede Methode eine Vor- und Nachbedingung.
- Die Methodenrümpfe werden implementiert. Dabei wird vorgegangen wie in der traditionellen Programmierung.







Vom Modell zum Programm (3)

- Überprüfung des Zustandsdiagramms: sind genau die im dynamischen Modell angegebenen Aufrufsequenzen zulässig? Unzulässige Aufrufe müssen durch Ausnahme/Fehlerbehandlung abgefangen werden! Dazu kann es sinnvoll sein, die Vorbedingung dynamisch zu überprüfen.
- Zum Testen werden die üblichen Verfahren angewendet.

Codeschablonen

Moderne Programmierumgebungen erzeugen aus einem Entwurf automatisch *Codeschablonen*:

1. Man entwirft ein System mit allen Klassen und Attributen.
2. Die Programmierumgebung erzeugt entsprechende Codeschablonen.
3. Nun müssen „nur noch“ die Methoden ausprogrammiert werden.







Checkliste: Grobentwurf

Der Grobentwurf im Rahmen des Software-Entwicklungs-Praktikums wird anhand der folgenden Anforderungen beurteilt.

- **Entsprechen die Klassen zentralen Konzepten der Aufgabenstellung?**

  Wichtige Hauptworte der Aufgabenstellung sollten als Klassen im Entwurf auftauchen.

- **Sind die Grundprinzipien der Zerlegung erfüllt?**







Checkliste: Grobentwurf (2)

- **Sind Hierarchien und Objektbeziehungen gut dokumentiert?**

  Dies soll in Form eines Objekt-Modells geschehen.

  Das Objekt-Modell sollte so präzise wie möglich sein (Multiplizitäten, Aggregation/Komposition, Zusicherungen).

  Im Entwurf sollten alle offensichtlichen Möglichkeiten für Restrukturierungen und Überarbeitungen bereits ausgeschöpft sein.







Checkliste: Grobentwurf (3)

- **Sind Informationsfluß und Zustandsübergänge hinreichend berücksichtigt?**

  Je nach Komplexität der Aufgabenstellung kann es notwendig sein,

  - ausgewählte Aspekte des Informationsflusses (Sequenzdiagramm, Kollaborationsdiagramm) und
  - Zustandsübergänge (Zustandsdiagramm)

  gesondert zu spezifizieren.