PiSoftware

www.pisoftware.de

Objektorientierte Softwareentwicklung

Analyse, Design und Programmierung mit der UML und Rational Rose

Ralf Pichocki

16., völlig neu gestaltete Auflage
2003

für die Durchführung
30.06.-04.07.03
bei der SMS-Demag AG
in Düsseldorf

im Auftrage der

16. Auflage, 2003

Bahnhofstraße 190
D-45770 Marl-Sinsen

Tel +49 2365 880480
Fax +49 2365 880482
info@pisoftware.de
http://www.pisoftware.de

Bearbeitung:
Gestaltung:
Druck:

Inhaltsverzeichnis

Einführung

Der Objektorientierte Ansatz nimmt in der Analyse- und Designphase von Software-Projekten rasch an Bedeutung zu. Zur Beschreibung objektorientierter Systeme ist die mittlerweile genormte „Unified Modeling Language“ (UML) geeignet, eine grafische Notationsweise, mit der alle Phasen, nämlich Analyse, Design und Programmierung, abgedeckt werden können.

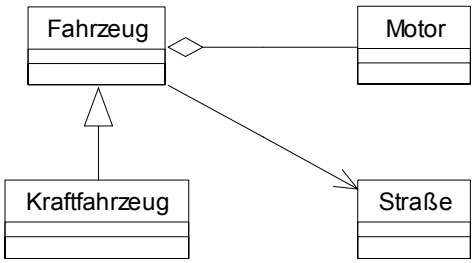
Objektorientierte Vorgehensweise

- Problembeschreibung
- Analyse:

  Objekte des Problembereiches

  sowie ihre Beziehungen untereinander als konzeptionelles Modell der Anwendung

  Beispiel: **Kraftfahrzeug** *ist* ein **Fahrzeug**, *hat* einen **Motor** und *nutzt* eine **Straße**

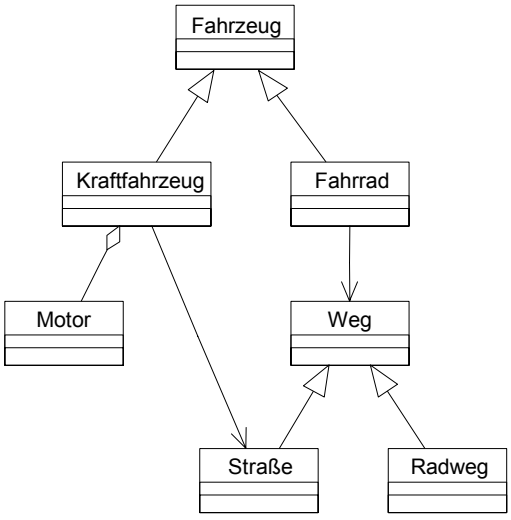


- Design:

  DV-technische Umsetzung, insbesondere mit den Zielen

  Wiederverwendbarkeit und Wartbarkeit

  Beispiel: : **Fahrrad** *ist* ein **Fahrzeug** und *nutzt* einen **Weg** (**Straße** oder **Radweg**)



- Programmierung:

  Umsetzung der Ergebnisse von Analyse und Design mit Hilfe objektorientierter oder objektbasierter Programmiersprachen, -tools und –systeme

  Quellcodebeispiel in Programmiersprache JAVA

```
public class Radweg extends Weg {
      //...
}

public class Fahrrad extends Fahrzeug {
      Weg fahrtStrecke = null;
      //...
}
```

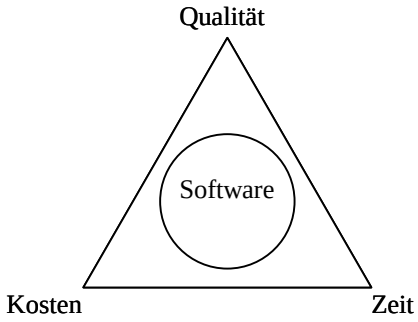
- Verschiedene Methoden: Peter Coad/Edward Yourdon, Grady Booch und andere

Überblick

- Objektorientierte Vorgehensweise in Analyse und Design, Methodenüberblick
- Prinzipien der objektorientierten Analyse
  - ❖ Identifikation von Objekten und Klassen
  - ❖ Identifikation ihrer Beziehungen untereinander (Vererbung, Assoziation, Aggregation)
  - ❖ Finden und Festlegen dynamischer und kommunikativer Objektbeziehungen
  - ❖ Beschreibung von Funktionen und Attributen der Objekte und Klassen
- Prinzipien des objektorientierten Designs
  - ❖ Modellierung elementarer Systembausteine
  - ❖ Abkapselung von Systemteilen, Definition von Schnittstellen
  - ❖ Wartbarkeit und Wiederverwendbarkeit
- Beispiele und Übungen

Grundlagen der Softwareentwicklung

Software-Entwicklung im Spannungsfeld von Qualität – Kosten – Zeit



Qualitätskriterien

- Funktionserfüllung
- Zuverlässigkeit 😐
- Robustheit ☺
- Erweiterbarkeit ☺
- Wiederverwendbarkeit ☺
- Kompatibilität 😐
- Portabilität 😐
- Benutzerfreundlichkeit
- Effizienz
- Wartbarkeit ☺

Probleme der Software-Entwicklung

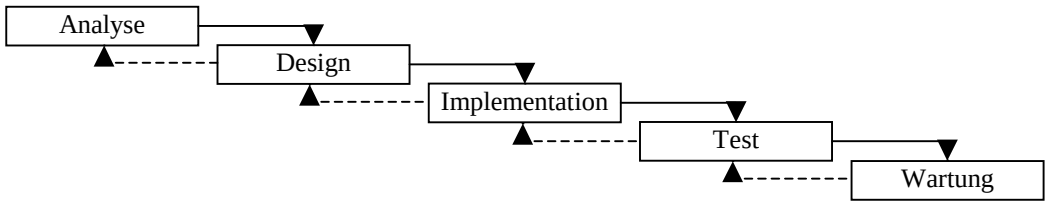
- Ende der 60er Jahre: „GOTO-Krise“, Lösung: strukturierte Programmierung
- Ende der 80er Jahre: „Wiederverwendbarkeits-Krise“, Lösung: Objektorientierung

Prinzip der Wiederverwendbarkeit

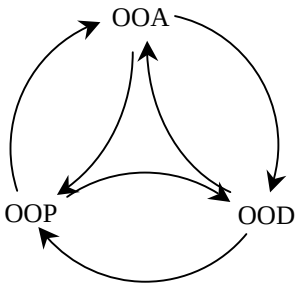
- Prozedurale Pogrammierung: Standarfunktionen und Module
- Objektorientierte Pogrammierung: Klassen und Klassenbibliotheken

Vorgehensmodelle

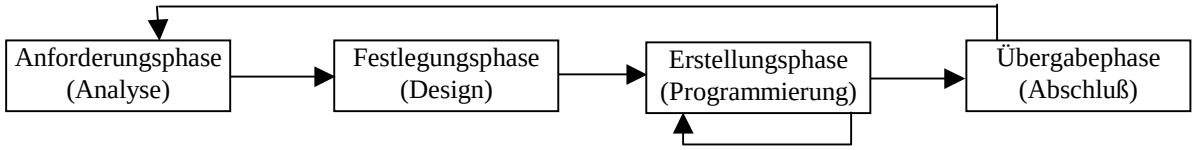
- Wasserfallmodell



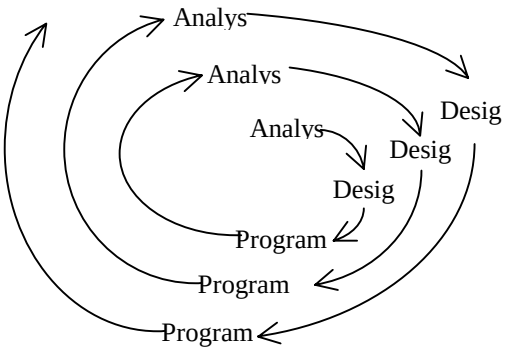
- Baseballmodell



- 4-Phasen-Modell



- Spiral-Modell



- Anwendungsfallgetriebene architekturzentrierte iterativ-inkrementelle Entwicklung

Objektorientierte Vorgehensweise

- Neue Art Probleme zu verstehen und Lösungen zu finden
- Blick auf Objekte als Einheit von Funktionen und Daten

  Beispiel **Auto.heizen()** und **Haus.heizen()** sind verschieden
- Softwaresysteme als Menge miteinander kommunizierender Objekte,

  wobei auch Klassen Objekte sein können
- Betonung auf Datenabstraktion, Informationskapselung und Wiederverwendbarkeit
- Neue Methoden für Analyse, Design und Programmierung

Eigenschaften und Ziele

Eigenschaften

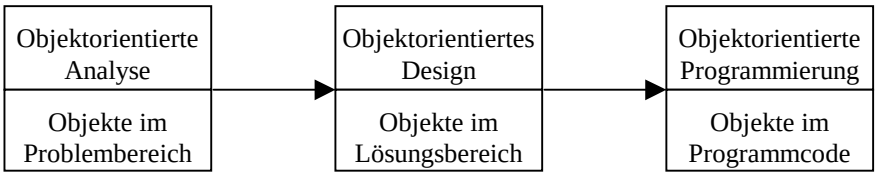
- Bessere Abbildung der realen Welt
- Wiederverwendbarkeit bereits entwickelter Module
- Einfache Modifikation und Wartung (Kapselung, Schnittstellen)
- Schnell verfügbare Prototypen, Entwicklung inkrementell und iterativ
- Besonders geeignet für offene Client/Server-Lösungen

Ziele

- Reduktion des Aufwandes bei Entwicklung und Wartung
- Wiederverwendbare Software-Bausteine und -Bibliotheken

Objektorientierter Lebenszyklus

- Analyse: Objekte im Problembereich
- Design: Objekte im Lösungsbereich
- Programmierung: Objekte im Programmcode



Beschreibung aller Phasen mit Hilfe der UML ist möglich

Konzepte der objektorientierten Entwicklung

Objekte

Ein Objekt ist die Repräsentation einer Einheit aus der realen oder der konzeptionellen Welt. Es repräsentiert also ein Objekt aus dem Problembereich. Aber auch Listen, Dialogboxen oder Prozesse können Objekte sein. Statt Objekt wird häufig auch – nicht ganz korrekt - von Instanz gesprochen. Dies ist weitgehend identisch mit dem Begriff Objekt. Mit besonderer Sorgfalt sollte man bei der Namensgebung vorgehen, um später möglichst anhand des Objektnamens den Zweck und die Funktion wiederzuerkennen.
Ein Objekt ist ein Konzept, eine Abstraktion oder eine Sache mit wohldefinierten Grenzen und Bedeutungen für eine Anwendung. Jedes Objekt im System hat vier Charakteristika:

- **Funktionen**

  Dies sind die Dienste, die ein Objekt anbietet, und werden auch als Methoden bezeichnet. In C++ heißen sie Member Functions.

  Beispiel: Die Klasse `Kraftfahrzeug` hat eine Methode `beschleunigen()`.

- **Eigenschaften**

  Dies sind die zustandsbeschreibenden Merkmale eines Objektes und werden auch als Attribute bezeichnet. In C++ heißen sie Member Variables.

  Beispiel: Die Klasse `Kraftfahrzeug` hat ein Attribut `geschwindigkeit`.

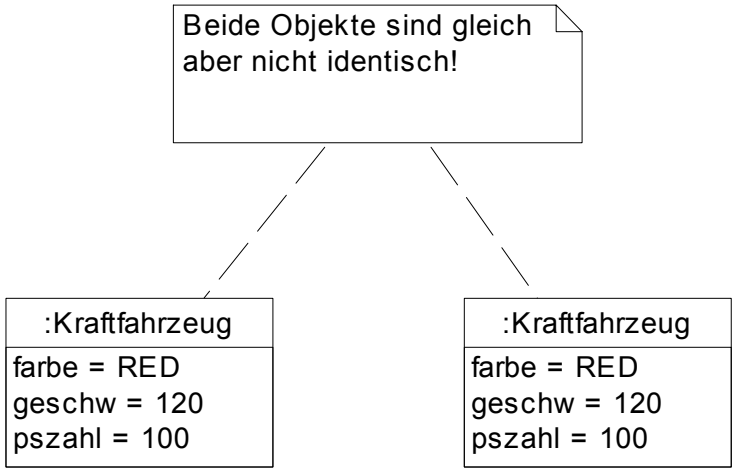
- **Eigenschaftswerte**

  Dies sind die konkreten Ausprägungen der Eigenschaften und heißen Attributwerte.

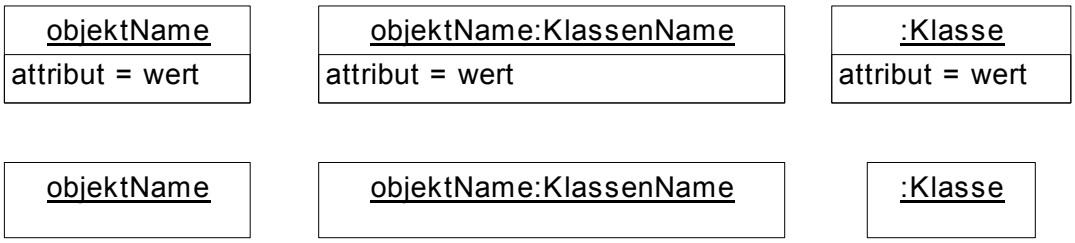
  Beispiel: Das Attribut `geschwindigkeit` hat den Wert `120`.

- **Identität**

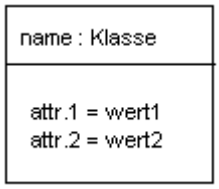
  Dies ist der – umgangssprachlich oft nicht so genau genommene Unterschied zwischen der sogenannten Gattungsidentität und der Individualidentität oder zwischen „das Gleiche“ und „das Selbe“. Zwei Objekte können zwar gleich sein im Sinne identischer Attributwerte, sind aber trotzdem niemals identisch. Jedes Objekt besitzt eine eindeutige Identität, über die es auch identifiziert werden kann.



Objekte in ihrer UML - Darstellung können benannt sein, zusätzlich einer Klasse angehören, können anonym sein (müssen dann aber einer Klasse angehören), und sie können mit oder ohne ihre Attributwerte dargestellt werden:



Allgemeine Darstellung eines Objektes in der UML:



Exkurs: Prüfung auf Gleichheit Identitätsprüfung in JAVA und C++:

„==“ : prüft – für einfache Datentypen* (int, float etc, keine Klassen!) - den Inhalt von Variablen

```
int x; int y;
x = 3; y = 3;
if ( x == y )
     ;//ist true
```

„==“ : prüft in JAVA für Klassen die Objektidentität:
Klasse String (Zeichenkette):

```
String str1 = „Text“;
String str2 = “Text”;
if ( str1 == str 2 )
     ;//ist false, weil Objekte verschieden
```

In JAVA ist für den Vergleich von Objekten die Methode equals() zu verwenden:

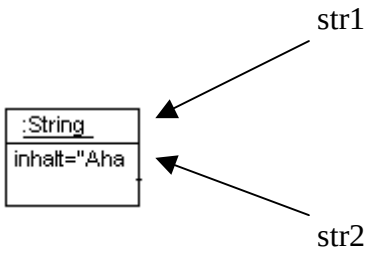
```
String str1 = „Text“;
String str2 = “Text”;
if ( str1.equals( str 2 ) )
     ;//ist true, weil Objekte gleich
```

„==“ : prüft in C++ auch für Klassen die Gleichheit:
Klasse std::string (Zeichenkette):

```
Std::string str1 = „Text“;
Std::string str2 = “Text”;
if ( str1 == str 2 )
     ;//ist true, weil Objekte gleich
```

Allerdings wird dieser Vergleich immer „shallow“ durchgeführt, weshalb der operator==() in der Regel geeignet überschrieben werden muss.

