



# 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*.

# Was ist ein Objekt?

Ein Objekt bietet eine Sammlung von *Diensten* (Methoden), die auf einem gemeinsamen *Zustand* arbeiten.

Typischerweise entsprechen

**Objekte** Gegenständen aus der Aufgabenstellung (“Ameise”,  
„Spielfeld“, „Markierung“)

**Methoden** Verben aus der Aufgabenstellung („bewegen“,  
„setzen“, „löschen“)

# 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?

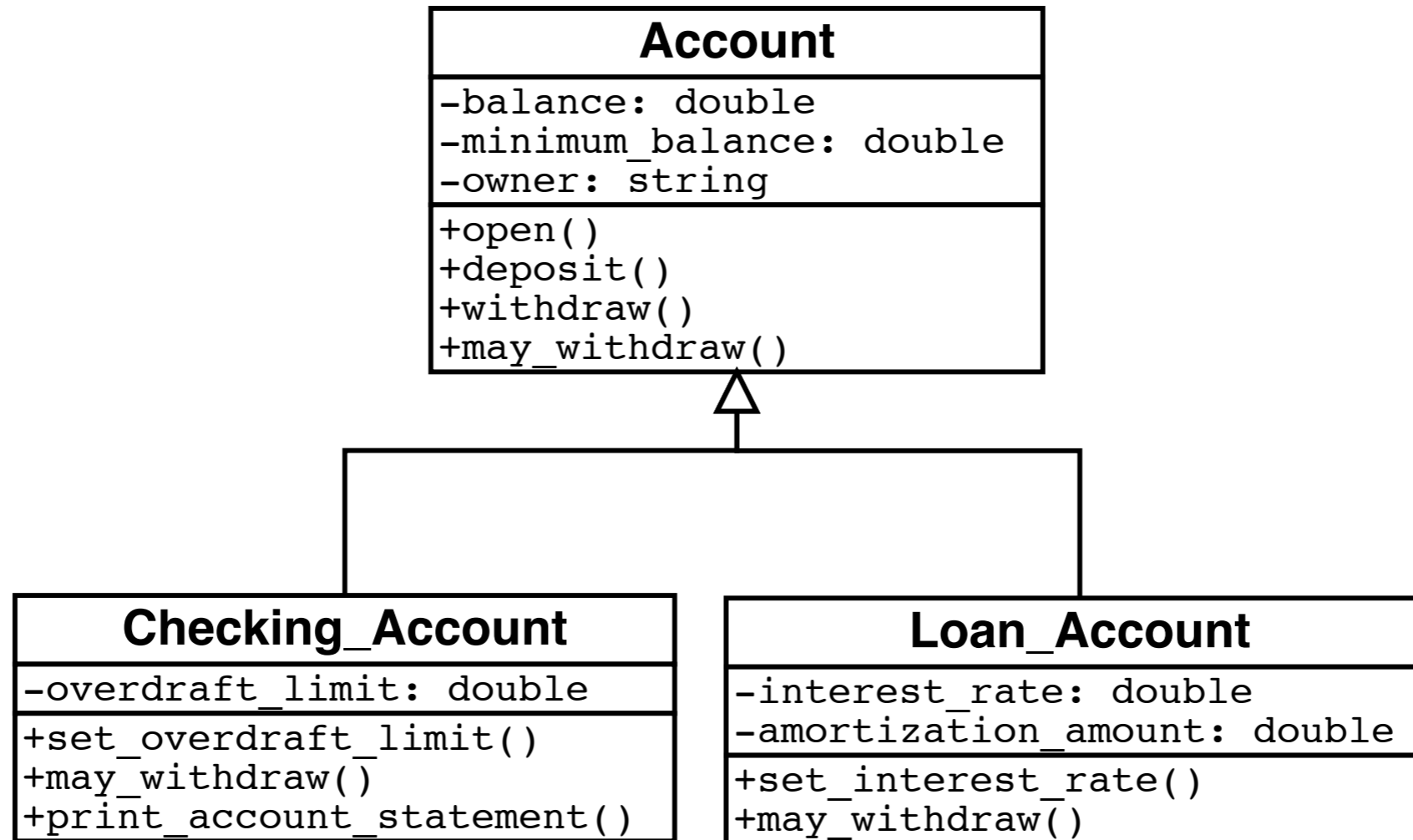
# Objekt-Modell: Klassendiagramm

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()`)