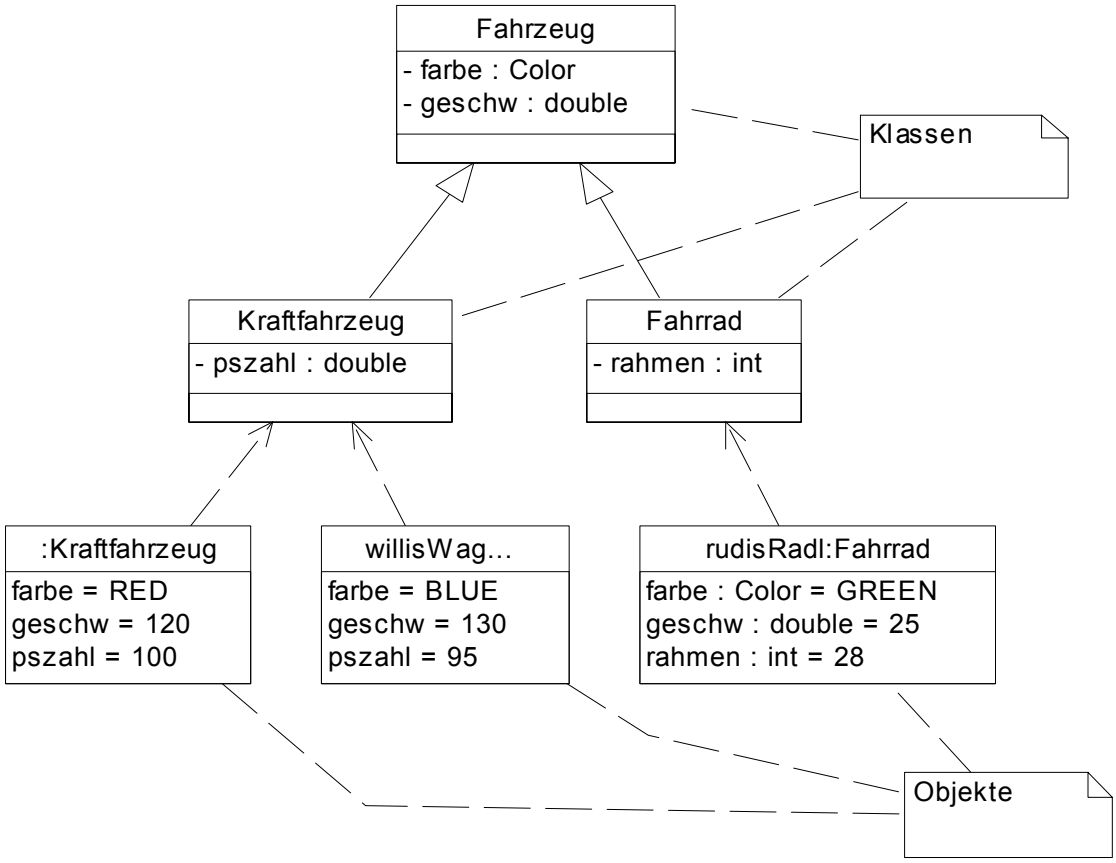
Beispiel für Identität von Objekten bei Vorhandensein verschiedener Referenzen:
`str1` und `str2` sind nur Referenzen auf das selbe Objekt



Übung 1: Was passiert in diesem Quellcode? Welche Bedingung ist wahr?

```
Bus b1 = new Bus ();
Bus b2 = new Bus ();
Bus b3 = b1;

b2=b3

if (b1==b2);
if (b1==b3);
```

Klassen



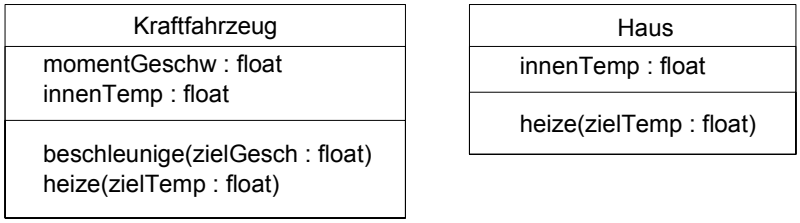
- **Gemeinsamkeiten**

  Alle Objekte einer Klasse haben die selben Attribute und bieten die selben Methoden an

- **Baupläne**

  Daher kann eine Klasse auch als Schablone für die nach ihrem Vorbild erzeugten Objekte betrachtet werden

  Jedes `Kraftfahrzeug`-Objekt hat die Attribute `momentanGeschw` und `innenTemp` und bietet die Methoden `beschleunige()` und `heize()` an.



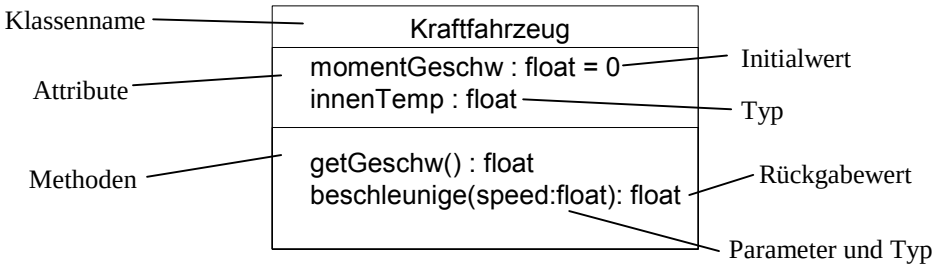
- **Datenabstraktion und Kapselung**

  Ein Objekt ist grundsätzlich nur „als Ganzes“ zu betrachten; insbesondere werden Implementierungsdetails verborgen und sind nicht zugänglich

- Klassifizierungsproblematik

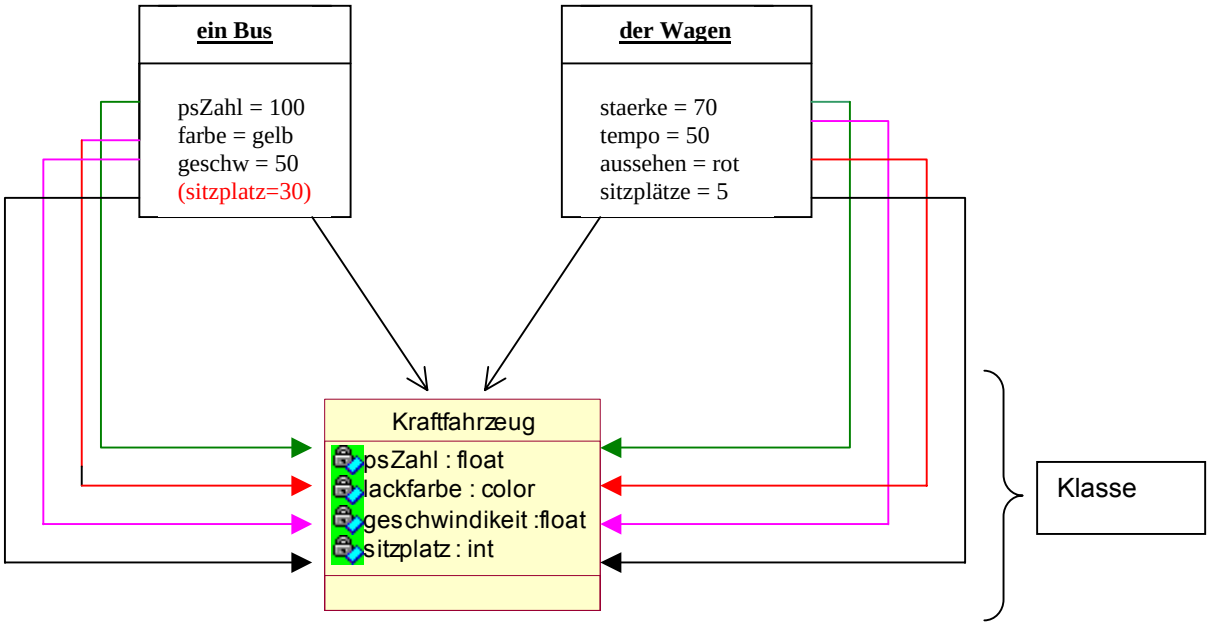
  Auch ein `Haus`-Objekt hat eine Methode `heize(),` gehört aber trotzdem zu einer anderen Klasse als die `Kraftfahrzeug`-Objekte

- Darstellung in der der UML mit Hilfe von Rational Rose



Vorgehensweise zur Klassifizierung von Objekten:

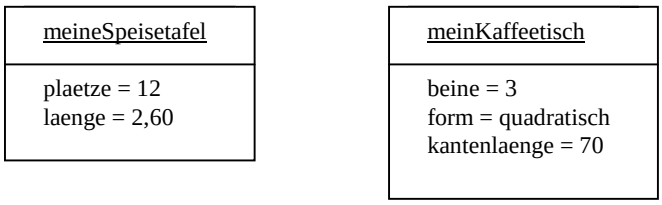
Zusammenfassungen der Gemeinsamkeiten gleichartiger oder ähnlicher Objekte



Methodik:

1. Objekte beschreiben
2. Gemeinsamkeiten finden und benennen
3. Klasse definieren
4. Attribute ggf. rückübertragen
5. Klassenzugehörigkeit der Objekte eintragen

Übung 2: Bilde eine passende Klasse



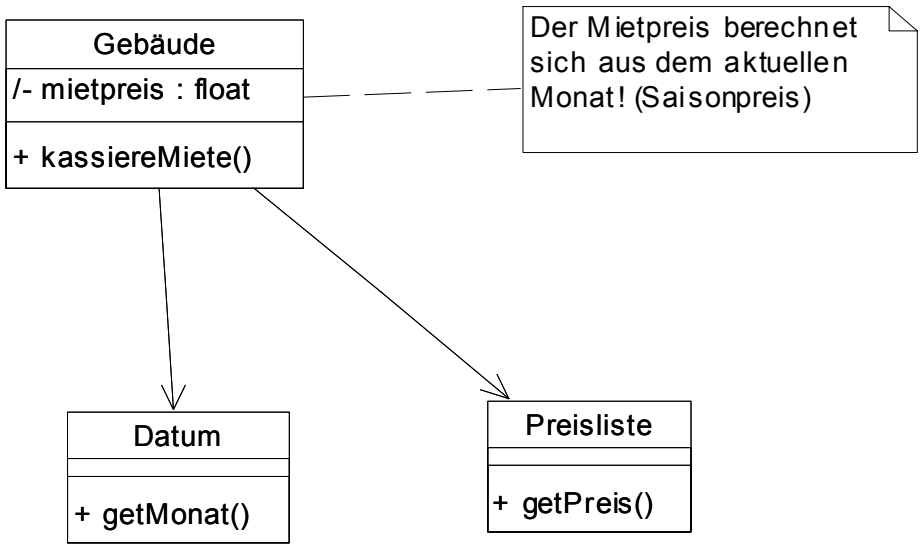
Attribute und Methoden

Tipp für das Finden von Attributen
Während der Analysephase sollte ein Fachwörterbuch erstellt werden, in dem die Bedeutung verwendeter Wörter erläutert wird.

Abgeleitete Attribute

Abgeleitete Attribute werden physikalisch nicht notwendigerweise durch einen Wert repräsentiert, sondern können bei Bedarf jederzeit berechnet werden.

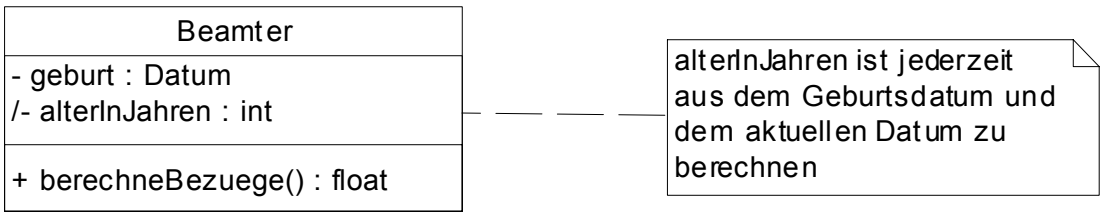
Beispiel:



Das Attribut Mietpreis kann mit Hilfe von de Klassen Datum und Preisliste berechnet werden, kann also aus anderen Objekten und Attributen abgeleitet werden.

weiteres Beispiel:

Ein Beamter erhält einen Teil seiner Bezüge abhängig von seinem Lebensalter in Jahren.



Erst im Design entscheidet sich, ob die Ergebnisse zwischengespeichert werden oder immer wieder erneut berechnet werden.

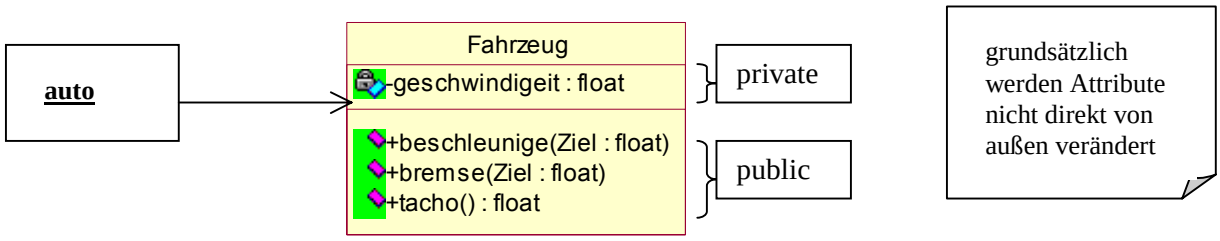
Zugriffsspezifizierer für Attribute und Methoden

Attribute und Methoden können Zugriffsspezifizierer erhalten, mit denen festgelegt wird, wie auf sie zugegriffen werden kann.

Symbole für Zugriffsmöglichkeiten:

+ public: für alle sichtbar und benutzbar
 # protected: die Klasse selbst sowie ihre abgeleiteten Klassen haben Zugriff
- private: nur die Klasse selbst hat Zugriff

Attribute sollten grundsätzlich nur durch die Klasse, in der sie definiert sind oder maximal in ihren Ableitungen, direkt verwendet werden dürfen. Andere Klassen sollten stets nur über Operationen auf sie zugreifen können.



Es wäre gefährlich, wenn direkt auf die Geschwindigkeit des Objektes „auto“ zugegriffen würde. So könnte beispielsweise Code der Art

```
auto.geschwindigkeit = 50;
```

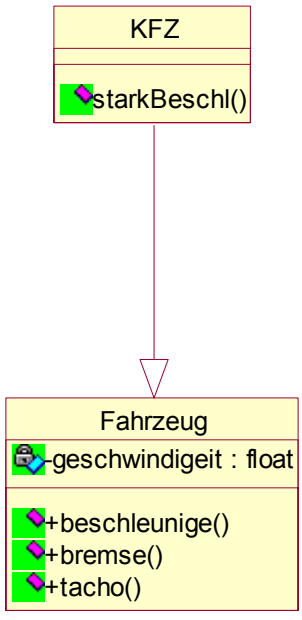
möglich sein, und das Auto würde die Geschwindigkeit 50 annehmen, ohne zunächst zu beschleunigen. Dies ist natürlich höchst unrealistisch .
Richtig ist der Zugriff über eine Methode:

```
auto.beschleunige(50).
```

Dies erreicht man durch Einschränkung des Zugriffs auf das Attribut:
Spezifiziere das Attribut als `private` und verhindere Direktzugriffe,
spezifiziere die Methode als `public`, und lasse diesen Zugriff zu.

Beispiel für direkten und indirekten Zugriff auf Attribute:

```
class Kraftfahrzeug {

     private float geschw;

     public float tacho() { return geschw; }
     public void beschl( float zielGeschw )
     {
          geschw = zielGeschw;
     }
}

//...

Kraftfahrzeug f = new Kraftfahrzeug()

f.geschw = 10; //falsch, da private,
//Zugriff nicht möglich

f.beschl(50);// korrekt, Zugriff erlaubt


class Sportwagen extends Kraftfahrzeug {

     void starkbeschl (ziel:float)
     {
          geschw = 2*Ziel;
          //falsch, da private
          //nicht protected, dann ok
     }
```

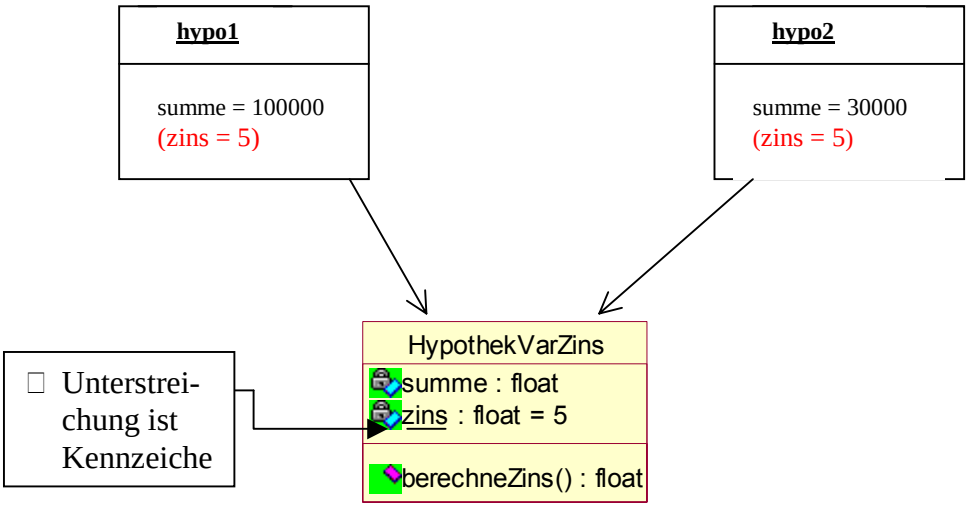


Klassenattribute

Klassenattribute gehören nicht einem einzelnen Objekt, sondern sind Attribut einer ganzen Klasse, das heißt, alle Objekte einer Klasse können auf ein solches gemeinsames Attribut zugreifen.

Mögliche Verwendung: Zählen oder Nummerieren der erzeugten Objekte

Andere Anwendung ist ein Attribut, das immer für alle Objekte einer Klasse gleich sein muss:



Das Attribut Zins mit seinem Wert (hier 5) gilt für alle Objekte.

Beispiel, Zugriff in Java:

```
HypoVarZins hypo1 = new hypoVarZins();
hypo1.summe = 200.000; // Objektname.Objekt
HypothekVarZins.zins = 5; // Klassenname.Klassenattribut

//auch möglich:
hypo1.zins; // Objektname.Klassenattribut;
     //dies sollte möglichst vermieden werden, da
     //sonst eventuell Mißverständlichkeiten auftreten
```

Beispiel für scheinbare Uneindeutigkeit:

```
HypoVarZins h1= new hypoVarZins();
HypoVarZins h2 = new HypoVarZins();
h1.zins =3;
h2 .zins = 4;
```

```
System.out.println( h2.zins); // der Zins beträgt 4
System.out.println(h1.zins); // der Zins beträgt immer noch
    //4, da die Klasse nur einen Zins für alle Objekte hat
```

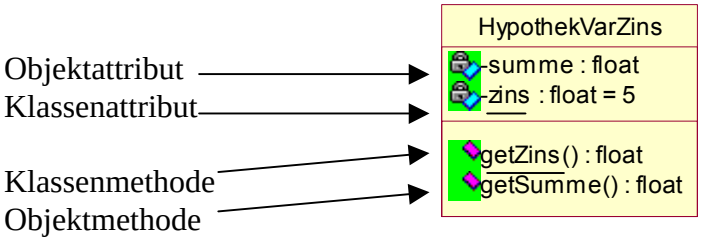
Anders verhält es sich jedoch mit dem Objektattribut Summe:

```
HypoVarZins.summe = 50.000 // falsch, hier handelt es sich
    //um ein Objektattribut; es wäre also nicht klar,
    //welches Objekt verändert wird
```

Klassenmethoden

Neben Klassenattributen gibt es auch Klassenmethoden.
Der Zugriff erfolgt nach demselben Prinzip wie bei Klassenattributen.

Objektmethoden können auf Objektattribute und Klassenattribute zugreifen!
Klassenmethoden können nur auf Klassenattribute zugreifen!