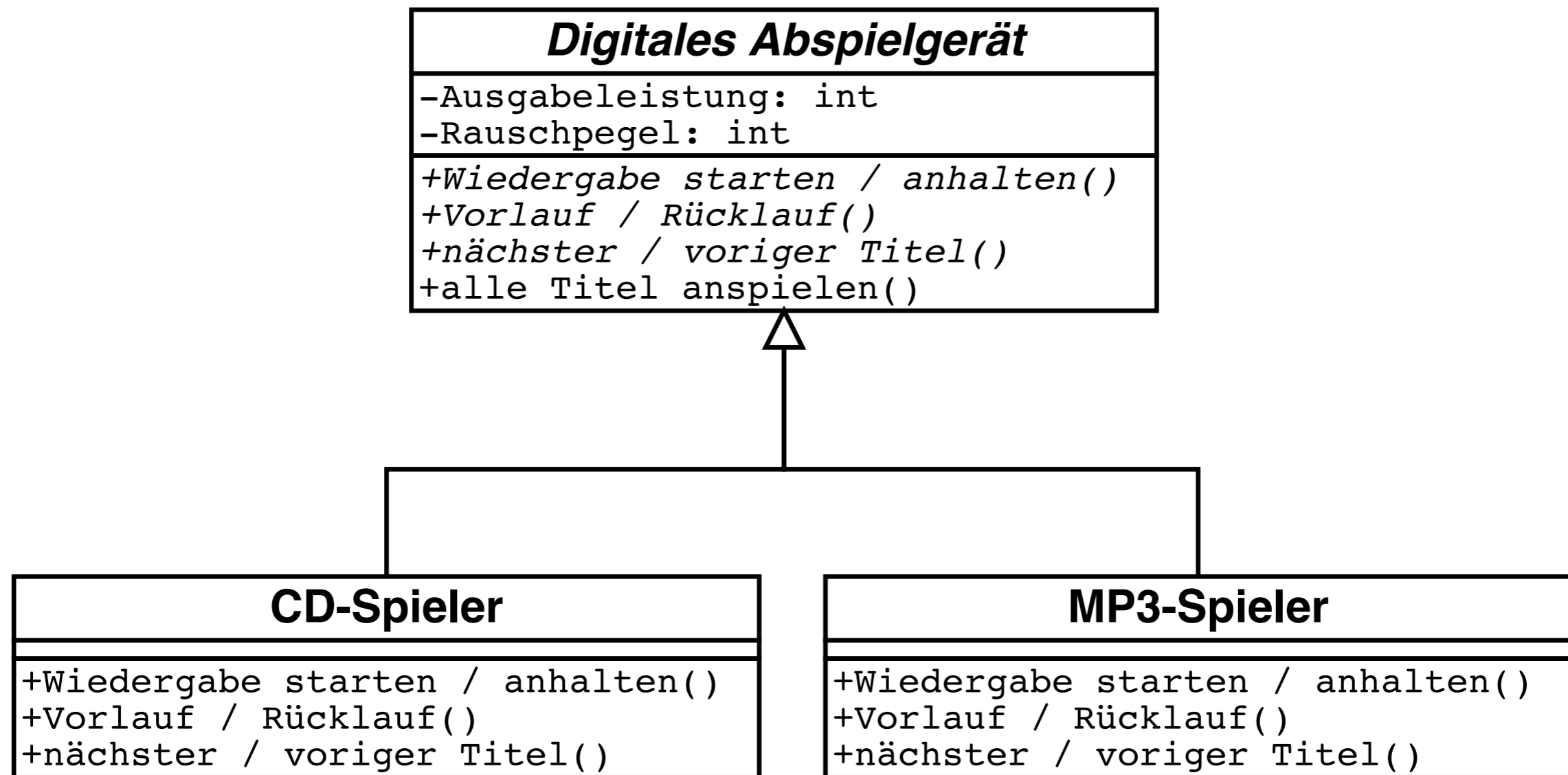
# Abstrakte Klassen und Methoden

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



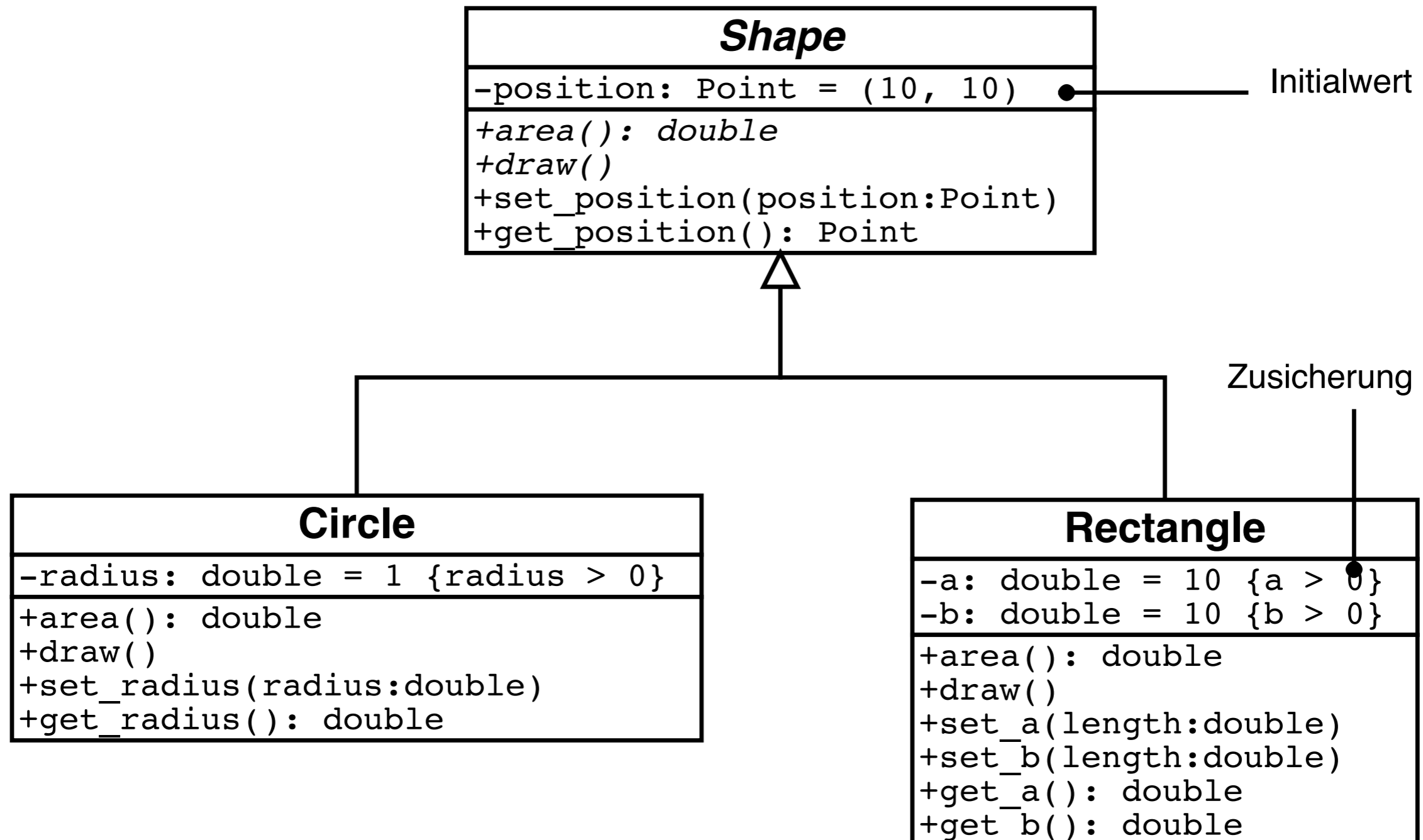
# Initialwerte und Zusicherungen

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.



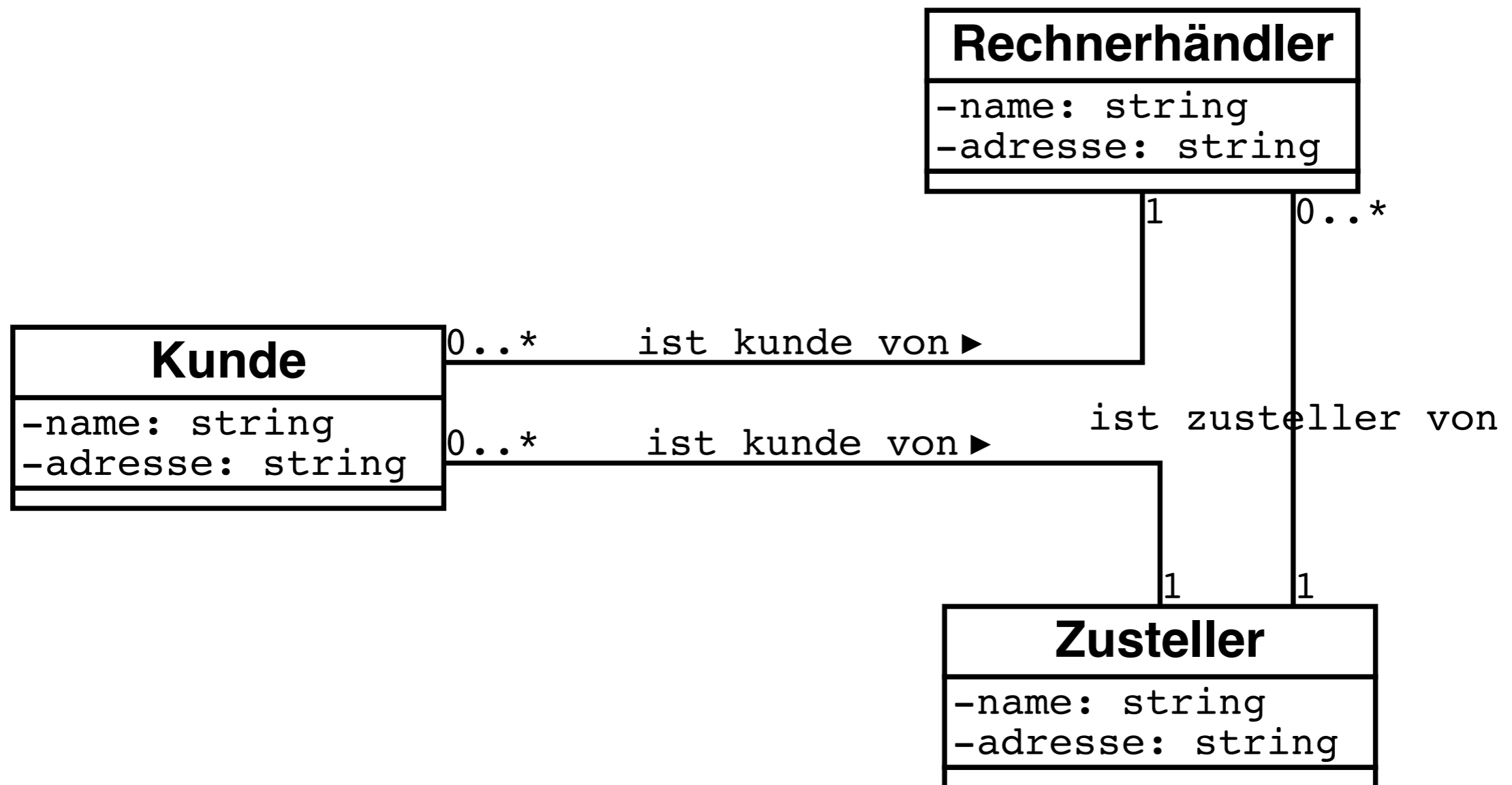
# Objekt-Modell: Assoziationen

## *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.

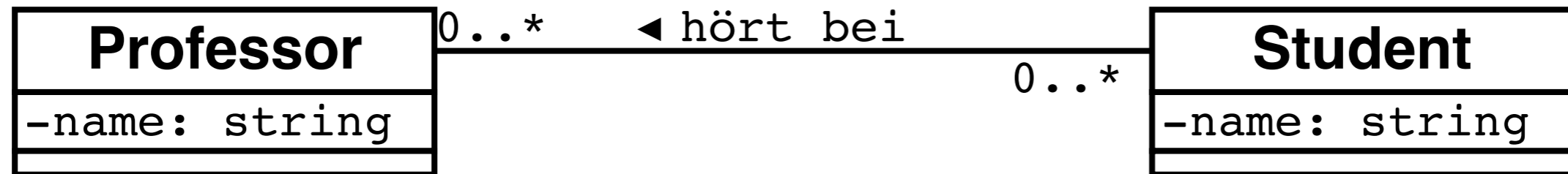
# Multiplizitätsangaben

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

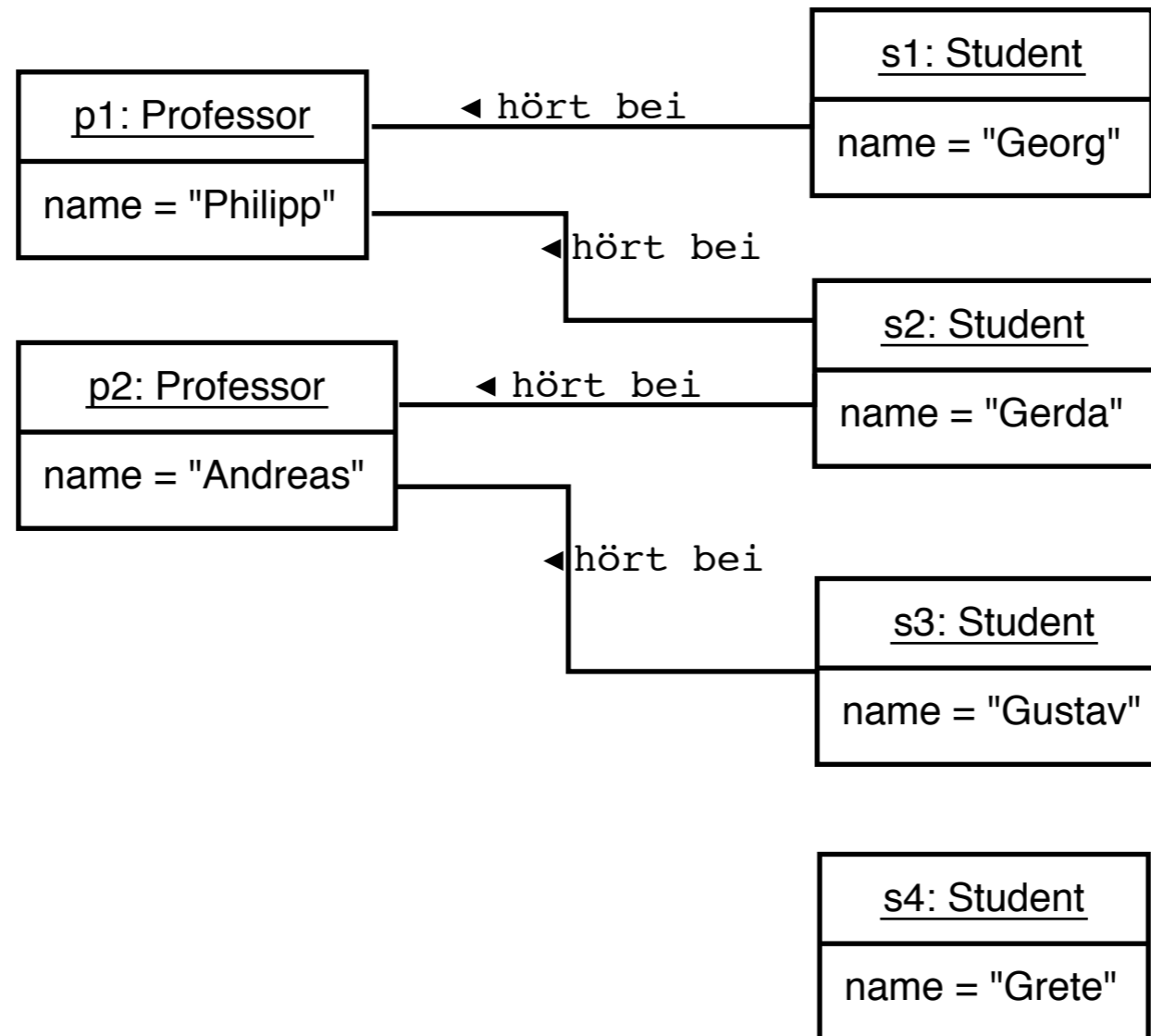


# Multiplizitätsangaben (2)

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



# Beispiel für Objektbeziehungen

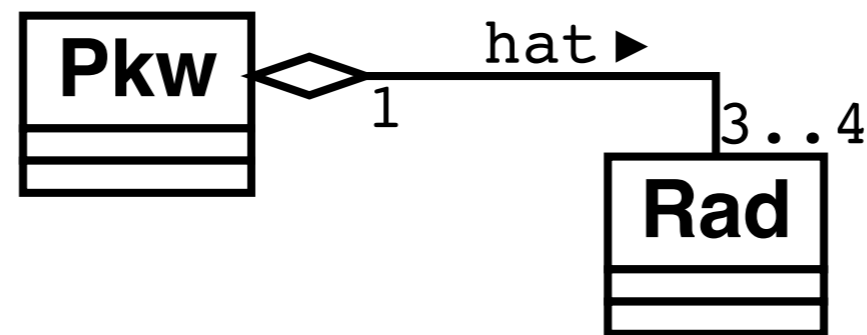


Darstellung von Exemplaren (Objekten) mit unterstrichenem Objektname; konkrete Werte für die Attribute

# Aggregation

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

Beispiel: Ein Pkw hat 3-4 Räder



# Ein Pkw



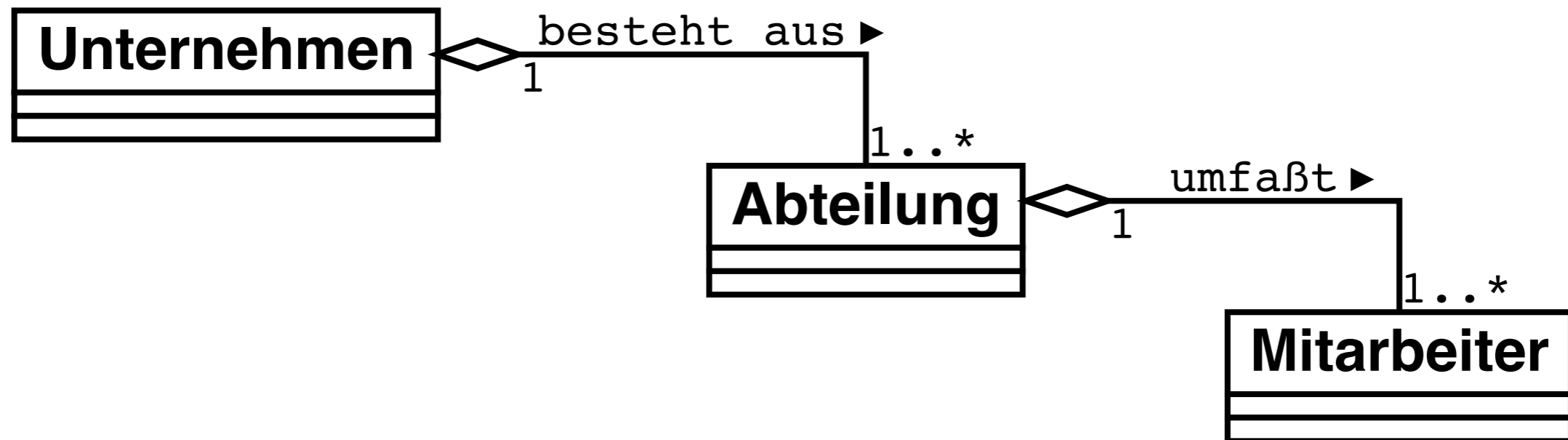


# Ein Pkw



# 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.

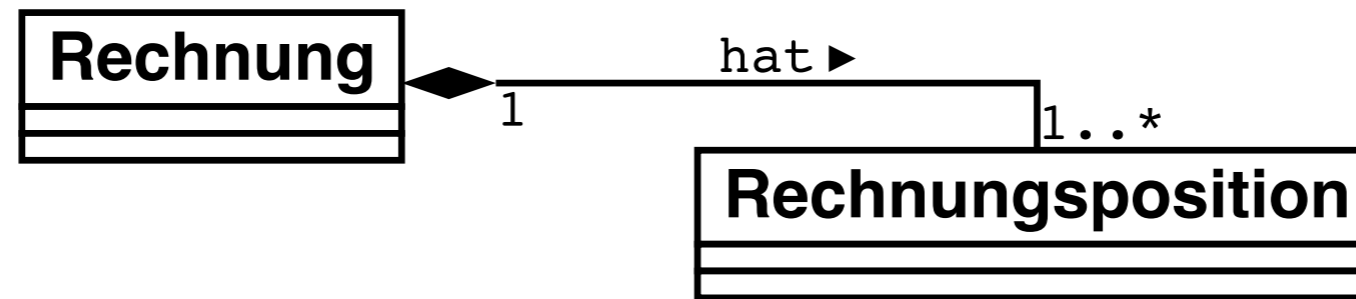
# Komposition

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.

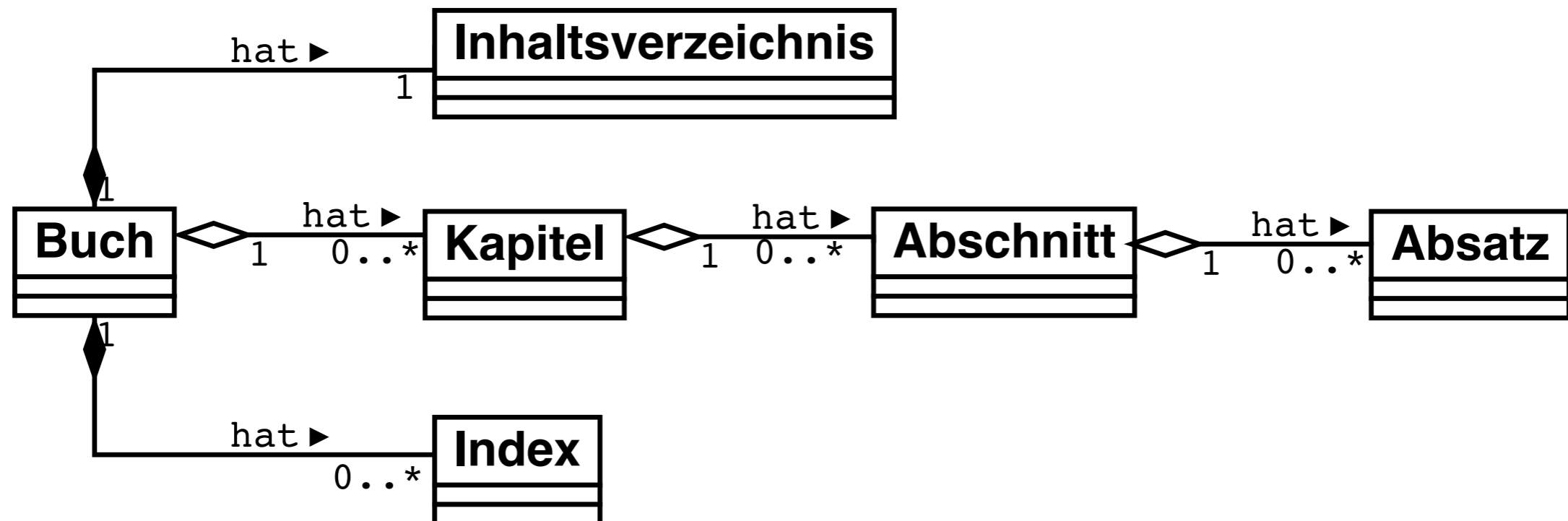
# Beispiel: Rechnungsposition

Eine Rechnungsposition gehört immer zu einer Rechnung.