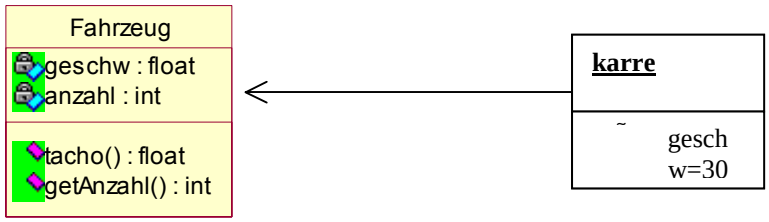



```
h1.getZins(); // ok, aber verwirrend
h1.get Summe; // ok
HypoVarZins.getZins(); // o.k.
HypoVarZins.getSumme(); // Fehler
```

weitere Beispiele:

```
HypoVarZins h1 = new HypoVarZins();
h1.summe = 100.000;
h1.getSumme ();     // liefert 100.000
h1.getZins();       // liefert 3

HypoVarZins h2 = new HypoVarZins();
h2.summe = 30.000;
h2.getSumme(); //liefert 30.000
h2.getZins();       // liefert3
```

```
h2.zins = 4
h1.getZins();       // liefert 4
```




```
Fahrzeug.tacho()          // Unsinn, weil keine
Klassenmethode
karre.tacho()       // Aufruf „tacho“ für das Objekt

Fahrzeug karre = new Fahrzeug();
karre .getAnzahl();      // liefert 1
Fahrzeug k2 = new Fahrzeug();
karre.getAnzahl();  // liefert 2
k2.getAnzahl();          // liefert 2
```

Ergebnis: Klassenmethoden gibt es nur einmal pro Klasse. Sie gelten nicht für einzelne Objekte, sondern für alle Objekte bzw. für die ganze Klasse.

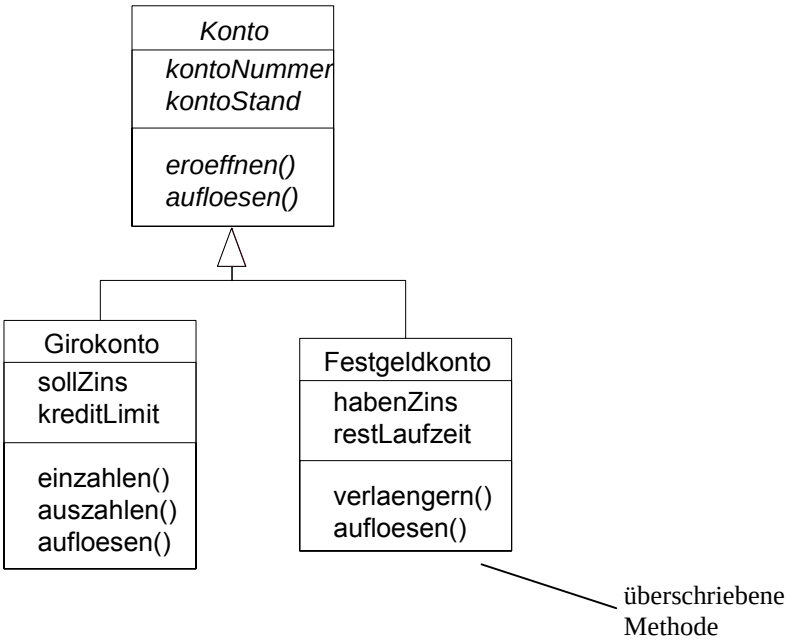
Vererbung

- **Spezialisierung:**

  von einer allgemeinen Klasse werden spezielle Klassen abgeleitet, Beispiele:

  Konto wird erweitert zu Girokonto und Sparkonto,

  Kfz wird erweitert zu Pkw und Lkw.



- **Erweiterung der Funktionalität**

  Abgeleitete Klassen haben meistens erweiterte Funktionalität und zusätzliche Eigenschaften, Beispiele:

  Girokonto hat zusätzlich einen `sollZins`, Festgeld bietet Methode `verlängern()`

- **Generalisierung:**

  Die Umkehrung der Spezialisierung fasst Gemeinsamkeiten vorhandener Klassen zusammen und bildet dann eine gemeinsame Basisklasse

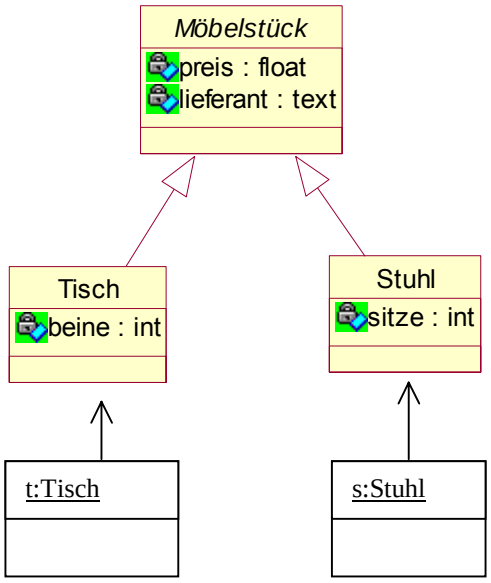
  Bsp.: Haus und Halle haben gemeinsamen Vorfahren Gebäude

- **abstrakte Klassen**

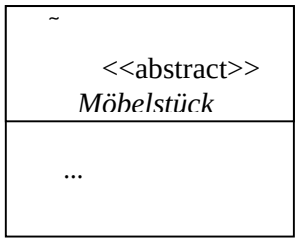
Abstrakte Klassen:

Eine Klasse, die lediglich dazu dient, Gemeinsamkeiten zusammenzufassen, von der es also keine konkreten Objekte geben kann, nennt man **→ "abstract“**

Kennzeichen: Name kursiv, <<abstract>>



oder

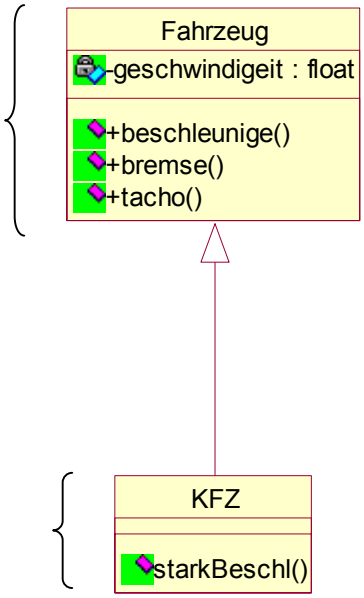


Tisch t = new Tisch();
Stuhl s = new Stuhl(); // möglich

Möbelstück m = new Möbelstück();
// nicht möglich, da abstract

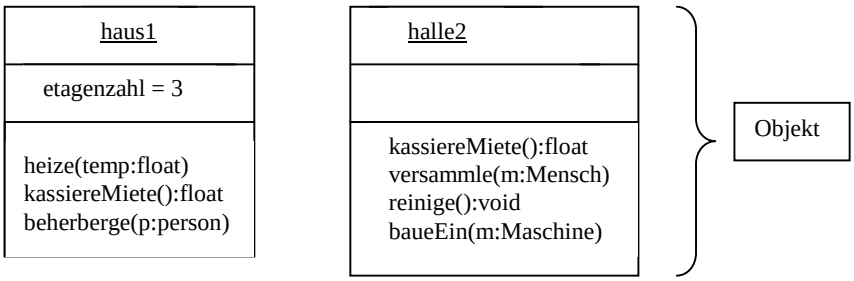
❖ **Benennungen von voneinander abgeleiteten Klassen**

Basisklasse
Elternklasse
Oberklasse
Superklasse
Generalisierung



abgeleitete Klasse
Kindklasse
Unterklasse
subklasse
Spezialisierung

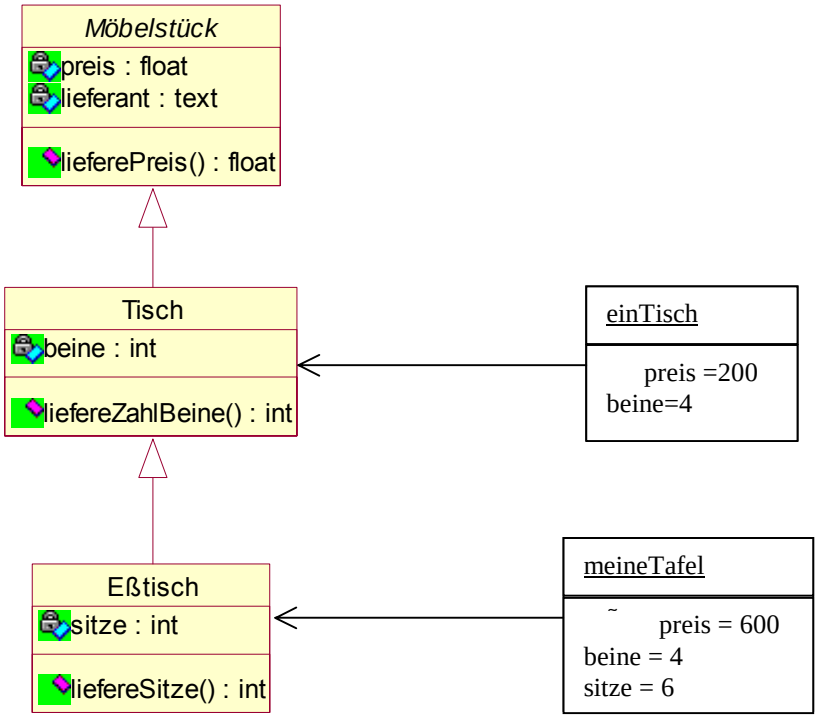
Übung 3: Bilde eine oder mehrere geeignete Klassen



Problem: Alle Attribute und Methoden müssen in der Klasse auftauchen und umgekehrt; es existiert hier ein 1:1-Verhältnis

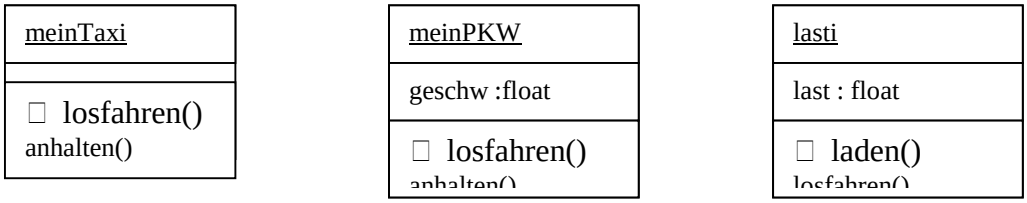
Vererben Methoden und Attributen

Eine abgeleitete Klasse **erbt alle Attribute und alle Methoden** einer Basisklasse.
Beispiel, mögliche Methodenaufrufe und Attribute:



Überschreiben von Methoden

Übung 4: Klassifizieren Sie die 3 Objekte!



Randbedingung:
Methode „losfahren()“: PKW ➔ „tritt auf´s Gas“
Taxi ➔ „Taxameter ein (Anzahl Fahrgeld), dann tritt auf´s Gas“

Beispiel für JAVA:

Java:
für Fahrzeug:

```
void losfahren()
{
  tritt auf´s Gas();
}
```

für LKW

```
void losfahren()
{
  taxameterEin()
  super.losfahren()  // gehe zur Superklasse, d.h., suche die
                     // geerbte Methode, gehe eine Ebene höher
}
```

Übung:

```
1. einFahrzeug = new Fahrzeug();
   einFahrzeug.losfahren(); //eigentlich (ursprüngliche) Methode
```

→ das Objekt einFahrzeug wird erzeugt für die Klasse Fahrzeug. Es hat alle Attribute und Methoden dieser Klasse, Fahrzeug kann losfahren

```
   einTaxi = new Taxi ();
```

```
   einTaxi.losfahren;        // die überschriebene Methode wird
                             // verwendet
```

→ fährt nach neuer Methode los

```
2. einPKW = new PKW();
   einPKW.laden();           // falsch, laden ist nur eine Methode
                             // der Klasse LKW, von Geschwistern
                             // kann nicht geerbt werden
   einPKW.losfahren;         // verwendet wird die geerbte Methode

3. LKW l = new LKW();
   l.losfahren();            // geerbte Methode
   l.laden();                // eigene Methode

4. KFZ q = new LKW();
   q.losfahren();            // eigene Methode von KFZ
   q.laden ();               // Fehler, weil KFZ.laden nicht
                             // existiert

5. aber:
   PKW s = new Taxi();
   s.losfahren();            // die Referenz s kann Taxi oder PKW
                             // enthalten, das System weiß aber,
                             // von welcher Klasse das Objekt ist,
                             // somit wird die Methode losfahren()
                             // von der Klasse Taxi verwendet.
   KFZ s = new Taxi();
   s.losfahren();            // überschreibt Methode (Polymorphie)
   s = new LKW();
   s.laden();                // Fehler, denn s kennt nur Methoden
                             // und Attribute der Klasse KFZ,
                             // laden() nicht vorhanden
```

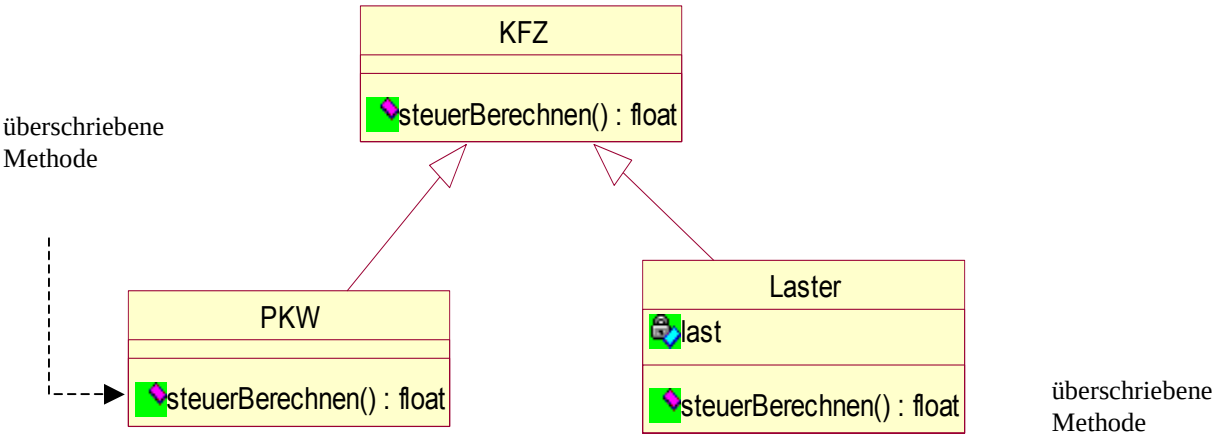
Weiteres Beispiel:

`steuerBerechnen()` für PKW und Laster unterschiedlich

```
KFZ k = k.steuerBerechnen(); // falsch

KFZ k = new LasterW();
k. steuerBerechnen();    // nur richtig, da bei KFZ auch
steuerBerechnen()

{
  for each KFZ k aus Liste aller KFZ do
```

```
     { k.steuerBerechnen()...;
     }                    // es erfolgt für jedes KFZ eine
separate Steuerbe-
                          rechnung, bei Änderung der
Steuersätze braucht
                          gesamtes Programm nicht geändert zu
werden
```

wenn Methode „*steuerBerechnen()*“ nicht bei KFZ implementiert, erfolgt keine Berechnung, denn für den Compiler ist k ein Objekt der Klasse KFZ, dort wäre keine Methode „*steuerBerechnen()*“. (siehe Thema Polymorphie)



Mehrfachvererbung

Mehrfachvererbung: erbe von mehr als einer Basisklasse

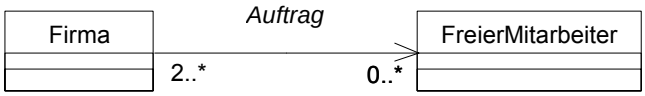
Übung 5:

1. Entwirf Klassen für die Objekte „einKaufvertrag“ und „einKreditvertrag“!
2. Generalisiere diese zu einer gemeinsamen Basisklasse!
3. Füge eine weitere Klasse „Ratenkauf“ ein! Wie passt diese in das Konzept?

Tipp: Beginn sofort mit der Klasse oder entwirf zunächst die typischen Objekte!

Assoziation

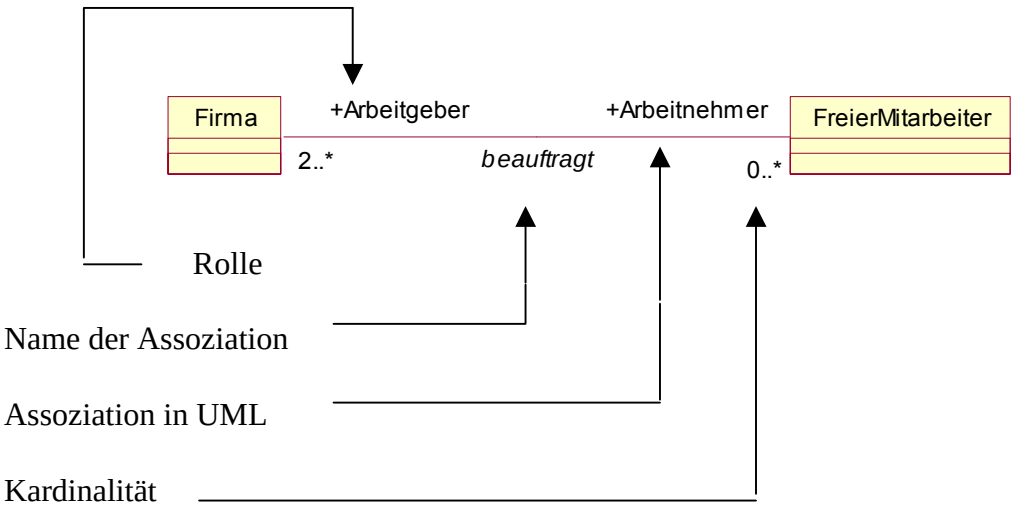
- „Nutzt“-Eigenschaft
- Kardinalität
- Rolle

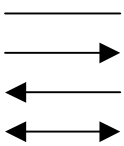


Assoziation

Notation

= „nutzt“-Eigenschaft
Damit ein Objekt einem anderen Objekt eine Nachricht schicken kann, muß eine Beziehung zwischen beiden vorhanden sein.