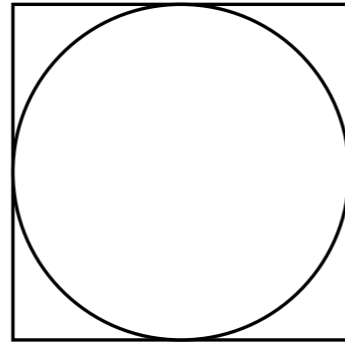
# Beispiel: Buch

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



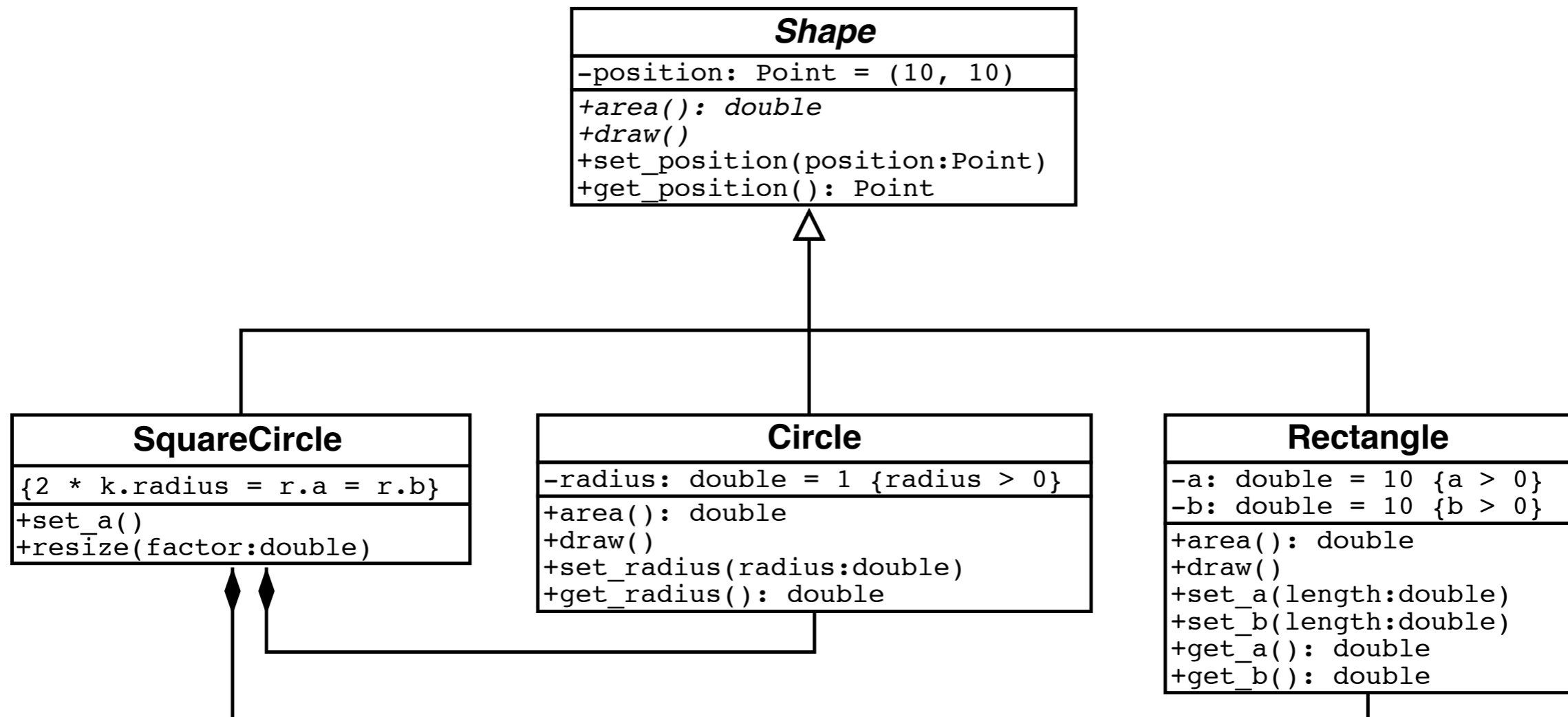
# Beispiel: Kreiseck

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.

# Sequenzdiagramme

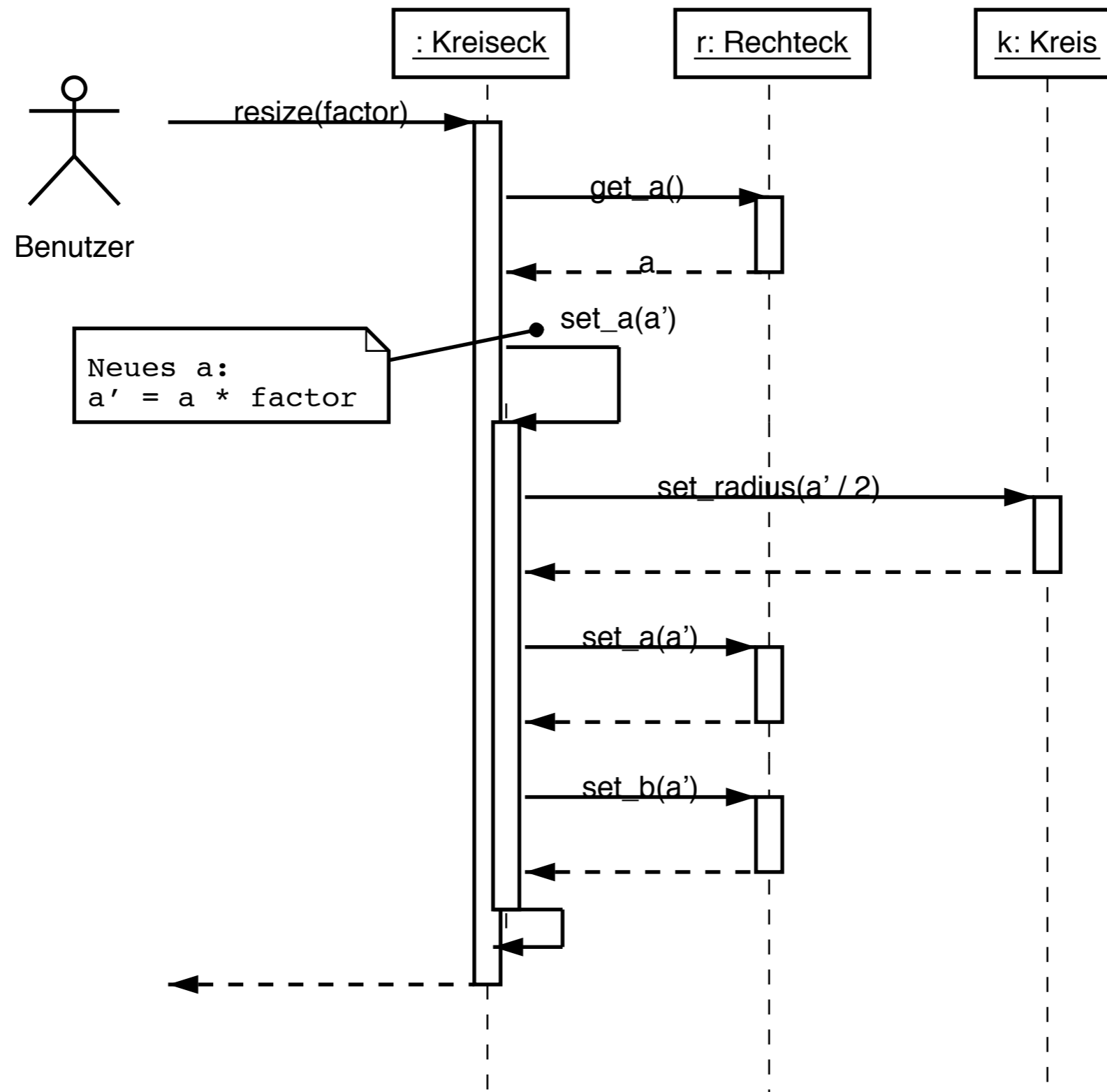
Ein *Sequenzdiagramm* gibt den Informationsfluß zwischen einzelnen Objekten wieder mit Betonung des *zeitlichen Ablaufs*.

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



# 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

*Ereignisname*[*Bedingung*] / *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: ADAC

The logo consists of the letters 'ADAC' in a bold, dark blue, sans-serif font, centered within a solid yellow rectangular background.

**ADAC**

# 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.

1. Beschreiben von typischen Szenarien mit *Use Cases*
2. Extrahieren der zentralen Klassen und Dienste aus den Use Cases



# Use Cases

(Anwendungsfälle)

- Beschreibt, wie ein Akteur (“Handelnder”) zu seinem Ziel kommt
- Welche *Akteure* gibt es, und welche *Ziele* haben sie?

# Definitionen

- Ein *Akteur* ist etwas, das sich verhalten kann eine *Person*, ein *System*, oder eine *Organisation*
- Ein *Szenario* ist eine Folge von Aktionen und Interaktionen zwischen Akteuren
- Ein *Use Case* (“Anwendungsfall”) ist eine Sammlung von verwandten Szenarien bestehend aus *Erfolgsszenario* und *Alternativszenarien*

# 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.*

- Wer sind die Akteure?
- Welche Ziele verfolgen sie?
- Was kann alles fehlschlagen?

# Alternative Szenarien

sind meist *spannender* als die Erfolgsszenarien

- **Beispiel: Parkhaus**  
Kein Zugang • Karte nicht lesbar • Stromausfall...
- **Beispiel: Geld abheben**  
Guthaben reicht nicht • Falsche PIN • Falsches Konto...
- **Beispiel: Flug buchen**  
Flug ausgebucht • Stornierungen...

# Beispiel: Rechnerversand

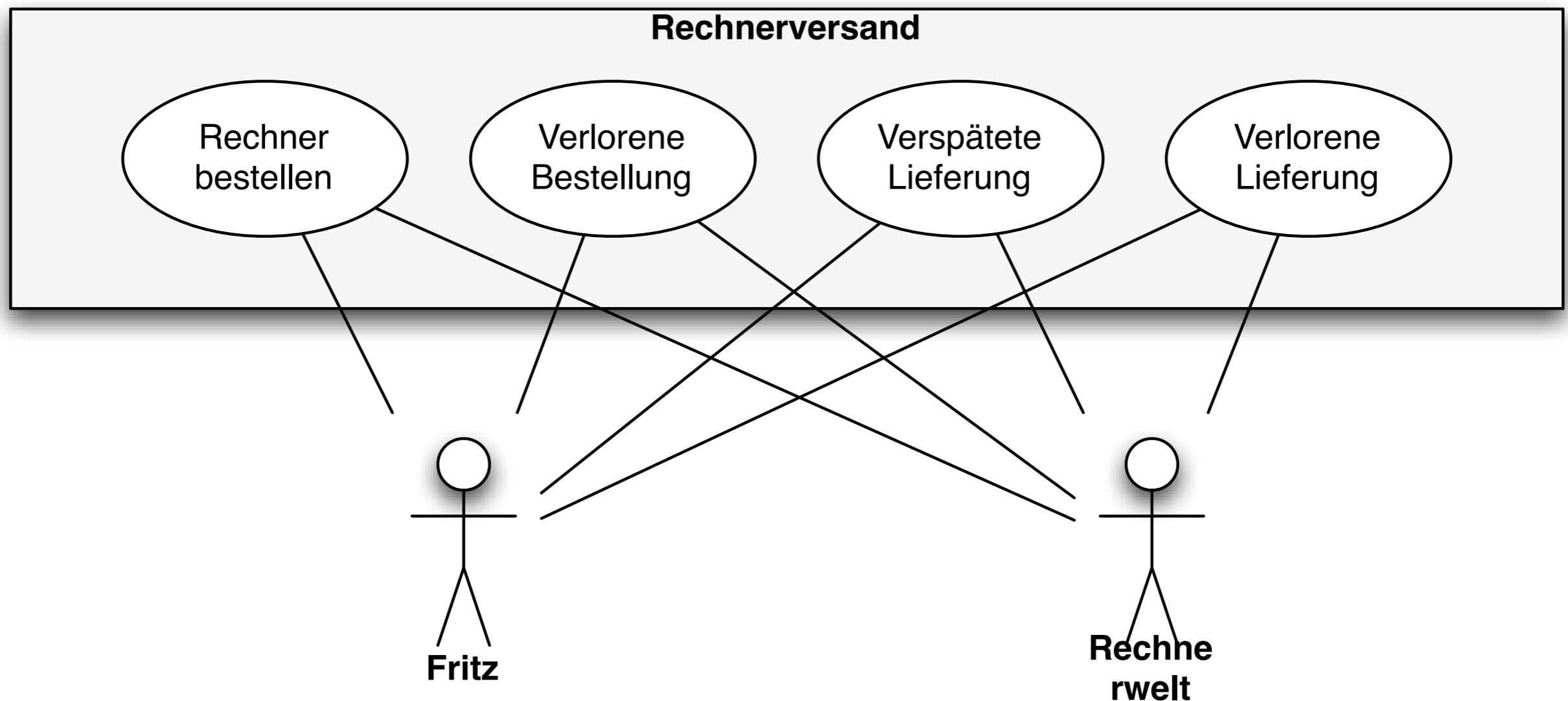
*Student Fritz bestellt mit einem Brief bei der Firma Rechnerwelt einen Rechner. Dieser wird ihm nach mehreren Tagen als Paket durch die Firma Gelbe Post zugestellt.*