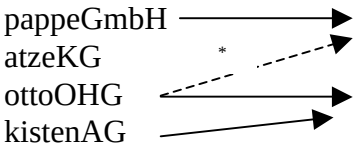




Kardinalitäten:

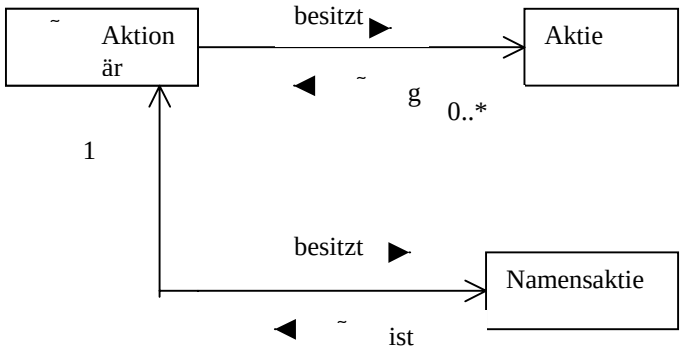
* → beliebig viele
.. → von – bis
, → oder

2..* → ein freier Mitarbeiter arbeitet für 2 bis beliebig viele Firmen
0..* → eine Firma kann keine bis beliebig viele Mitarbeiter beschäftigen



* muß, da ein freier Mitarbeiter bei mindst 2 Firmen beschäftigt sein muß

Beispiel:

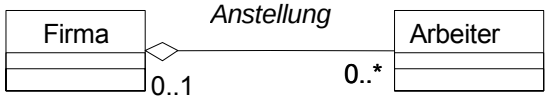


Übung 6:

Moduliere eine Assoziation für das Mitfahren eines Fahrgastes im Omnibus!
Gerichtet/ungerichtet? Rollen? Kardinalitäten?

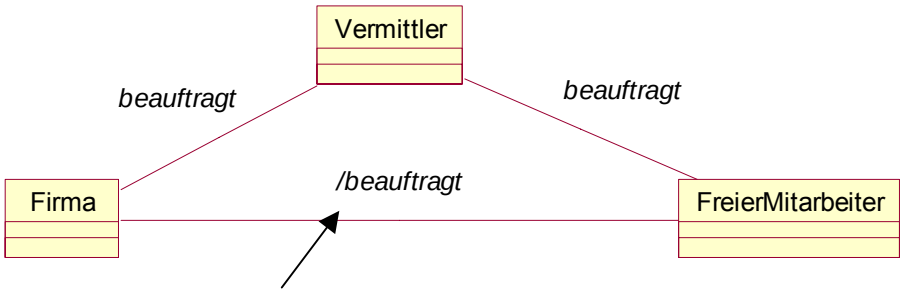
Aggregation

- „Hat“-Eigenschaft
- Kardinalität
- Kann- oder Muß-Beziehung
- Komponenten auch allein existenzfähig



Sonderformen von Assoziationen

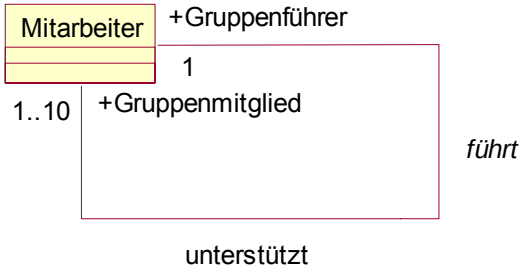
abgeleitete Assoziation



Darstellung in UML für abgeleitete Assoziationen

= Assoziation, deren konkrete Objektbeziehung jederzeit aus Werten anderer Objektbeziehungen: Assoziation oder Aggregation und ihrer Objekte
berechnet werden kann.

rekursive Assoziation

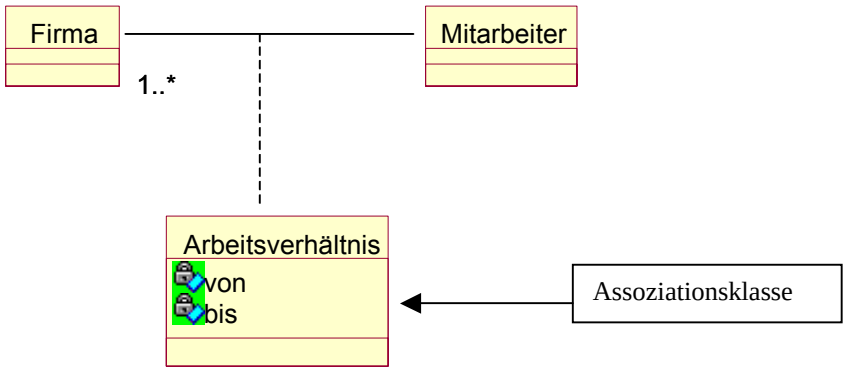


→alle sind Mitarbeiter, aber einer ist Gruppenführer und führt die Gruppenmitglieder

→eine Assoziation, die sich auf sich selbst, dem Objekt bezieht, braucht man nicht zu simulieren

Attributierte Assoziation

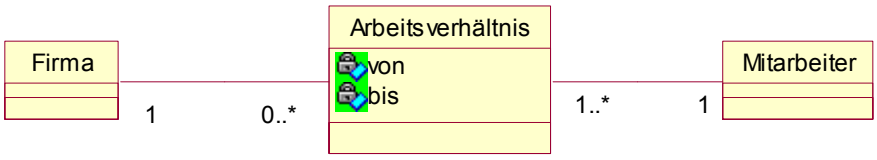
Thema:Lebenslauf



Die Eigenschaften von / bis sind Attribute dieser Verbindung

Grund: In derselben Firma kann ein Mitarbeiter auch 2x gearbeitet haben.
Würden diese Attribute Attribute von Mitarbeiter sein, hätte ein Objekt von der Klasse Mitarbeiter nur einmal Daten von dem einem Beschäftigungszeitraum in der Firma.

Attributiere Assoziationen können in zwei normale Assoziationen aufgelöst werden:



Aggregation

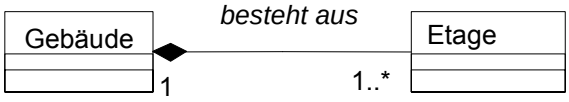
- „hat“ oder „enthält“ – Eigenschaft

- Name
- Rolle
- Kardinalität

Typischerweise umfaßt eine Aggregation eine gerichtete Assoziation vom Aggregat zum Einzelteil

Komposition

- Form der Aggregation
- „Besteht aus“ -Eigenschaft
- Kardinalität
- Muß-Beziehung
- Komponenten allein nicht existenzfähig



Komposition

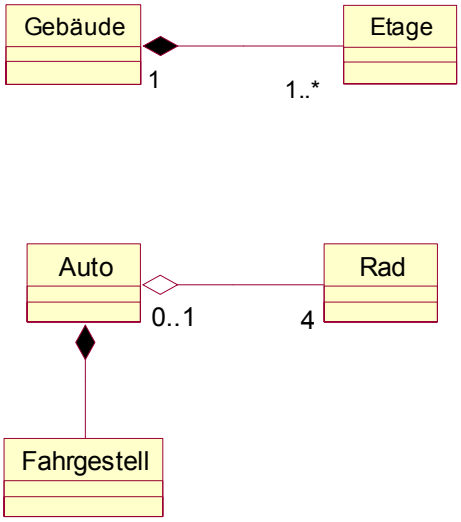
schärfere Form der Aggregation

- besteht aus
- hat als Teil

} Eigenschaft

- Name, Rolle, Kardinalität
- Komponenten sind existenzabhängig vom Ganzen, dem sogenannten „Kompositum“

Beispiele:



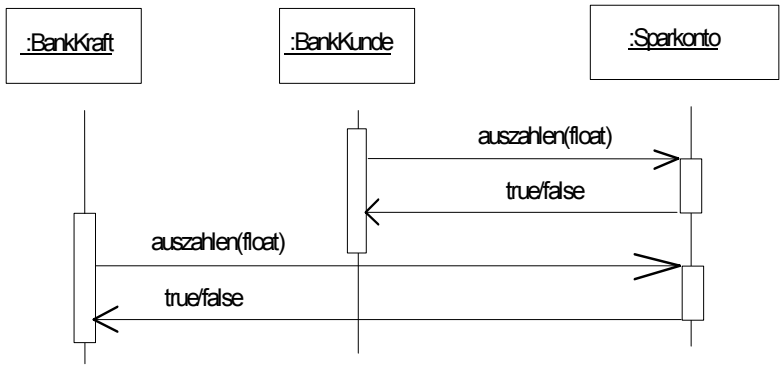
Übung Teil 7: Zulassung eines Autos

Entwirf Klassen (wenige Attribute) und ihre Beziehungen
(Vererbung, Assoziation, Aggregation) für Versicherungsnehmer, Bereich KFZ Versicherungen

Kunde, KFZ, KFZ-Schein, Bankverbindung, Vertrag, Versicherungsgesellschaft, Sonderausstattung, Straßenverkehrsamt

Nachrichten

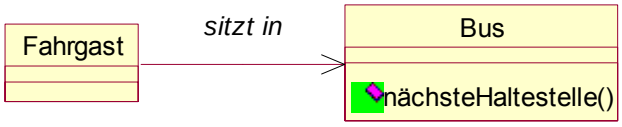
- Senden entspricht Methodenaufruf
- Kommunikationspfad (Link) ist notwendig
- Sender *kennt* und *sieht* den Empfänger
- Entspricht Einschreiben *mit Rückantwort* aber *ohne Absender*



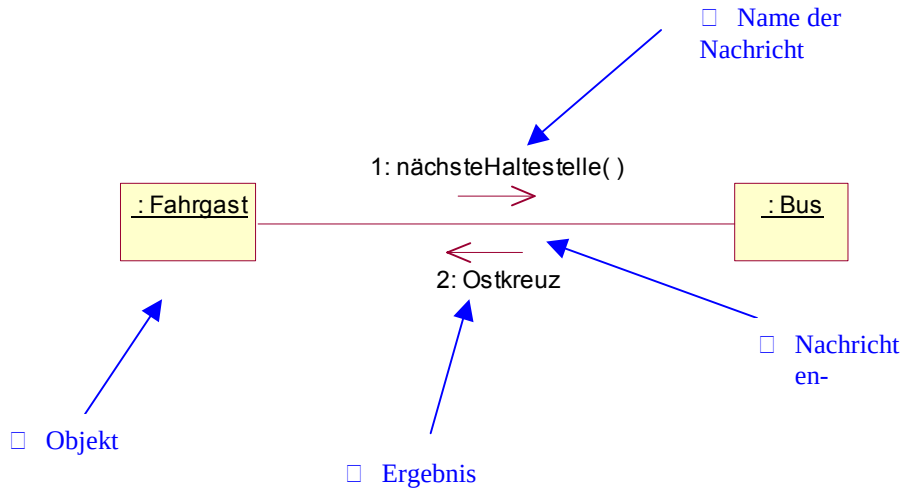
- Ein Objekt fordert ein anderes auf, eine Aufgabe zu erfüllen (fordert einen Dienst an)
- Sender ➔ Client und Empfänger ➔ Server
- Empfang einer Nachricht zieht immer die Ausführung einer Methode nach sich (wird daher in der Regel gleichgesetzt)
- notwendig: Verbindung = Kommunikationspfad vom Sender zum Empfänger
- link kann sein **Assoziation** (gerichtet) oder **Aggregation** (gerichtet)

Beispiele:

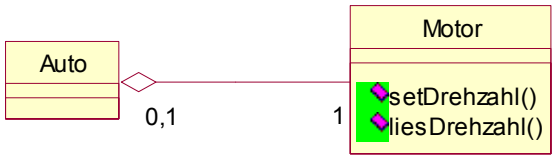
Klassendiagramm



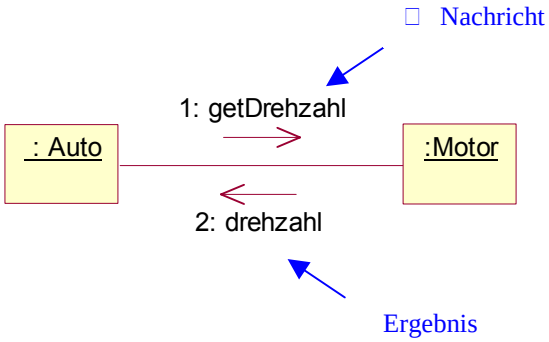
Kollaborationsdiagramm



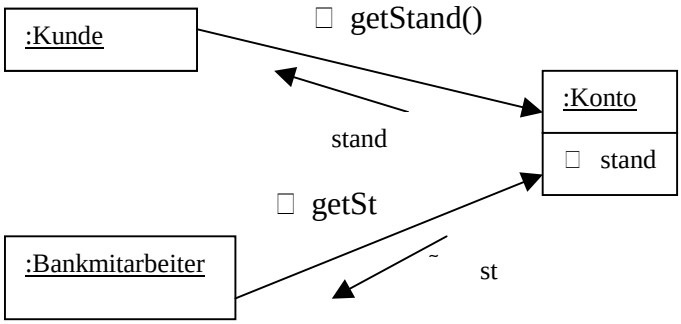
Klassendiagramm



Kollaborationsdiagramm



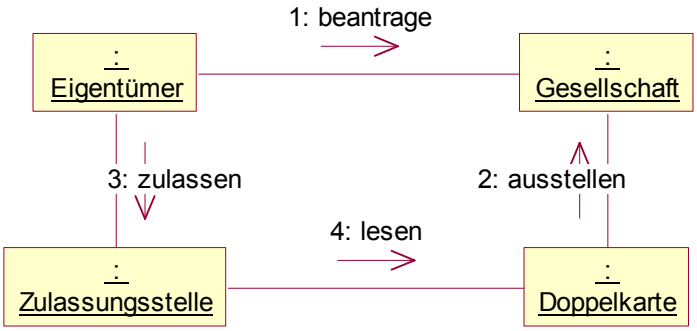
- Sender sieht Empfänger (Adresse), umgekehrt nicht notwendig
- Einschreiben ohne Absender, eventuell mit Rückantwort (Funktionswert)



Für Konto ist es unerheblich, wer nach dem Stand fragt.

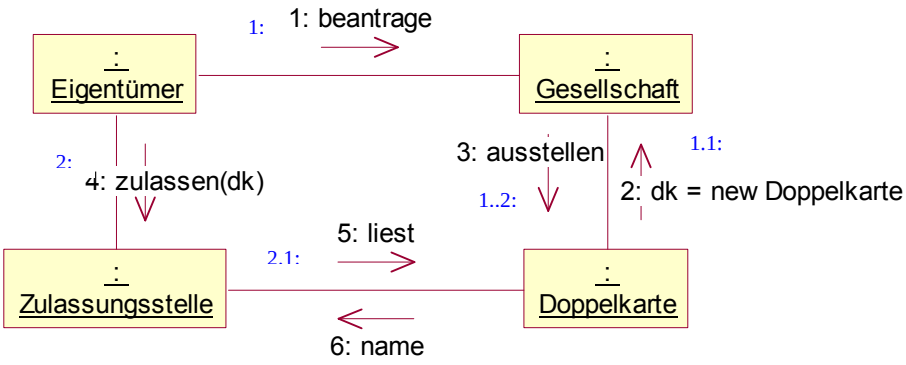
Darstellung vom Nachrichtenfluß

Kollaborationsdiagramm



- zeigt hauptsächlich, welche Klassen miteinander kooperieren und wie sie das tun

Kollaborationsdia-
gramm mit
Ergebnissen



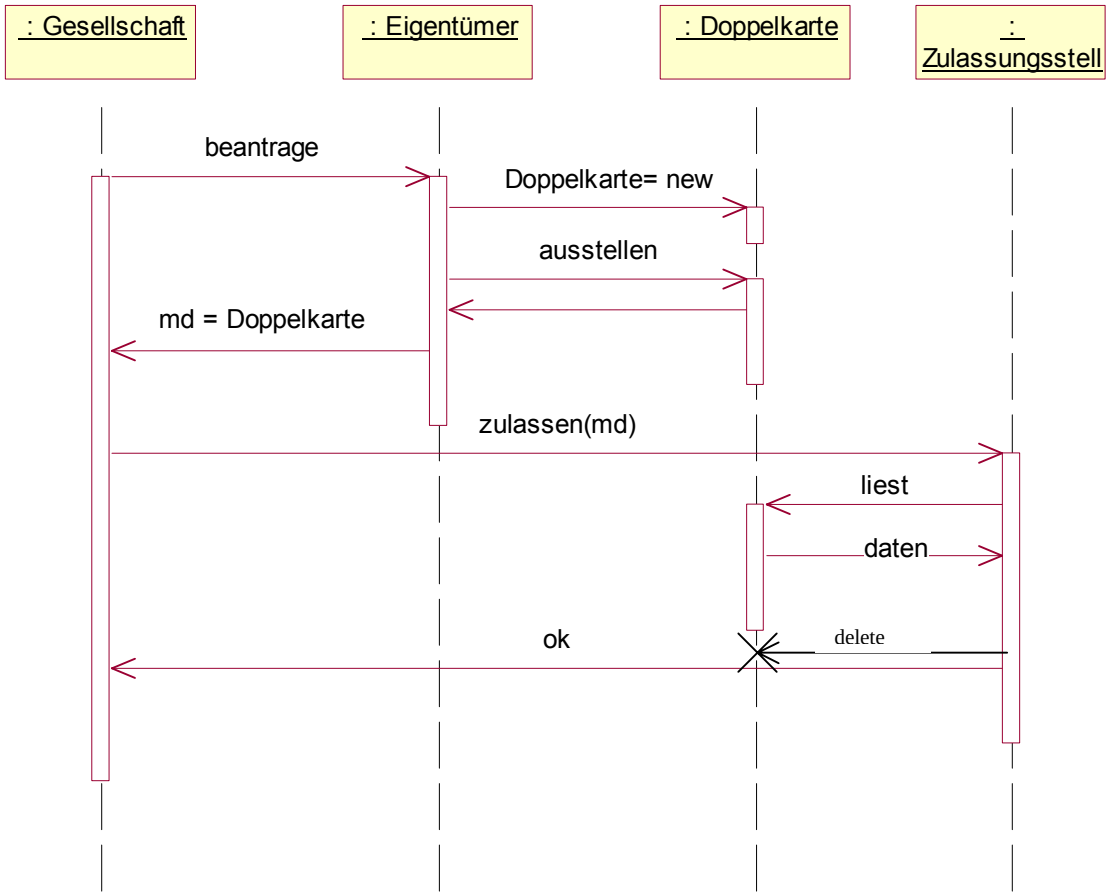
- UML-korrekte hierarchische Numerierung

Übung 8: Zulassung eines Autos

Entwirf den Nachrichtenfluß zwischen Eigentümer, Zulassungsstelle und KFZ-Schein vom Antrag der Zulassung bis zur Übergabe des KFZ-Scheins.
Tip: Es fehlt noch ein Objekt.