## Alternative Szenarien:

- **Bestellung trifft nicht ein**
  - kann nicht ausgeführt werden –
- **Rechner (noch) nicht lieferbar**
  - Kunden verständigen; ggf. stornieren; gutschreiben –
- **Paket nicht zustellbar**
  - Kunden verständigen; ggf. stornieren; gutschreiben –

# Use Cases in UML

fassen Szenarien zusammen



# 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:

<i>Klassename</i>	<i>Zusammenarbeit mit</i>
<hr/> <i>zuständig für</i>	

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

# Eine erste Näherung

## Student

*zuständig für*  
bestellen  
Paket annehmen

*Zusammenarbeit mit*

Rechnerhändler  
Zusteller

## Rechnerhändler

*zuständig für*  
Bestellung annehmen  
Paket absenden

*Zusammenarbeit mit*

Student  
Zusteller

## Zusteller

*zuständig für*  
Paket annehmen  
Paket abgeben

*Zusammenarbeit mit*

Rechnerhändler  
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

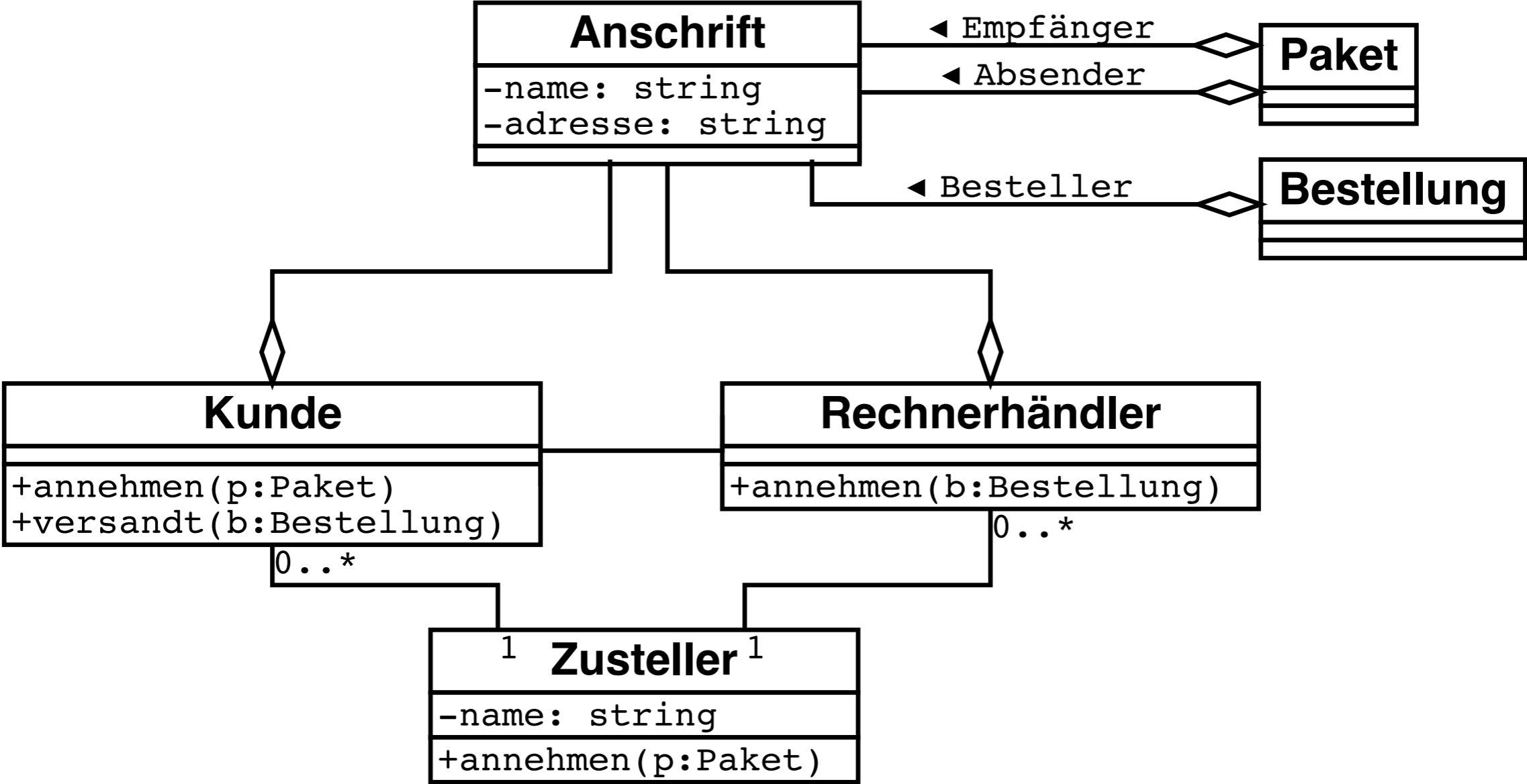
# Überarbeitung des Entwurfs (4)

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

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

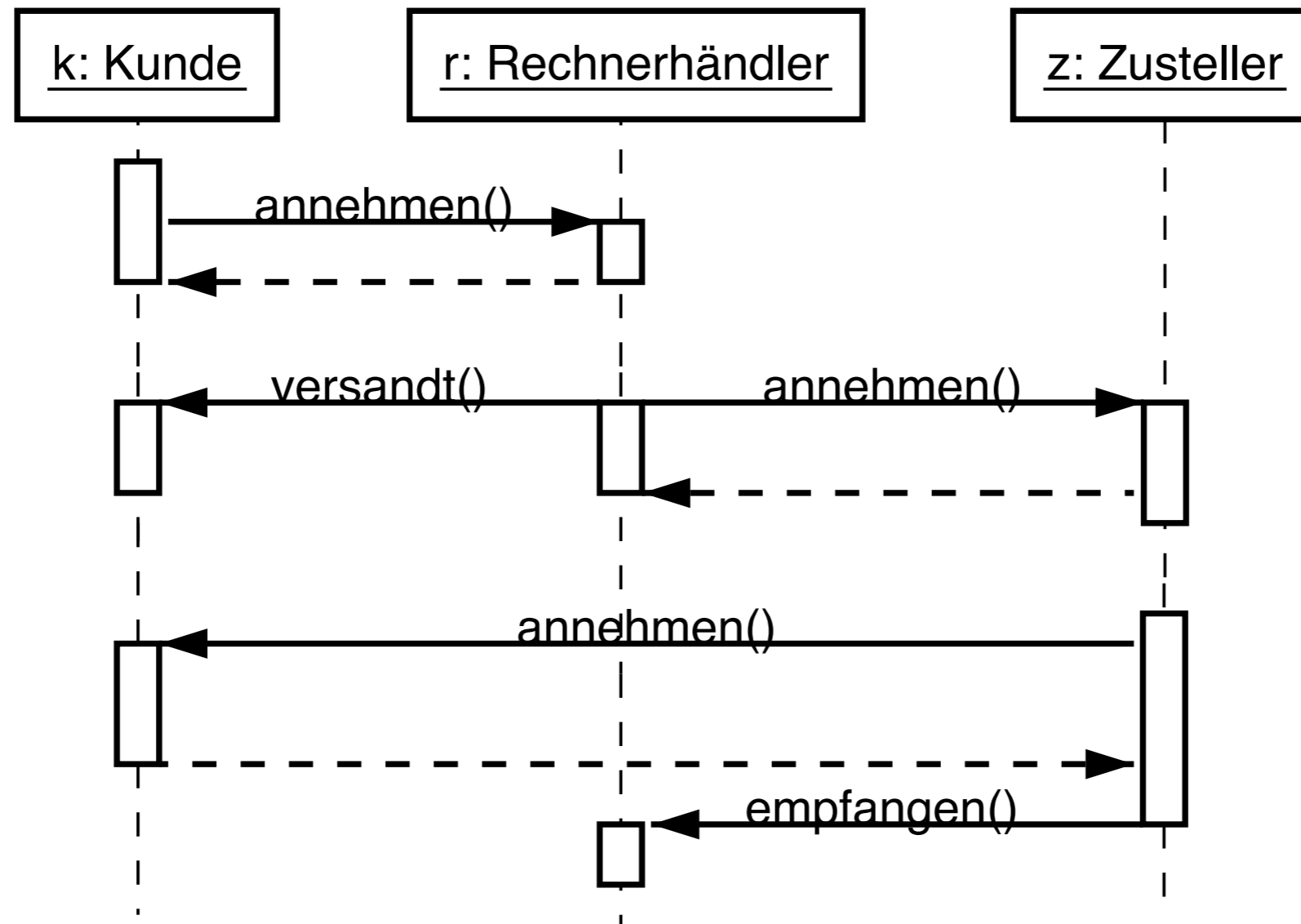
**Entwurfsmuster.** *Kann man Standard-Architekturbausteine einsetzen?*

# Objektmodell Rechnerversand





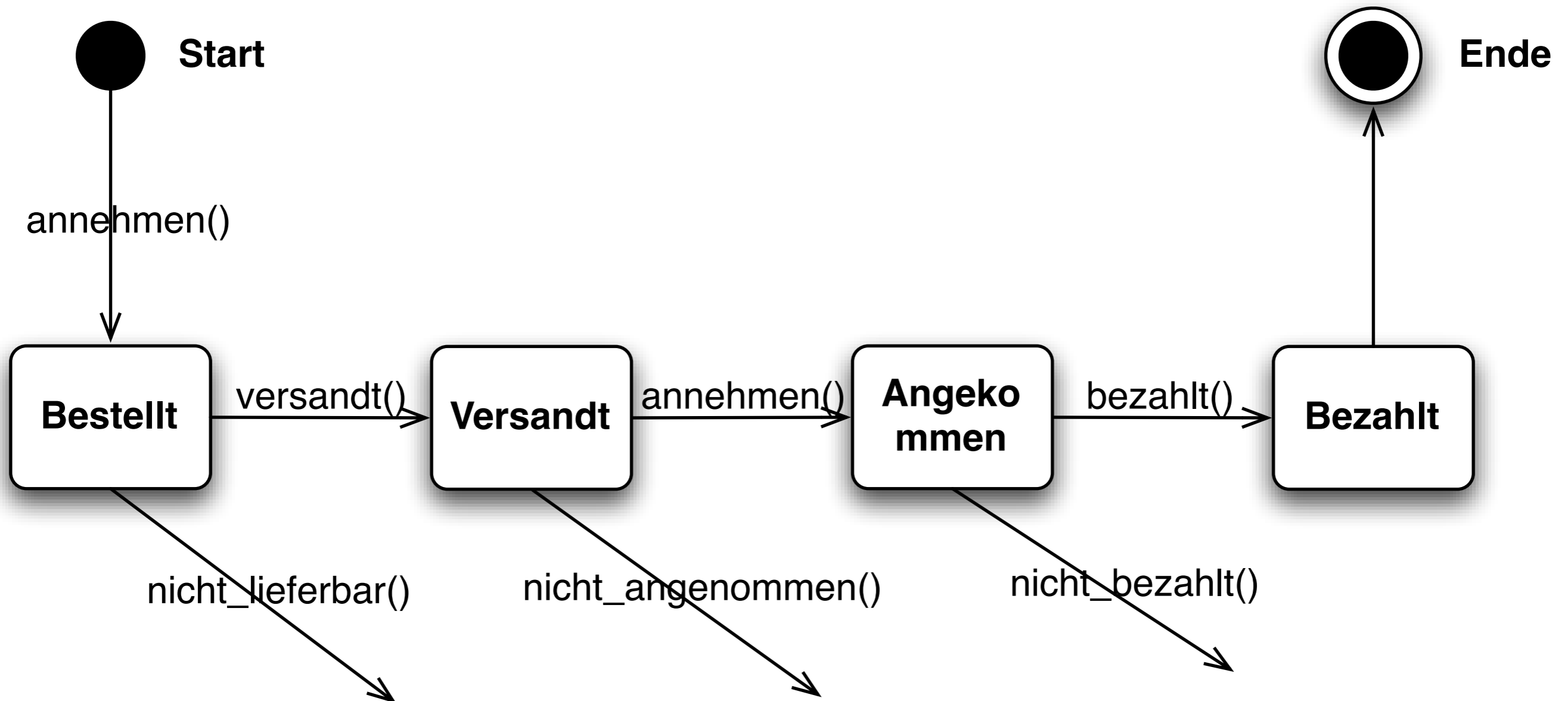
# Sequenzdiagramm Rechnerversand



- **Betrachtet nur den Erfolgsfall**  
Fehlschläge müssen gesondert beschrieben werden