Sequenzdiagramm

Das Kollaborationsdiagramm ist unübersichtlich, wenn es um viele aufeinanderfolgende Nachrichtenaufrufe geht, sog. Sequenzen von Nachrichten.
Dann ist ein **Sequenzdiagramm** besser geeignet.



- Kollaborationsdiagramm geeignet für viele Objekte und wenige Nachrichten
- Sequenzdiagramm geeignet für wenige Objekte und viele Nachrichten

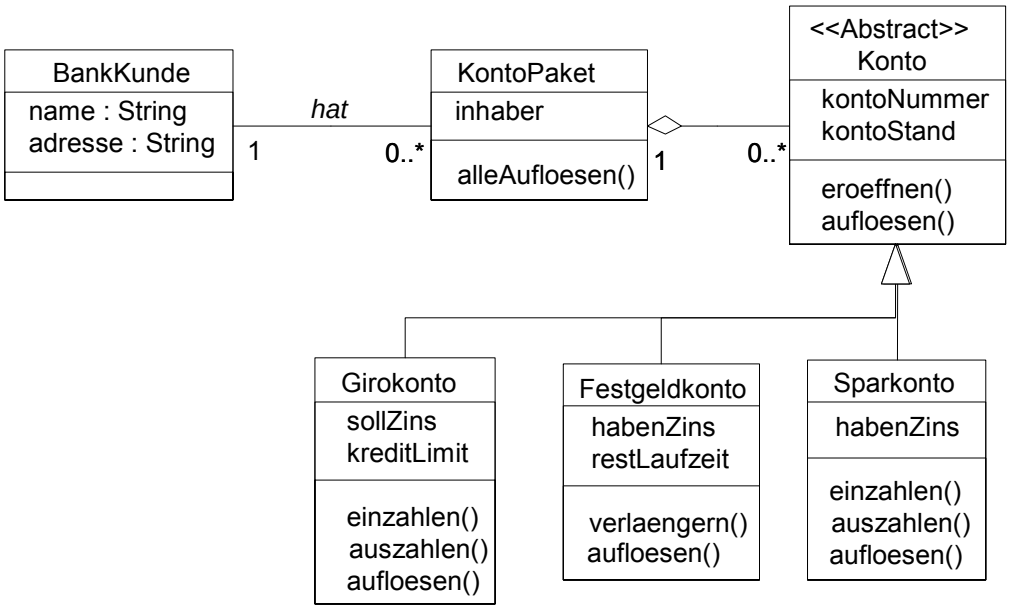
➔ Darstellung des Nachrichtenflusses im Zeitablauf

Übung 9: Zulassung

Entwirf ein Sequenzdiagramm für den Falle KFZ-Schein ausstellen und eines für den Fall KFZ abmelden!

Polymorphie

- Statische Polymorphie
- Dynamische Polymorphie
- Senden derselben Nachricht an Objekte verschiedener Klassen derselben Basisklasse



- trotzdem Aufruf der richtigen Methode

- ***Polymorphie***

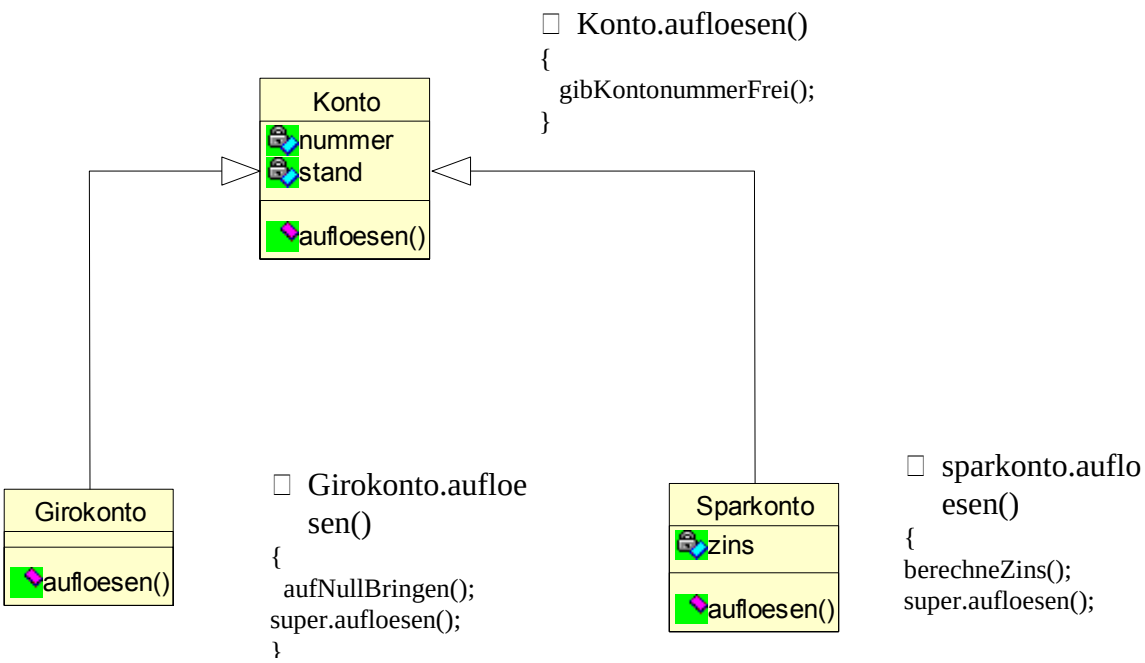
statische Polymorphie

3+4	→	7	(Ganzzahl)
3.14 + 1.414	→	4.554	(Fließkommazahl)
„3“ +“4“	→	„34“	(Text/Zeichenkette)

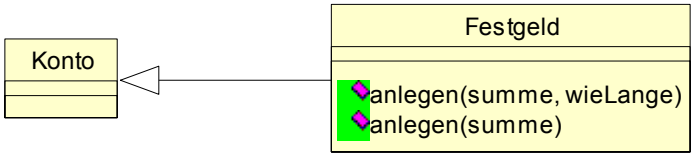
„+“ hat verschiedene Bedeutung → polymorph

Der Datentyp ist zur Kompilierzeit bekannt, der Compiler kann also jetzt schon die Funktion festlegen.
Dies entspricht der „frühen Bindung“ oder auch „statische Bindung“ genannt.
Schon während der Kompilierzeit (Übersetzung) kann die Bedeutung von „+“ richtig und eindeutig interpretiert werden.

statische Polymorphie für Objekte




```
Girokonto g = new Girokonto(); // Hier kann ich schon zur
Sparkonto s = new Sparkonto(); // Kompilierzeit feststellen,
                               // zu welcher Klasse g und s
g.aufloesen();                 // gehören
s.aufloesen();                 // und somit die jeweiligen
                               // Methoden aufrufen (stat.
                               // Polymorphie, frühe Bindung)
```



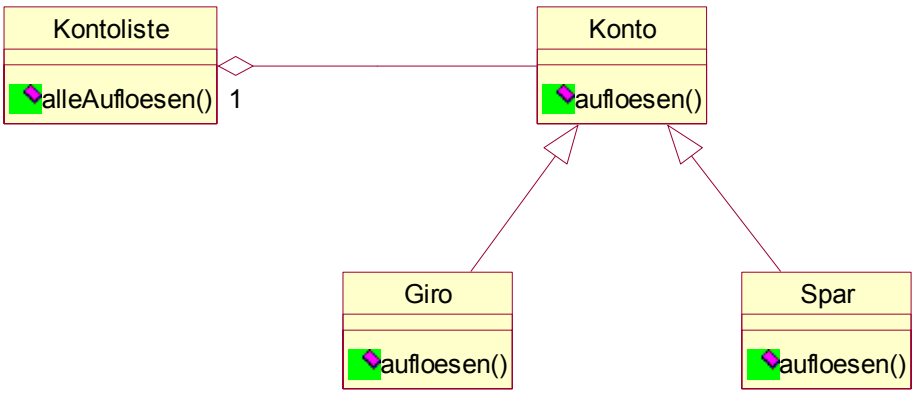
Zwei Methoden mit gleichem Namen aber **unterschiedlichen Parametern** heißen **überladen**.

```
Festgeld f = new Festgeld();
f.anlegen(10.000);       // vorgegebener Zeitraum
f.anlegen(5000, 90)      // 90 Tage lang
```

Auch hier kann schon früh die „richtige“ Methode ausgewählt werden:
statische Polymorphie.

Dynamische Polymorphie

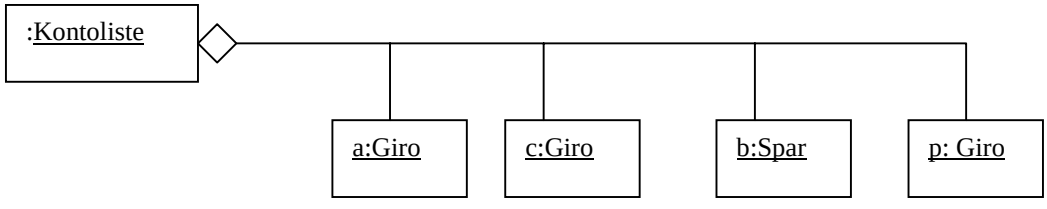
Erst zur Laufzeit des Programms (also spät), kann festgestellt werden, welche Methode benötigt wird (späte Bindung).



Ziel: Alle Konten „in einem Rutsch“ aufloesen.

```
Kontoliste.alleAufloesen()
{
  for each Konto k in Liste do k.aufloesen;
}
```

Da zur Kompilierzeit noch nicht bekannt ist, welche und wie viele Konten in welcher Reihenfolge in der Liste sind, kann auch nicht entschieden werden, welches (jeweils) die richtige Methode ist.
Diese Entscheidung muß spät (dynamiscch) fallen!
Beim Aufruf von k.aufloesen() wird nun geprüft, zu welcher Klasse k letztendlich gehört (zu Konto oder zu einer der Subklassen von Konto)
Die dazugehörige Methode wird aufgerufen!



Hier werden aufgerufen:

```
1. a.Giro. aufloesen() //weil a instance of Giro
2. c.Giro.aufloesen() //weil c instance of Giro
3. b.Spar.aufloesen() //weil b instance of Spar
4. p.Giro.aufloesen() //weil p instance of Giro
```

Java: „instance of“
→ gehört zu Klasse

- Voraussetzungen für dynamische Polymorphie

  Die Methode, die polymorph sein soll, muss in der betrachteten Basisklasse vorhanden sein (sonst kann sie gar nicht aufgerufen werden) und in der abgeleiteten Klasse überschrieben werden.

Übung 10: Konto

Ergänze die *Klassen Giro* und *Spar* um jeweils eine Methode `zinsVerrechnen()`,die die jeweiligen Soll- und Habenszinsen berechnet und dem Kontostand ab- oder zuschlägt.

Übung 11: Konto

Führe die *Klasse Giro2000* ein, bei der es zusätzlich für Guthaben über 2000 DM einen *Guthabenszins* gibt.

Übung 12: Konto

Führe eine *Kontoliste* ein, die Konten enthält und einen Dienst `jahresabschluß()` bietet. Formuliere diese Methode! Was ist zusätzlich notwendig?

Übung 13: Konto

Füge eine weitere *Klasse Festgeld* ein. Wie passt sie ins Konzept?

Bemerkung:
In C++ kann für jede einzelne Methode die dynamische Polymorphie ein- oder ausgeschaltet werden:

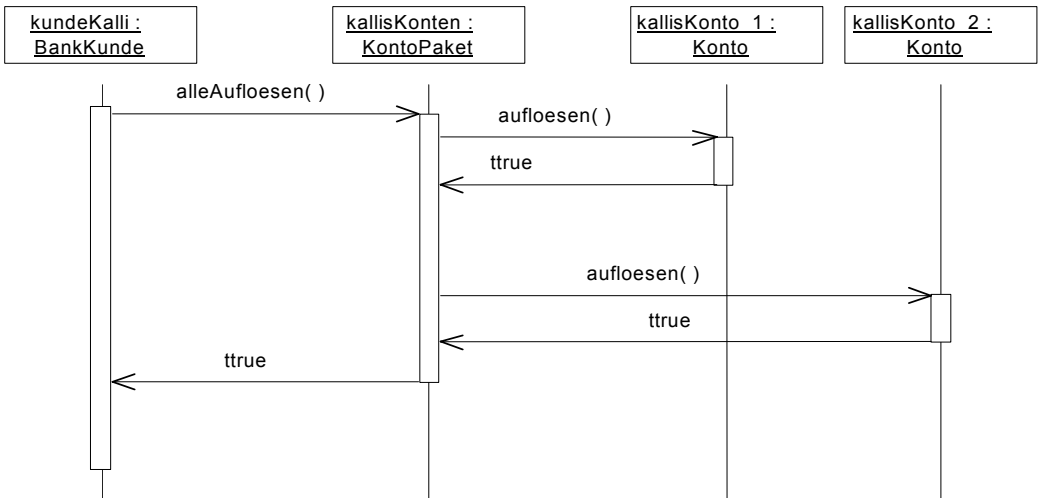
```
virtual void zinsV(); // dyn. polymorph
float getStand();     // muss nie überschrieben werden, kann also
                      // früh gebunden werden
```

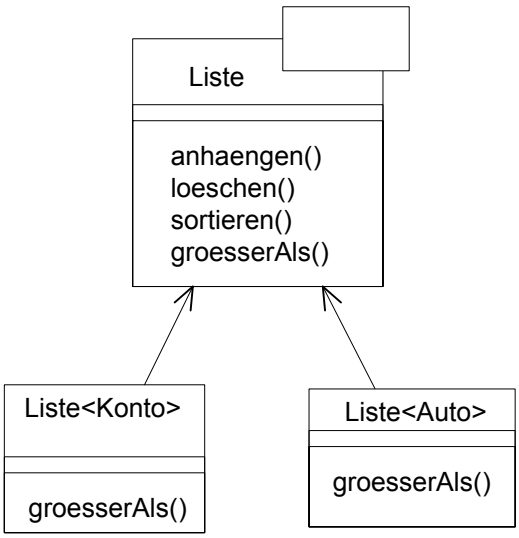
In Java sind grundsätzlich alle Methoden dynamisch polymorph.

```
void zinsV();      // auch ohne weiteren Befehl
float getStand();  // wie alle dyn. polymorph
```

Generizität

- anderer Weg zur Wiederverwendbarkeit





- hauptsächlich Collection -Klassen
- aber auch andere Klassen, die Objekte als Ganzes bearbeiten

Generische Klassen

- generische Klassen, Template-Klassen, Schablonen-Klassen
  UML ➔ parametrisierbare Klassen

Beispiel:

Entwirf eine Klasse zur Darstellung einer Warteschlange für Fahrkartenkunden!
Nutze dazu als interne Datenstruktur ein Array aus Fahrkartenkunden!

Exkurs: Arrays = Felder

```
int x;     // definieren
x = 3;     // zuweisen
print (x); // auslesen und schreiben
```

Variable = Referenz (zeigt auf ein „Kästchen“, darin ganze Zahl)

Beispiel für ein Datenfeld:

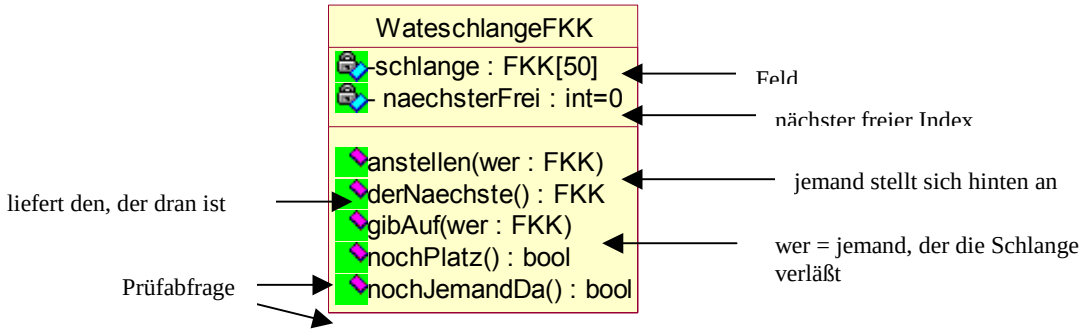
Umsätze des ganzen Jahres:

```
int umsatz jan       = int umsatz 01
int umsatz feb...    = int umsatz 02
```

Berechnung des Gesamtumsatzes = umsatz01 + umatz02
12 Variablen, die zusammengehören➔ bilden ein Datenfeld
(Referenz Umsatz zeigt darauf)

Klasse Warteschlange

1. Entwickle die Klasse Warteschlange!



Ergänzung:Weil „Array“, gibt es eine Maximalgröße, d. h. es soll eine Warteschlange mit 50 Plätzen entworfen werden.

Realisierung der Methoden

Rückgabetyp

Methodenname

```
1.  bool noch Platz( )
    {
      if ( naechstFrei <50)
       return true;
     else
       return false;
     }
2.  void anstellen(wer FKK)
    {
      if (nochPlatz() = = false) return;
    Schlange [naechstFrei] = wer;
    naechstFrei ++;                  // erhöhe um 1
    }
3.  FKK der Naechste()
    {
      if (nochJemandDa())             // liefere den zurück, der
                                      // vorn in der Schlange steht
      {
        return Schlange [0];
        for (index = 1 bis index = naechstFrei-1)
        do {
          schlange[index-1] = schlange [index];
          index ++;
          }
       naechstFrei --;                // verringere um 1
     }
    }
```

Wieviel weiß/muss wissen die Warteschlange von der Klasse FKK?

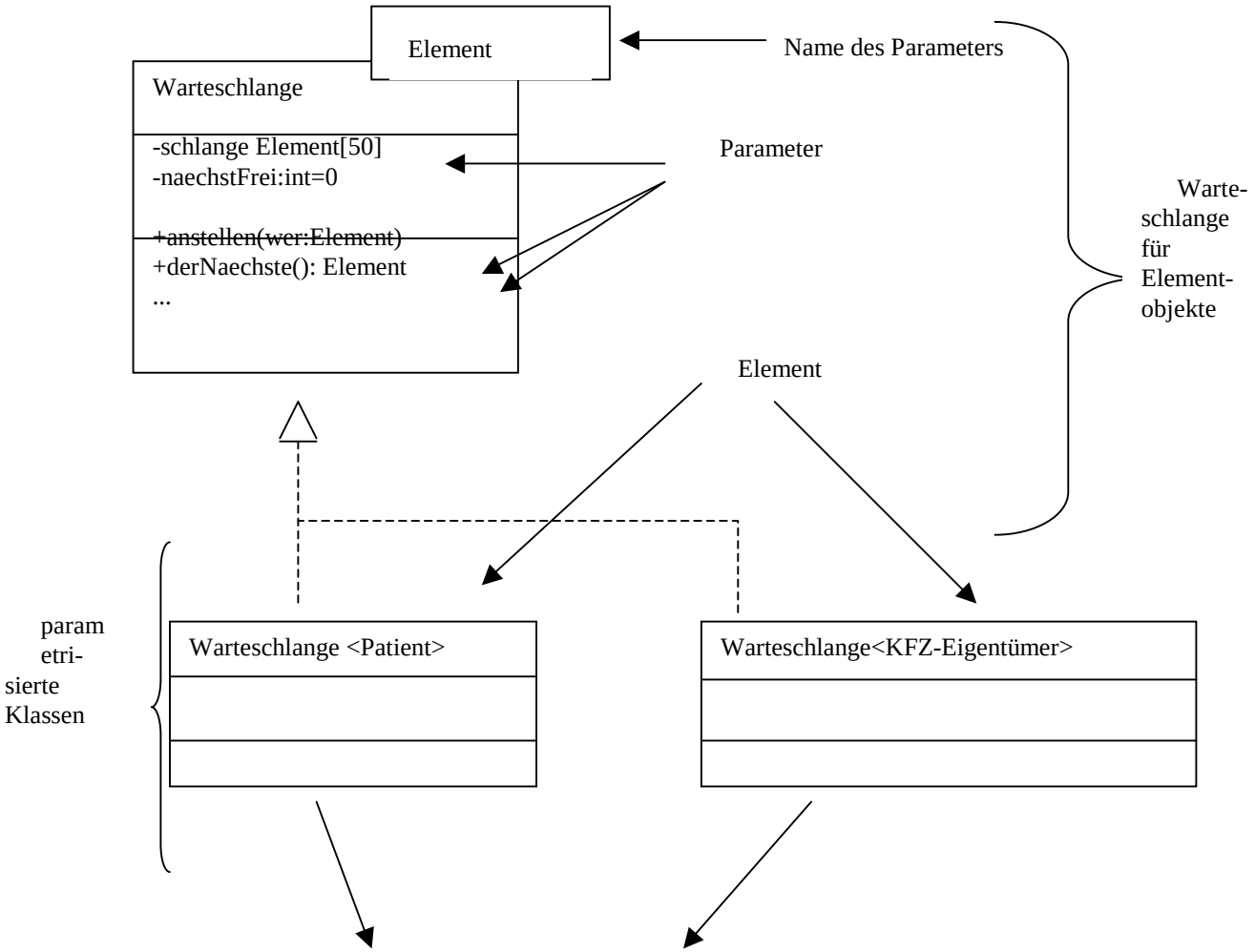
- Existenz der Klasse muss bekannt sein.

Welche Methoden oder Attribute von der Klasse FKK werden verwendet?
M.a.W. wieviel interne Struktur von FKK ist bekannt?

Nichts!

Verwende daher statt der vorherigen Festlegung „FKK“ einen Platzhalter, sozusagen einen „Parameter für die Klassen“,
erstelle also eine sog. **„parametrisierbare Klasse“.**

Notation

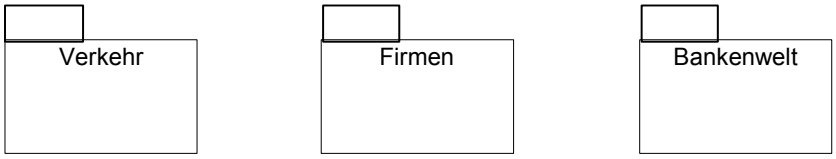


Hier sind keine neuen Methoden, keine neuen Attribute nötig.

von Java nicht realisierbar

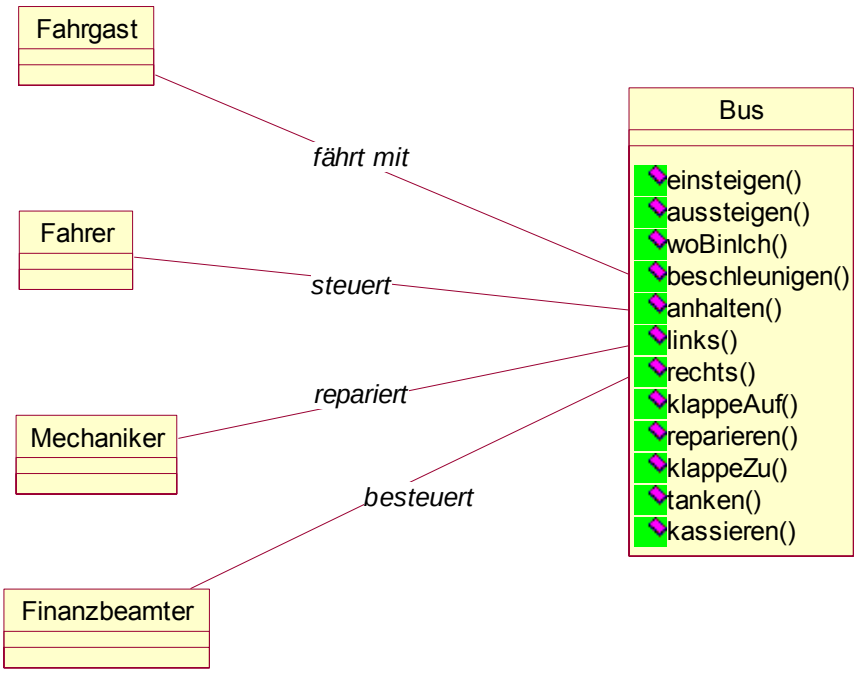
Subsysteme

- benannte, logische Zusammenfassungen zu Bibliotheken oder Modulen



Schnittstellen

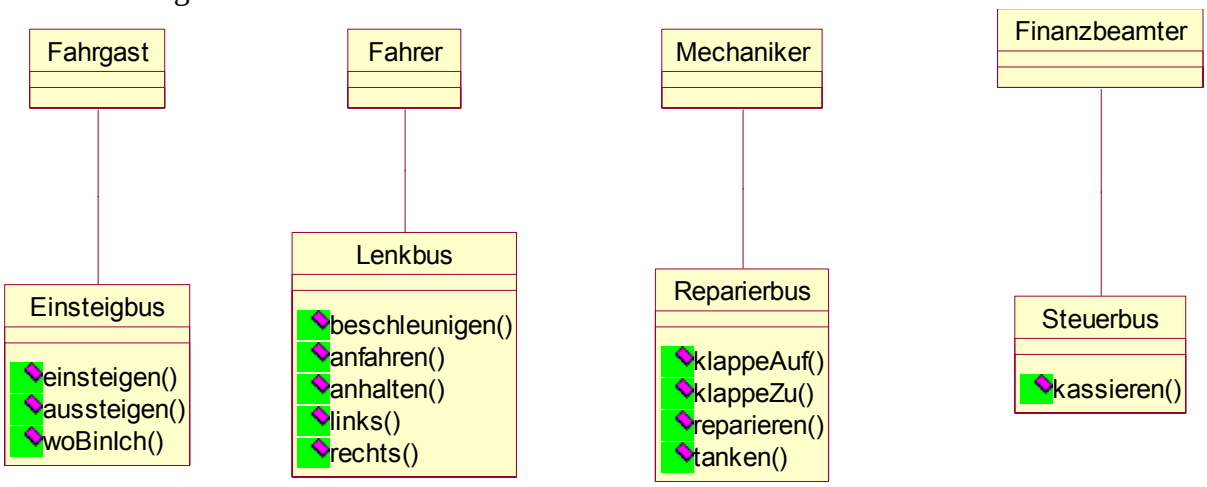
Schnittstellen



- Methodenorientierte Programmierung
- Dienste werden angeboten

1. Problem: zu viele Methoden ➔ unübersichtlich
2. Problem: Jeder Nutzer darf jede Methode aufrufen, d. h. der Fahrgast darf Bus fahren, der Mechaniker darf kassieren.

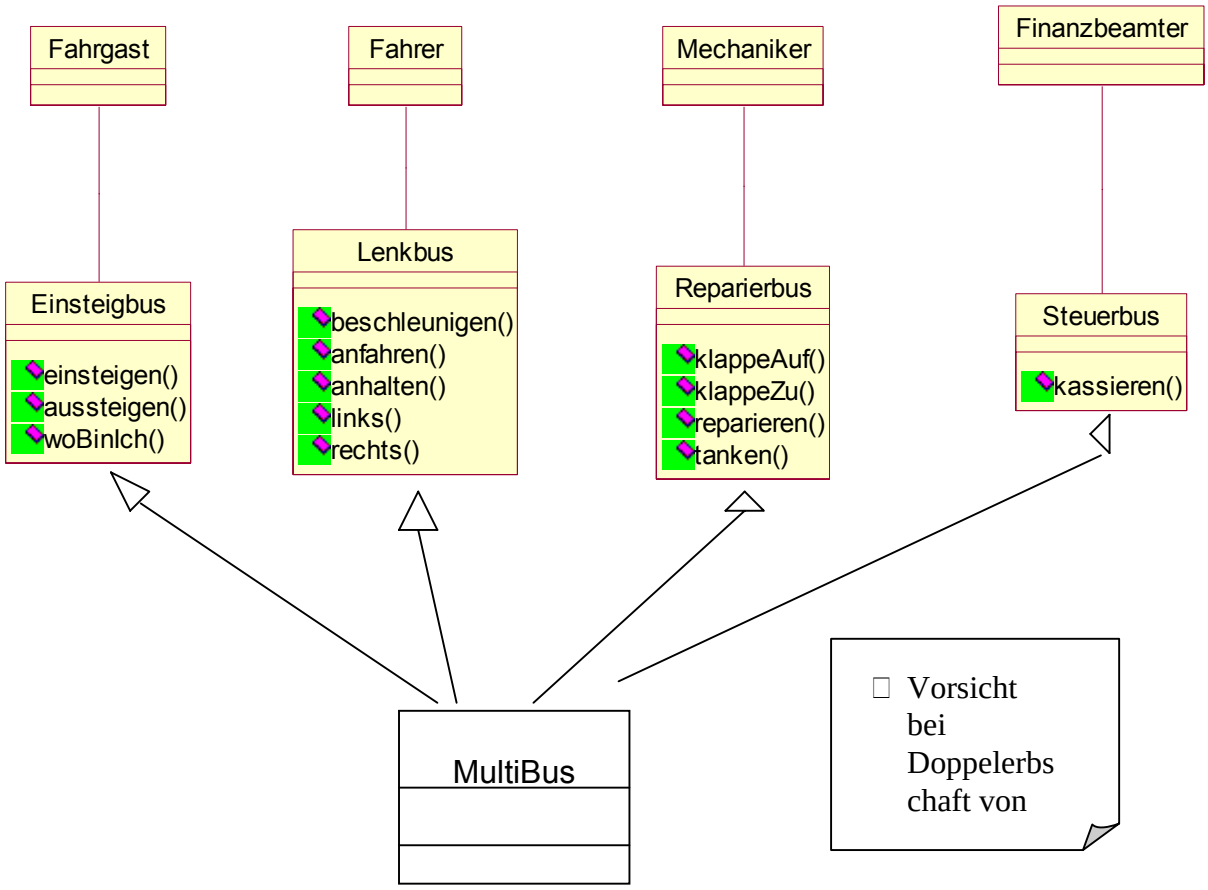
Lösung: Entwirf für jeden Nutzer eine eigene Klasse für den Bus.
Problem 1 und 2 gelöst.



3. Problem
   Für jede der vier Klassen muss ein Objekt existieren. Diese Klassen haben außerdem Überschneidungen (Methode `woBinIch()`!), z.B sind die Attribute `standort` beider Objekte gleich.

   Idee: Nutze

   Mehrfachvererbung**!**



Mehrfachvererbung

Jetzt greift jeder Nutzer über seine Klassen zu.

Bsp. Quelltext

```
MultiBus mb = new MultiBus(); //Objekt erschaffen
```

```
Einsteigbus e = mb;          //eingeschränkte Referenz

e.einsteigen();              //P, ok
e.beschleunigen();           //Fehler! Unbekannt

Lenkbus l = mb;              //nur Referenz

l.links();                   //P, ok
l.woBinIch();                //P, auch ok

ReparierBus r = mb;          //Referenz

r.tanken();                  //P, ok

VerstBus v = mb;             //Referenz
v.anhalten();                //Fehler! Unbekannt
v.kassieren();               //P, ok
```

P = Per Polymorphie wird die Methode aus Multibus aufgerufen, denn e, r, l, v sind identisch mit mb, also eigentlich ein Multibus

Problem: Methode `woBinIch()` doppelt (welche richtig?) !
Daher sinnvoll: Überschreiben, damit immer genau eine –die von MultiBus- genommen wird.

Regeln für die Mehrfachvererbung

1. Keine Attribute vererben (oder höchstens aus einer Klasse)!
2. Methoden grundsätzlich überschreiben (zumindest alle bis auf die aus einer ausgewählten Klasse)

Fazit:

Erben von einer Klasse ist ok.
Attribute von anderen Klassen nicht erben.
Methoden von anderen Klassen überschreiben.

Regel:

Soll eine Klasse als Basisklasse bei der Mehrfachvererbung verwendet werden, gilt:

1. Lege alle Attribute an!
1. Lege alle Methoden als „abstrakt“ an, d.h. sie müssen überschrieben werden.
   (Dann ist die ganze Klasse abstrakt.)

Zusammengefasst:

Lege die Basisklasse als abstrakte Basisklasse ohne Attribute mit nur abstrakten Methoden an. Eine solche Klasse enthält also nur noch **Namen und Parameterlisten** für Methoden, also eine Liste von **Funktionsvereinbarungen.**

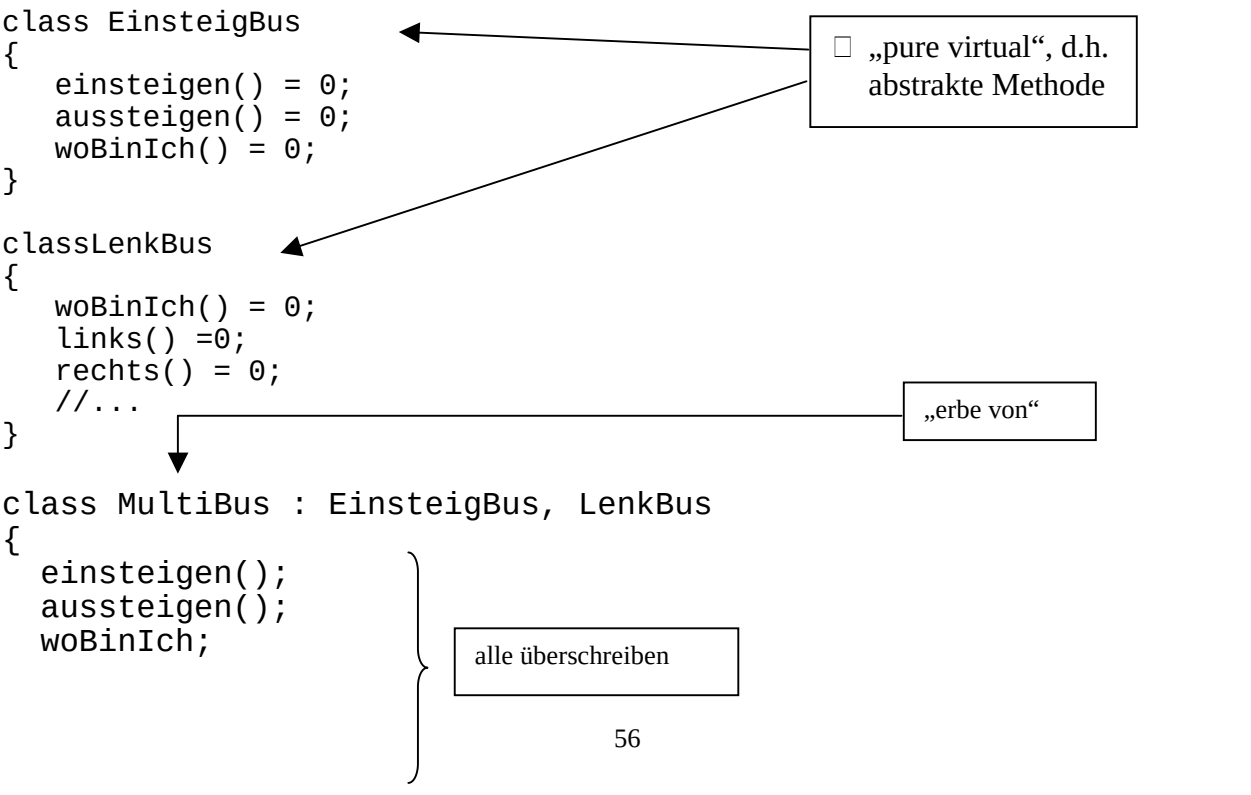
Eine solche Konstruktion heißt

Schnittstelle.