# Zustandsdiagramm

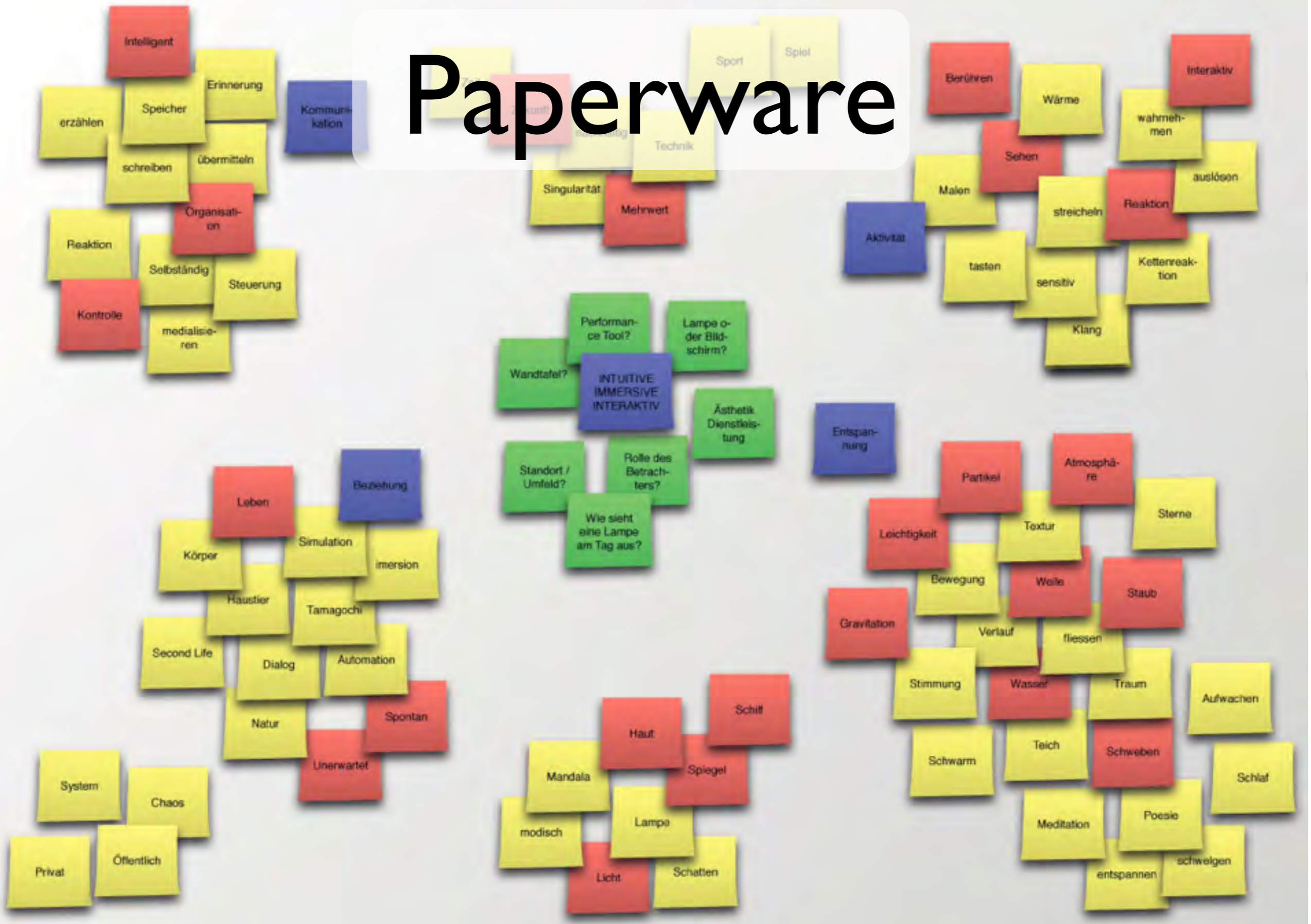
mit (einigen wenigen) Fehlschlägen



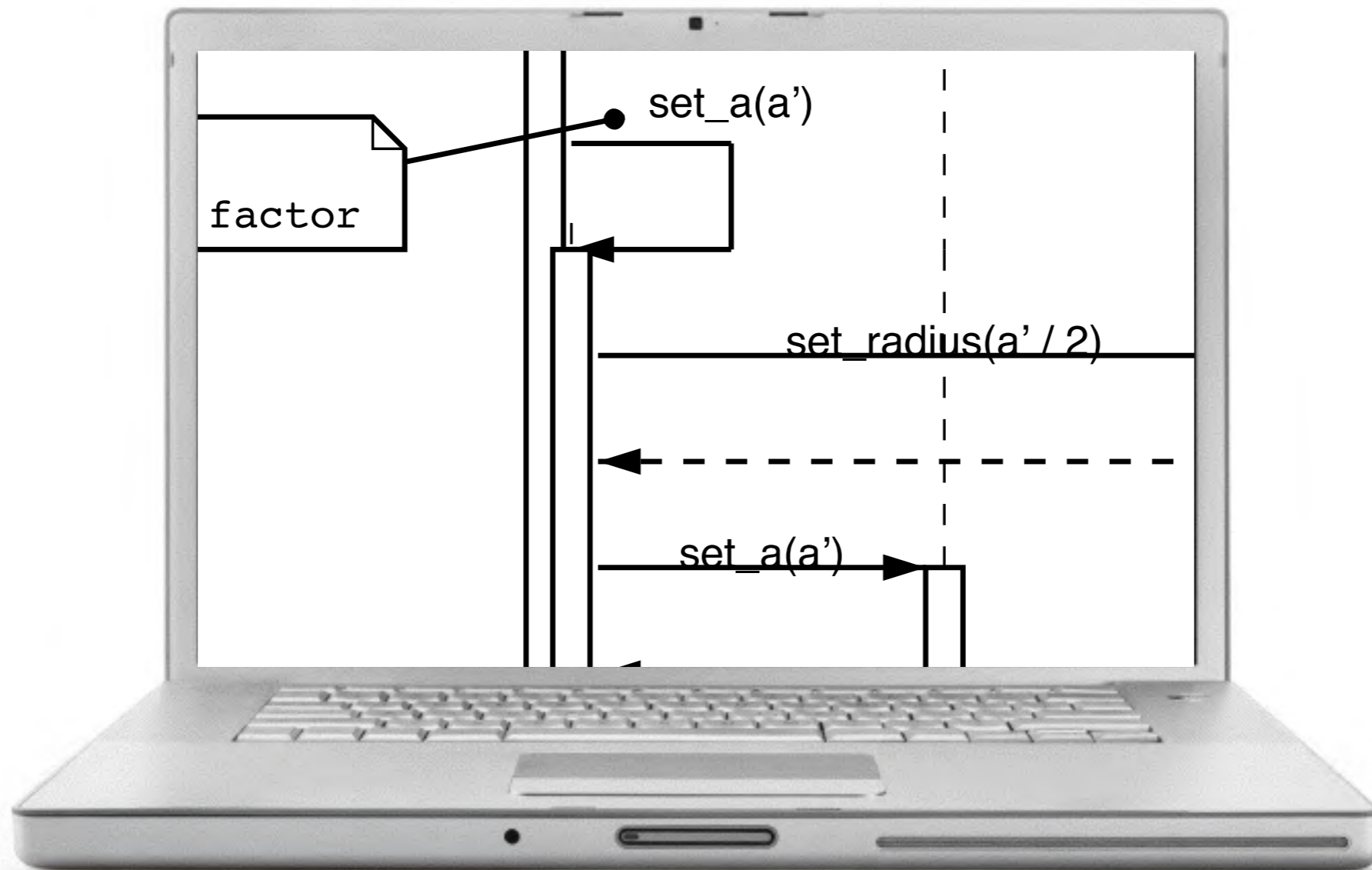
# Ein Wort zu KZK

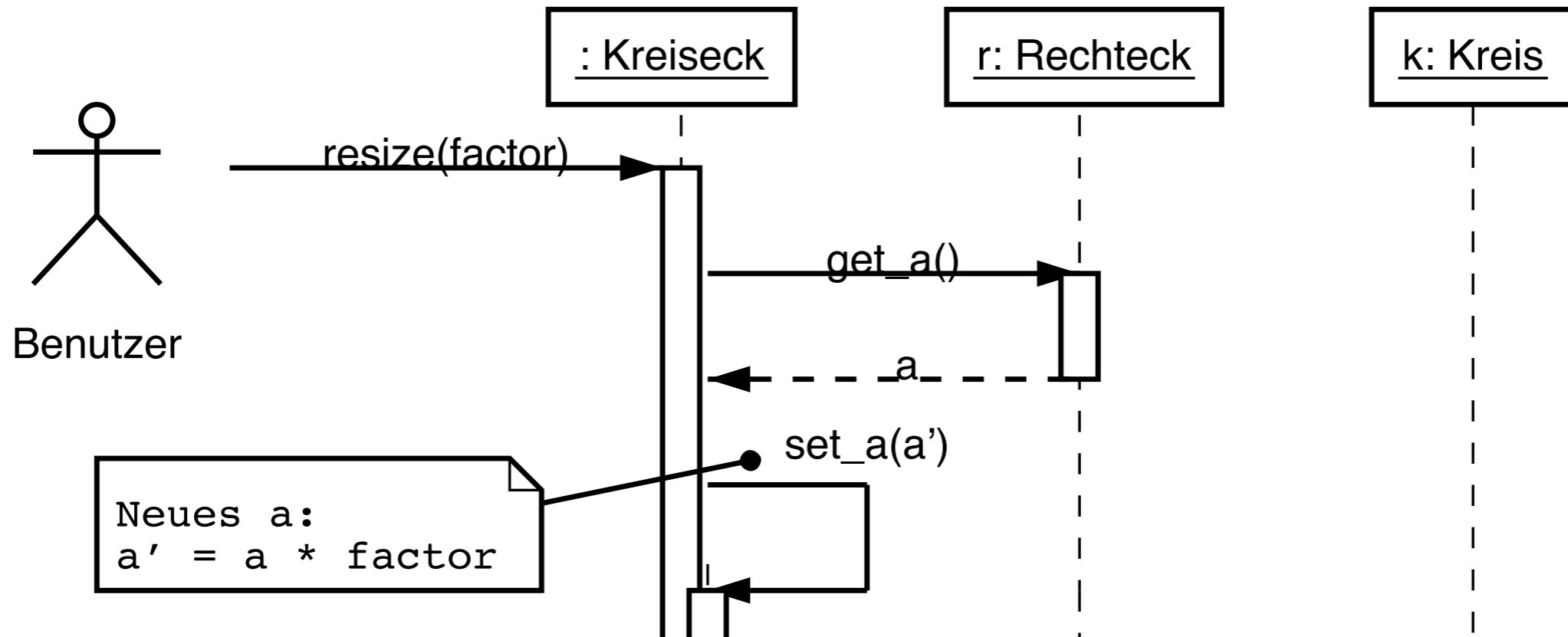
*“One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code.”* (C. Horstmann)

# Paperware

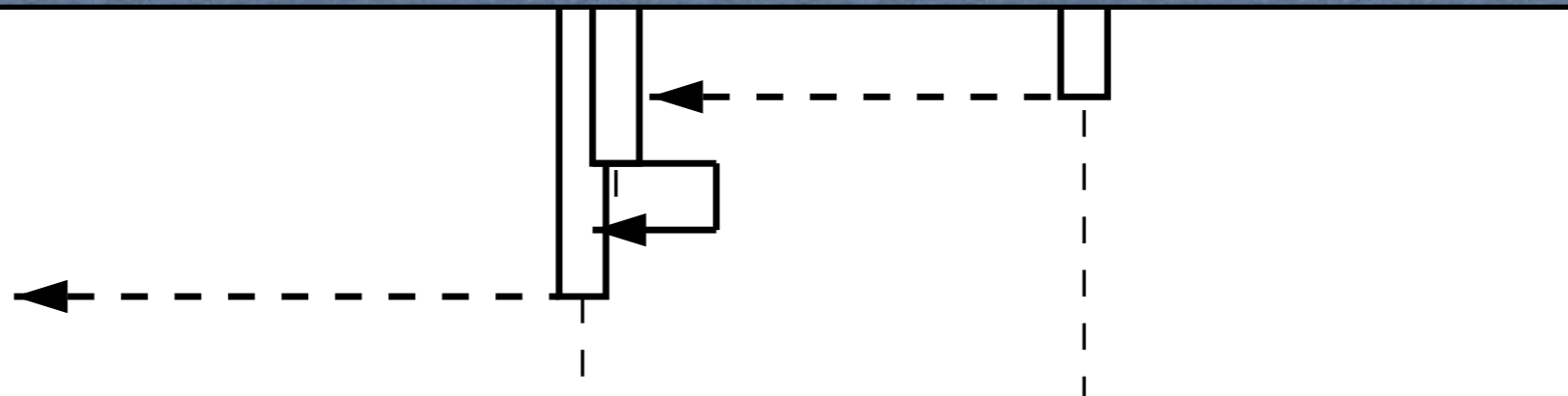


# Papier vs Laptop





Der *Einzelne* arbeitet am *Bildschirm*;  
die *Gruppe* arbeitet auf *Papier*!



# 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.

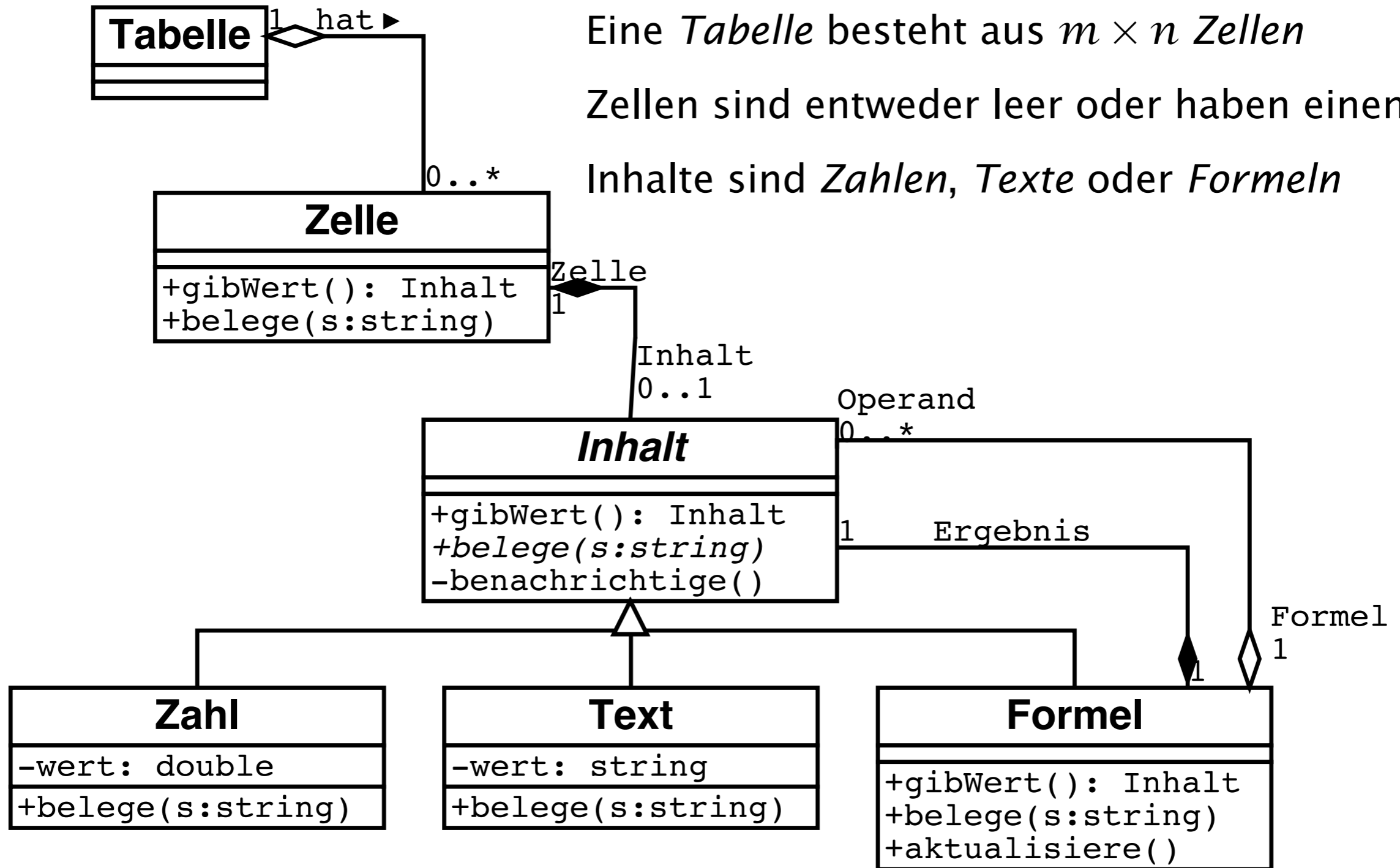
# Fallstudie: Tabellenkalkulation

Eine *Tabelle* besteht aus  $m \times n$  *Zellen*  
Zellen sind entweder leer oder haben einen *Inhalt*  
Inhalte sind *Zahlen*, *Texte* oder *Formeln*

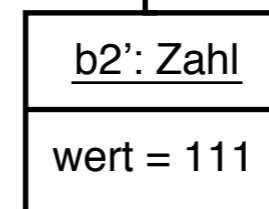
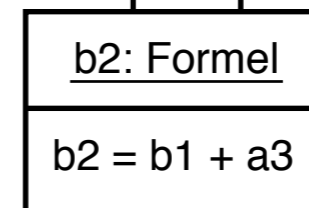
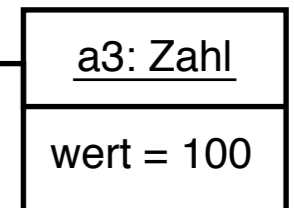
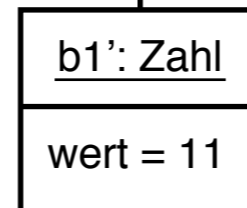
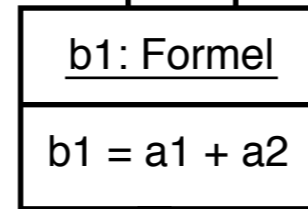
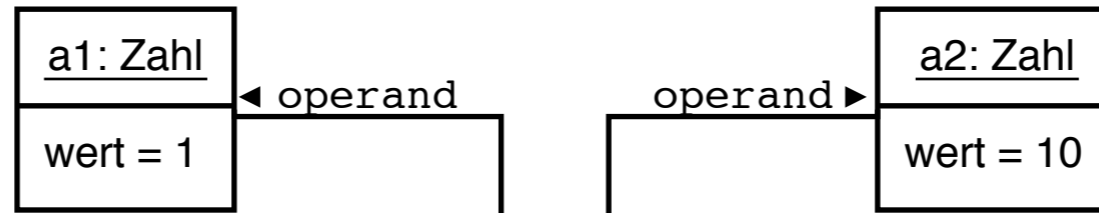
Zu einem Inhalt gibt es mehrere Formeln (die die Inhalte ansprechen)

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

# Objektmodell



# Beispiel für Objektbeziehungen



	A	B
I	I	II
2	10	III
3	100	

ergebnis ►

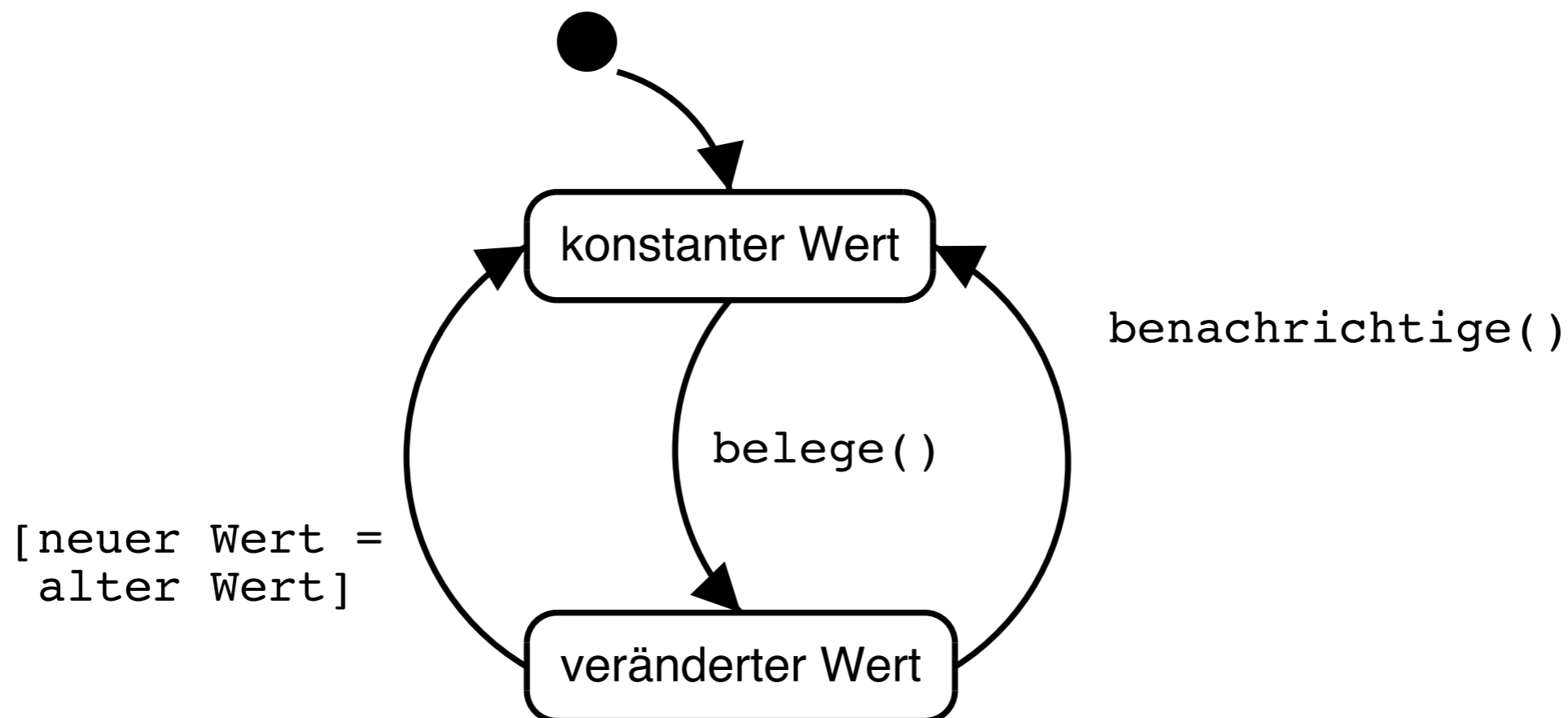
◀ operand

operand ►

ergebnis ►

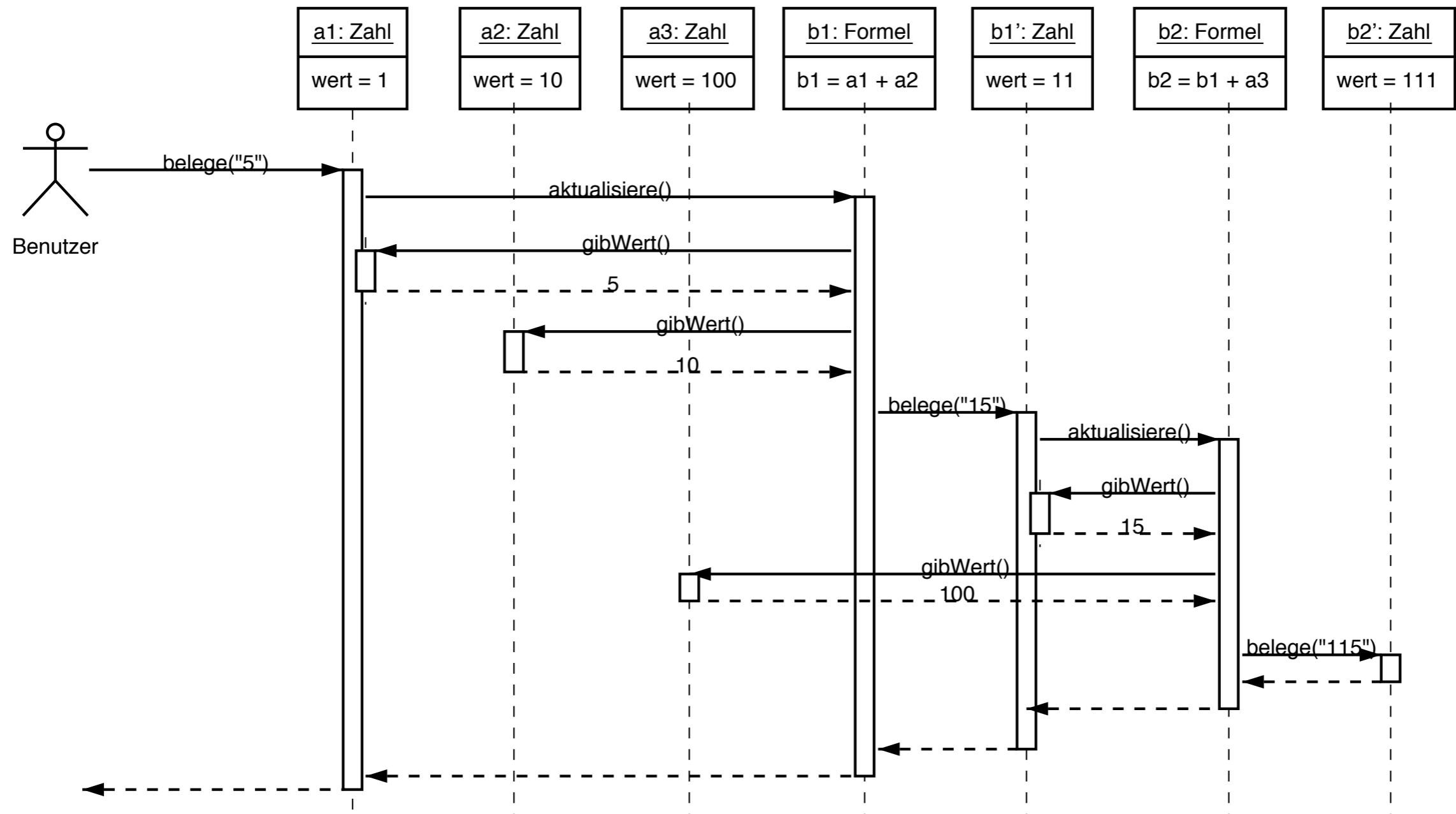
# Zustandsdiagramm

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.



# Checkliste Grobentwurf

*Objektmodell* mit

- Wesentliche Klassen und ihre Beziehungen, einschliesslich Kardinalitäten
- Die wesentlichen Methoden jeder Klasse
- Die wesentlichen Attribute jeder Klasse

*Arbeitsplan* – wer macht wann was



# Checkliste Feinentwurf

Zusätzlich sechs *Sequenzdiagramme* mit

- Start des Servers (bis zur 1. Anweisung)
- Verarbeiten von vier zentralen Zügen  
(jeweils Erfolgsfall + alternative Szenarien)
- Ausgabe des Endergebnisses (Punkttestand)
- Details auf Webseite

# Checkliste: Grobentwurf

Der Grobentwurf im Software-Praktikum 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?**
- **Sind alle Klassen, Attribute und Methoden aufgeführt?**

# 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.