C++: Es gibt keine expliziten „Schnittstellenklassen“, verwende stattdessen die o.g. „abstrakte Basisklassen ohne Attribute mit nur abstrakten Methoden“ sowie die Mehrfach-vererbung.

Java: Es gibt keine Mehrfachvererbung, erbe stattdessen von einer Klasse, implementiere Darüber hinaus beliebig viele Schnittstellen.

C++:



```
  links();
  rechts();
  //...
}
```

Java

```
interface EinsteigBus
{
  einsteigen();
  aussteigen();
  woBinIch();
}
```

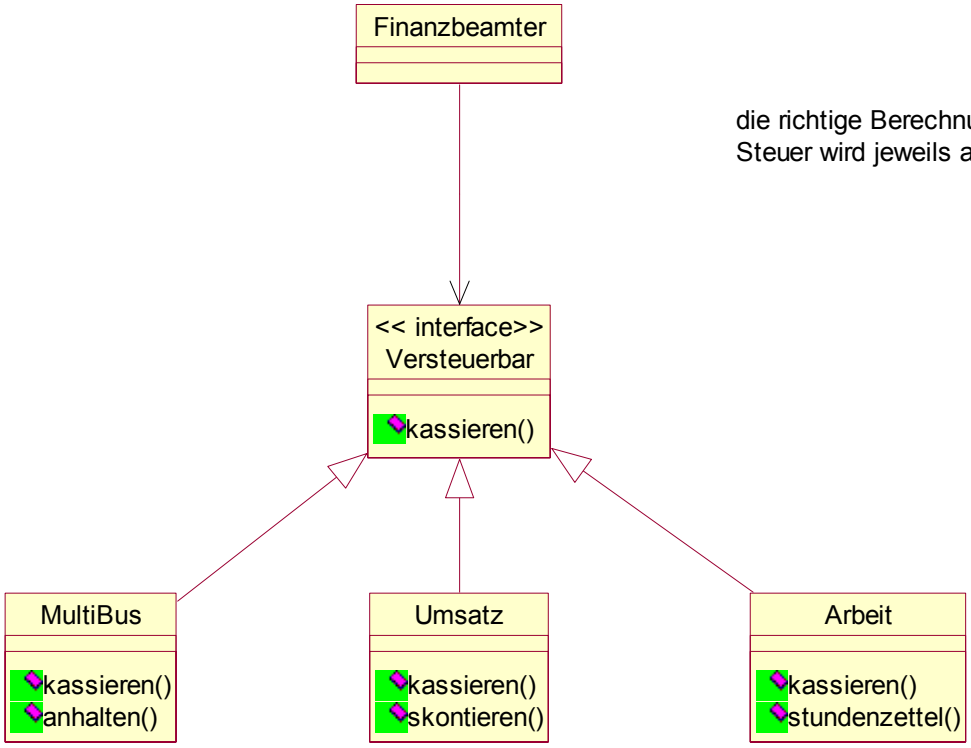
Schnittstellenmethoden sind grundsätzlich abstrakt.

```
interface LenkBus
{
  woBinIch();
  links();
  rechts();
  //...
}

class MultiBus implements EinsteigBus, LenkBus
{
  einsteigen();
  aussteigen();
  woBinIch();
  links();
  rechts();
//..
}
```

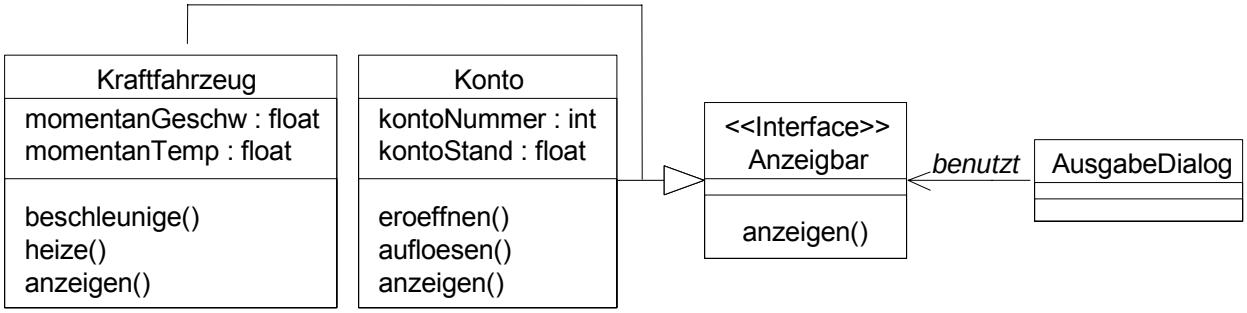
~ Implementiert folgende Schnitt-

Ein Objekt wird nur von Klasse MultiBus erzeugt.

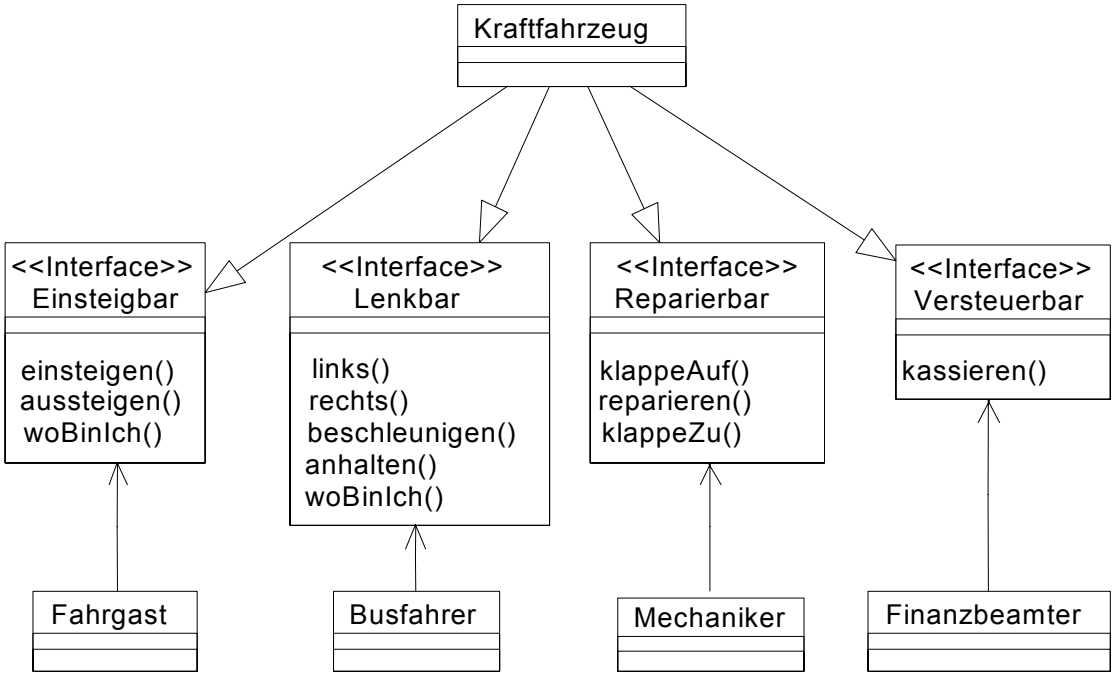
Anderes Beispiel:



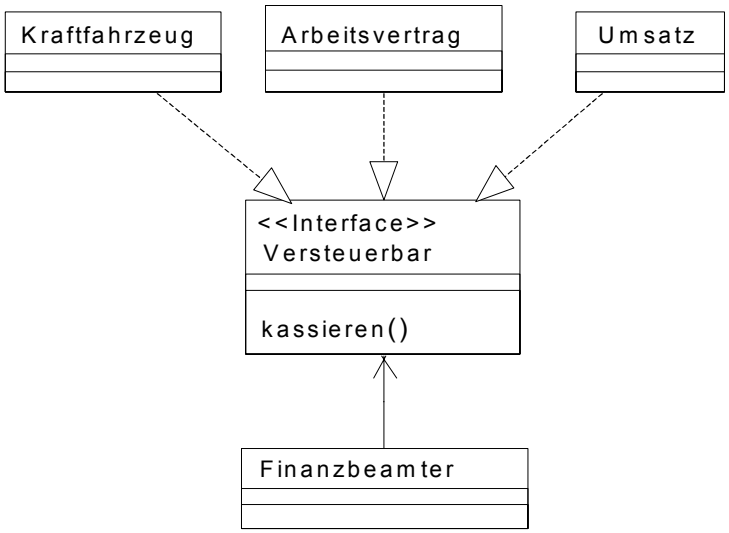
- Bereitstellung von Verhaltensmustern, die erst später implementiert werden



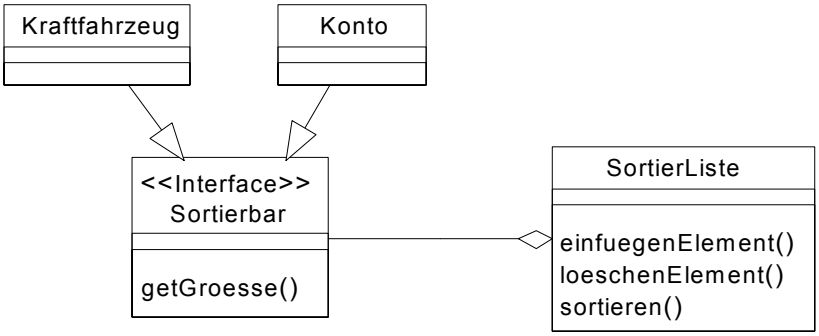
- Beispiel: Schnittstelle zum eingeschränkten Zugriff auf Objekte einer Klasse



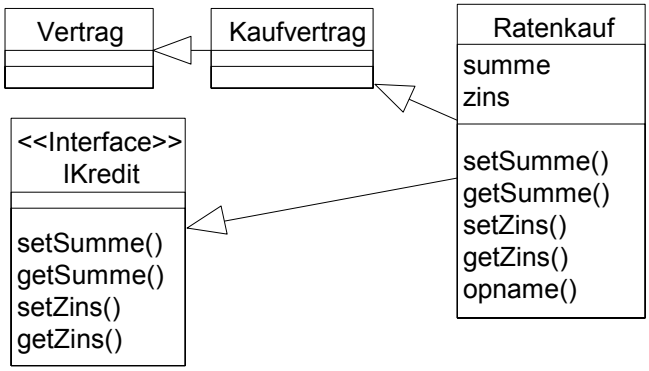
❖ Beispiel: Dieselbe Schnittstelle bietet einheitlichen Zugriff auf verschiedene Klassen



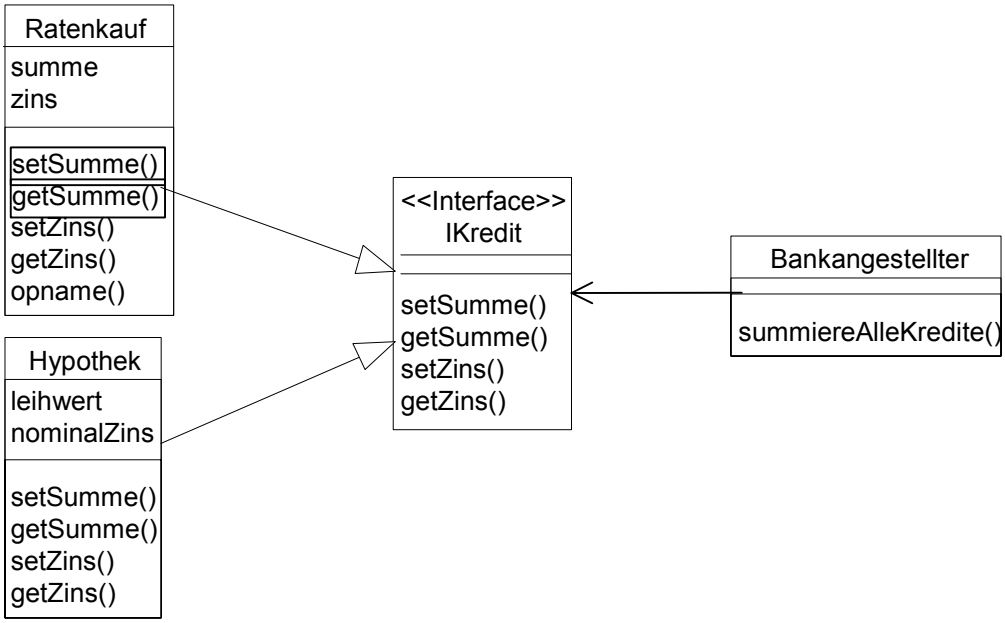
❖ Beispiel: Schnittstelle „Sortierbar“ statt Verwendung einer generischen Klasse



❖ Beispiel: Schnittstelle zur Simulation von Mehrfachvererbung



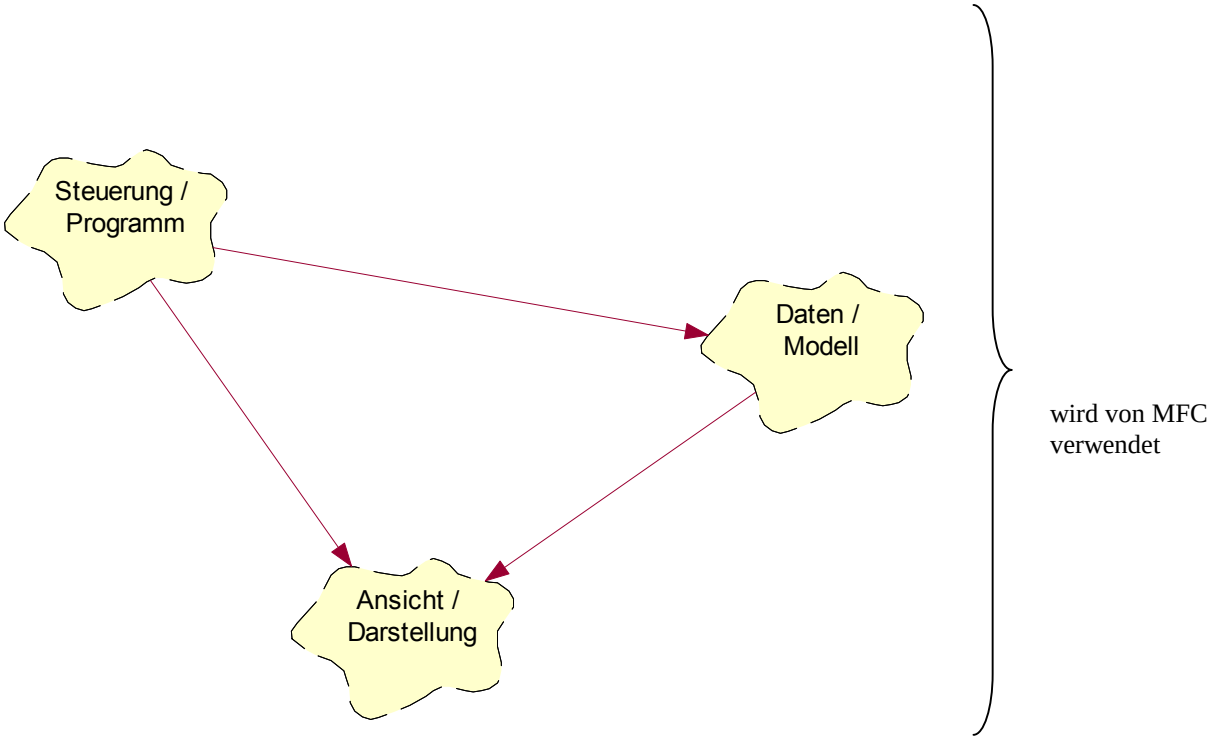
❖ Beispiel: Dieselbe Schnittstelle bietet einheitlichen Zugriff auf verschiedene Klassen



Exkurs: MFC – Microsoft Fondation Class

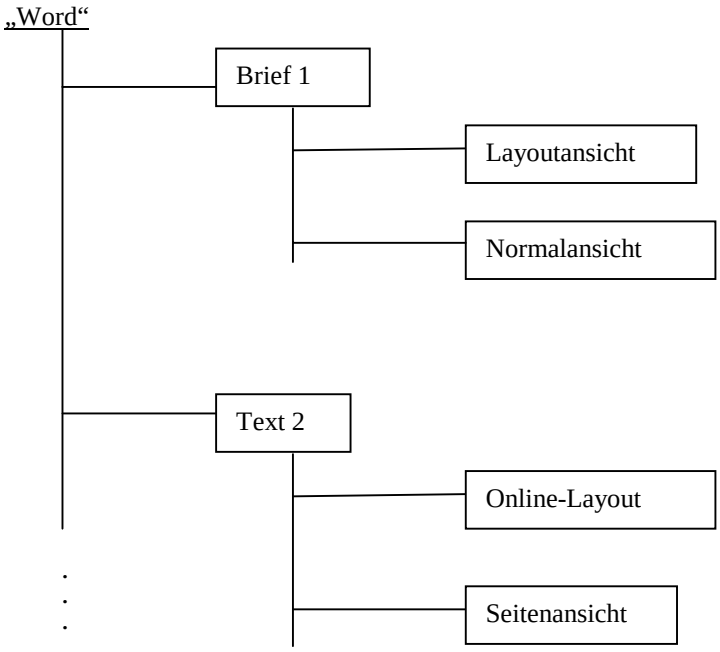
➔ Sammlung von Klassen
➔ Trennen **Kontrollfluß**, **Daten** und ihre **Darstellung**

♦ **Entwurfsmuster** („Patentlösung“) ➔ „Model – View – Controller“
= allg. Modell



Übung 14: Anwendungsarchitektur Programm – Dokument – Ansicht

Gegeben sei die folgende *Anwendungsarchitektur* (Entwurfsmuster)
1. Anwendungsrahmen →Ablaufkontrolle
2. Dokumentenklasse → Daten (beliebig viele pro Anwendung)
3. Ansichtsklassen →View (beliebig viele pro Dokument)



Aufgabe: Klassenstruktur, Nachrichtenfluß

Entwirf für die genannte Klassenstruktur (nur notwendige Methoden und Attribute) sowie den Nachrichtenfluß (entweder Kollaborations- oder Sequenzdiagramm) für das Eintreffen der Nachricht `paintAllDocuments()` beim Rahmen.

Übung 15: Erweiterung der Architektur um Klasse Textclip

Erweitere diese Architektur um eine Klasse TextClip zur Darstellung von (beliebig vielen) zusätzlichen Textschnipselchen.
Diese gehören zum Dokument und werden in jeweils allen Ansichten dargestellt.


```
Hierdurch muss die Methode ergänzt werden:
Dokument.zeigeAlleDarstellungen()
{
 for each Ansicht  a do a.anzeigen();
 for each TextClip t do t.anzeigen();
}
```

Völlig getrennte Methoden

Übung 16: Erweiterung um Attribut `ownerDraw()`

Erweitere das TextClip um ein Attribut `ownerDraw() (besitzerGezeichnet)`, entscheide hiermit, ob das TextClip sich – wie bisher – selbst zeichnet oder ob es seinen Besitzer
(Dokument) auffordert, dies zu tun.

Problem:
TextClip ist abhängig von Dokument. Falls eine weitere Klasse von Dokument abgeleitet wird, ist dies unschädlich, funktioniert dank Polymorphie genau wie vorher. Sobald aber eine neue- nicht von Dokument abgeleitete – Dokumentenklasse verwendet werden soll, kann TextClip nicht unverändert verwendet werden.

Übung 17: Entkopplung
Löse das genannte Problem!
Entkopple TextClip vom Dokument durch Zwischenschalten einer Schnittstelle!

Objektorientierte Analyse

Ziele

- Probleme verstehen, Organisation aufdecken
- Komplexität auflösen, Dekomposition einer Anwendung
- Anforderungen beschreiben
- Objekte im Problembereich
- Analyse soll das Problem beschreiben:
- Was soll das System tun? (aus Anwendersicht)

Vorgehensweise

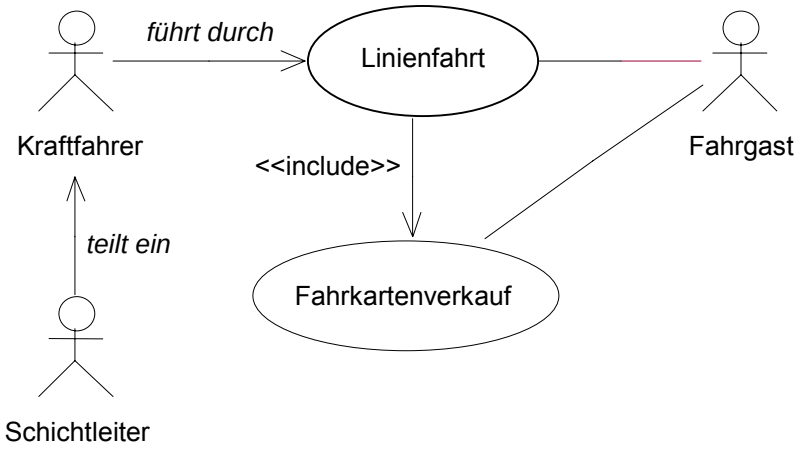
- Vollständig dokumentiertes logisches Modell des relevanten Realweltausschnittes
- Nur das modellieren - was jetzt und „in naher Zukunft“ gebraucht wird
- Darstellung des Anforderungsverhaltens in verschiedenen Modellen und Diagrammen
- Ermittlung der Abstraktionen, die den Anforderungen zugrunde liegen
- Objekte und Objektverhalten finden und beschreiben
- Statische Zusammenhänge zwischen Objekten finden (Beispiel: Bus – Motor)
- Dynamische Zusammenhänge zwischen Objekten finden (Beispiel: Bus – Fahrgast)
- Klassen identifizieren
- Zu Generalisierungen zusammenfassen und per Vererbung spezialisieren
- Assoziationen modellieren
- Aggregationen und Kompositionen erkennen und modellieren

Ergebnisse

Statische Sicht:

- Objekte
- Klassen
- Strukturen
- Nachrichtenverbindungen

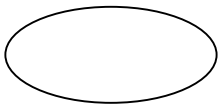
❖ Darstellungsmittel: Anwendungsfalldiagramme



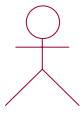
- beschreibt „Geschäftsvorfälle“ als Ganzes
- Nicht zur funktionalen Zerlegung!
- Zur Kommunikation mit dem Anwender
- Anwendungsfall: typ. Interaktion des Anwenders mit dem System
- Kein Designhilfsmittel, nur Anforderungsanalyse

Symbole:

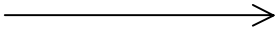
Anwendungsfall



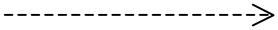
Akteur



Abhängigkeit/Nutzung

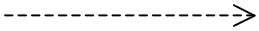


„schließt ein“



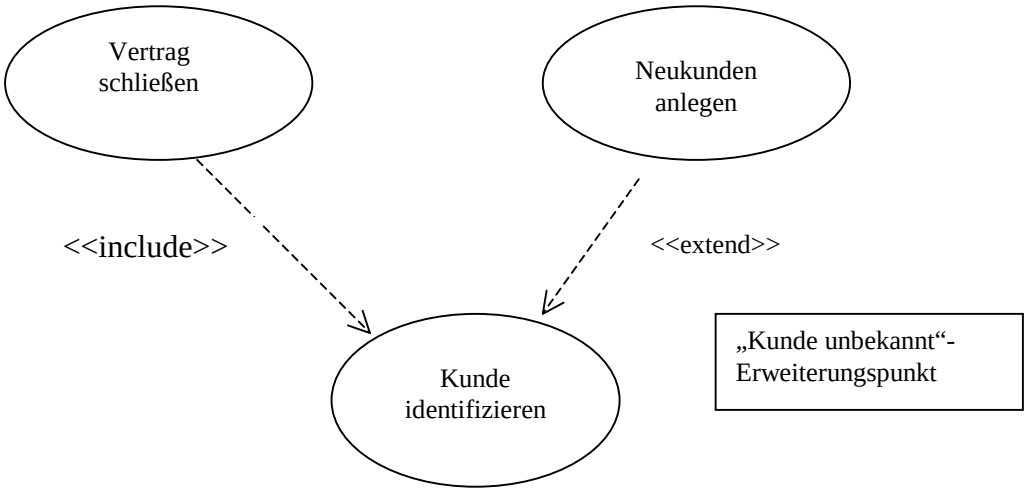
<< include>>

„erweitert“



<<extends>>

Beispiel:



Textuelle Beschreibung der Anwendungsfälle (zusätzlich -Vorschlag!):

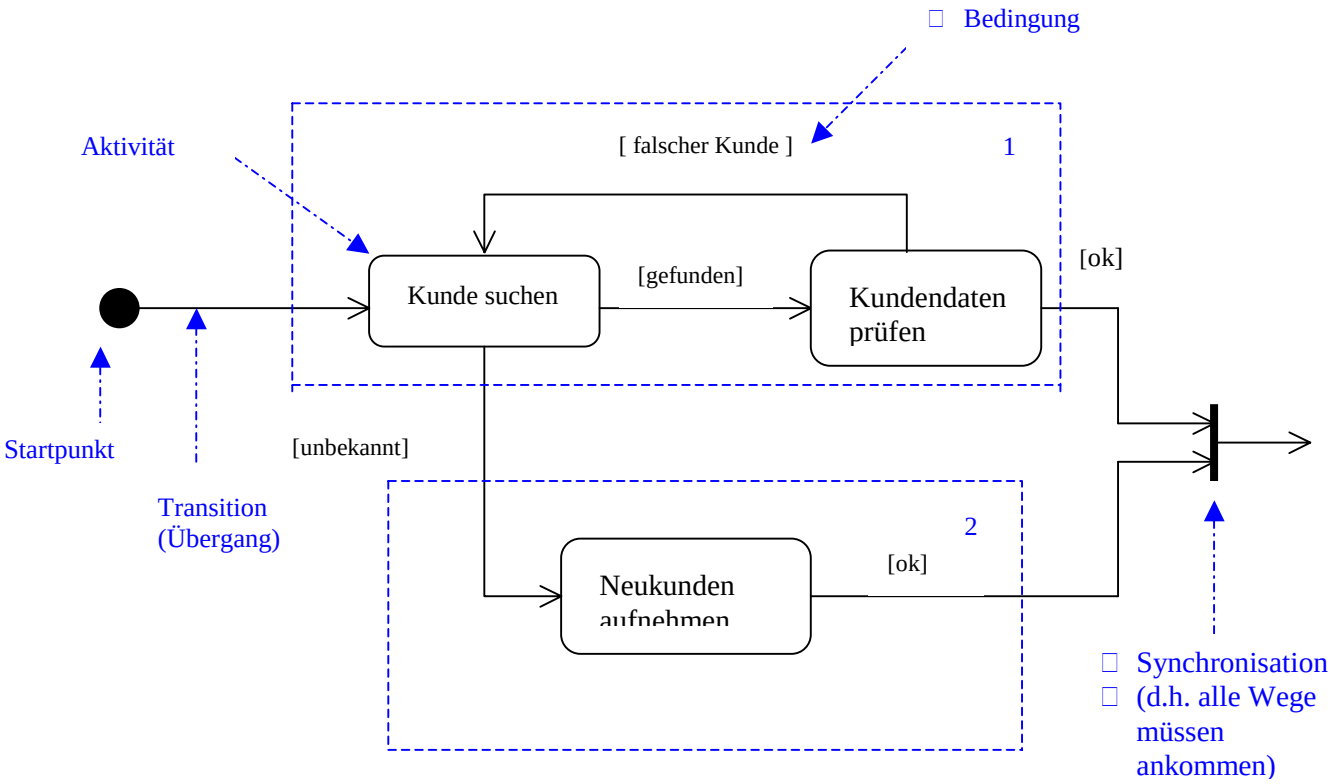
- Nummer und Name, Kurzbeschreibung
- Auslöser, Vorbedingung: Erwarteter Systemzustand vor Eintreten
- Ergebnisse, Nachbedingung: Erwarteter Systemzustand nach Abschluss
- Nicht-funktionale Anforderungen, z.B. Plattform, Prioritäten
- Ablaufbeschreibung, gegliedert nach separat beschriebenen Einzelaktivitäten
- Ausnahmen, Varianten, Abweichungen, Alternativen
- Offene Punkte, Fragen

Daumenregel: Pro Aufwandsjahr etwa 5 Use Cases.

- Darstellungsmittel Aktivitätsdiagramm

Aktivitätsdiagramm

- Verfeinerung der Anwendungsfälle durch Modellierung von Einzelaktivitäten
- Jede Einzelaktivität der Use Case-Beschreibung wird hier eine Aktivität)

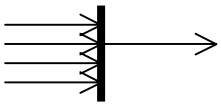


*1 : Use Case „Kunde identifizieren“
2 : Use Case „Neukunde anlegen“

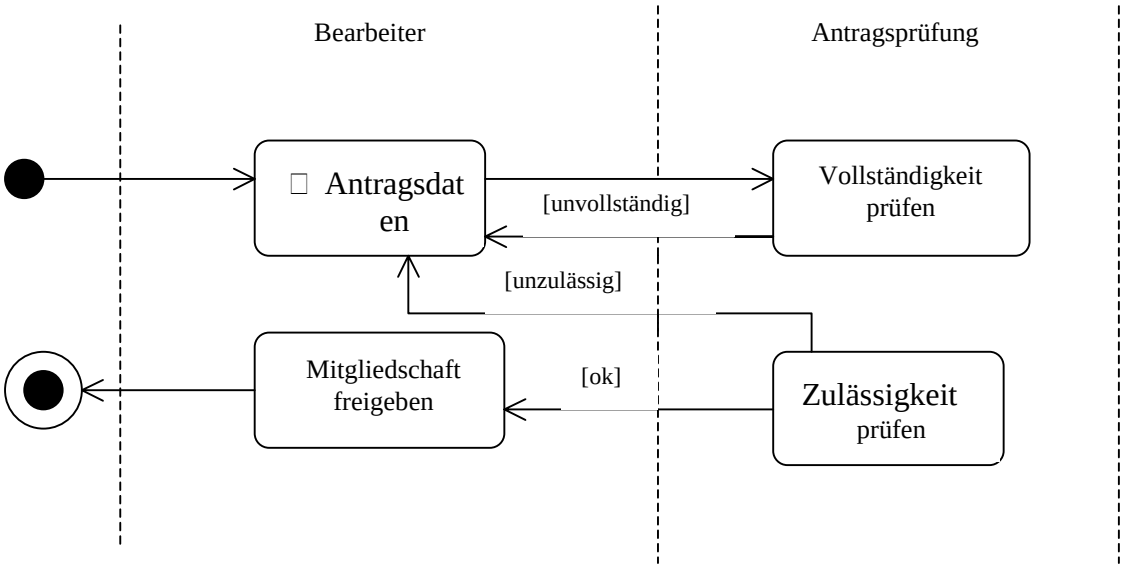
*Anwendungsfälle nur zur Erläuterung eintragen, sie gehören nicht ins Aktivitätsdiagramm/

- Aktivitätsdiagramme entsprechen grob dem (bekannten) Programmablaufplänen/ Flussdiagrammen
- Aktivitäten können in Einzelaktivitäten zerlegt werden. (Dies sollte jedoch erst später geschehen.)
- Erweiterung der Symbolik (Oesterreich):
  Markiere einen Synchronisationsbalken mit der **Zusicherung**

  {AND} – alle Wege müssen eintreffen
  {OR} – mindestens ein Weg trifft ein
  {XOR} – ausschließendes OR
  {>=3} – mindestens 3 Wege kommen an

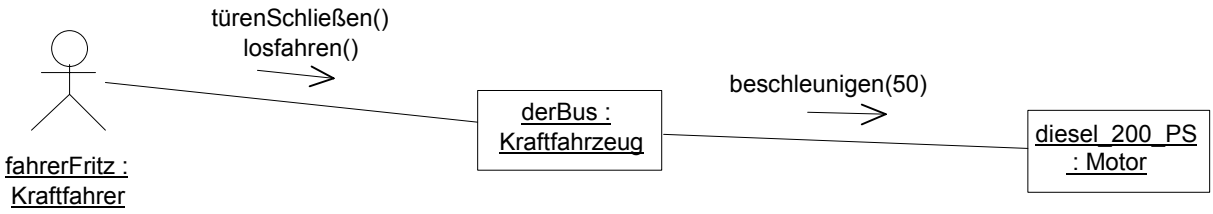


- Zuweisung von Verantwortlichkeiten durch Anordnung in sogenannten „SwimLanes“



- Darstellungsmittel: Kollaborationsdiagramme

Darstellung, welche Objekte durch welche Nachrichten miteinander kommunizieren –



Betonung auf die Frage WER!

- Darstellungsmittel: Moduldiagramme
- Darstellungsmittel: Objektdiagramme

Zeigt die am System beteiligten Objekte – eventuell prototypisch, wo möglich aber vollständig – sowie die zwischen ihnen bestehenden Beziehungen

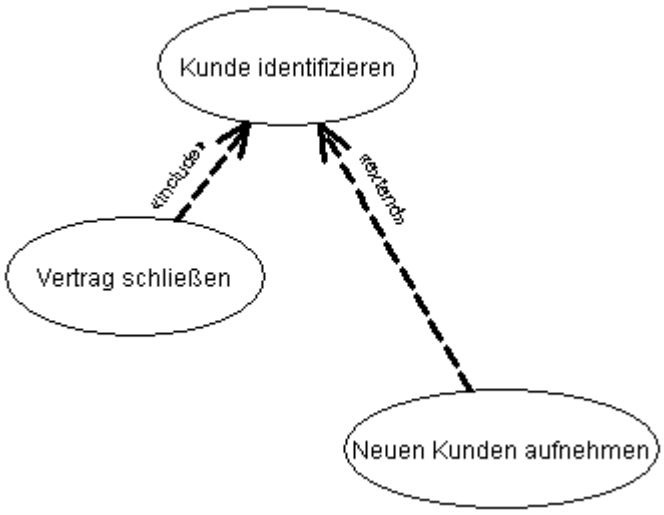
- Darstellungsmittel: Klassendiagramme

Zeigt die am System beteiligten Klassen – so weit fachlich notwendig, vollständig – sowie die zwischen ihnen bestehenden Beziehungen: Vererbung, Assoziationen und Aggregationen

Anforderungsanalyse

Anwendungsfalldiagramme

- Beschreibung einer typischen Interaktion eines Anwenders mit dem System
- Anwendungsfall entspricht etwa einem Geschäftsvorfall
- Keine Beschreibung des internen Verhaltens, keine funktionale Zerlegung
- Kein Designhilfsmittel sondern nur für die Anforderungsanalyse



- Textuelle Beschreibung zusätzlich zum Anwendungsfalldiagramm

  Nummer und Name des Anwendungsfalles sowie eine Kurzbeschreibung

  Auslöser sowie Vorbedingungen: Erwarteter Systemzustand vor dem Eintreten

  Ergebnisse und Nachbedingungen: Erwarteter Systemzustand nach dem Durchlaufen

  Nicht-funktionale Anforderungen: Design, Plattformfragen, Entwicklungsprioritäten

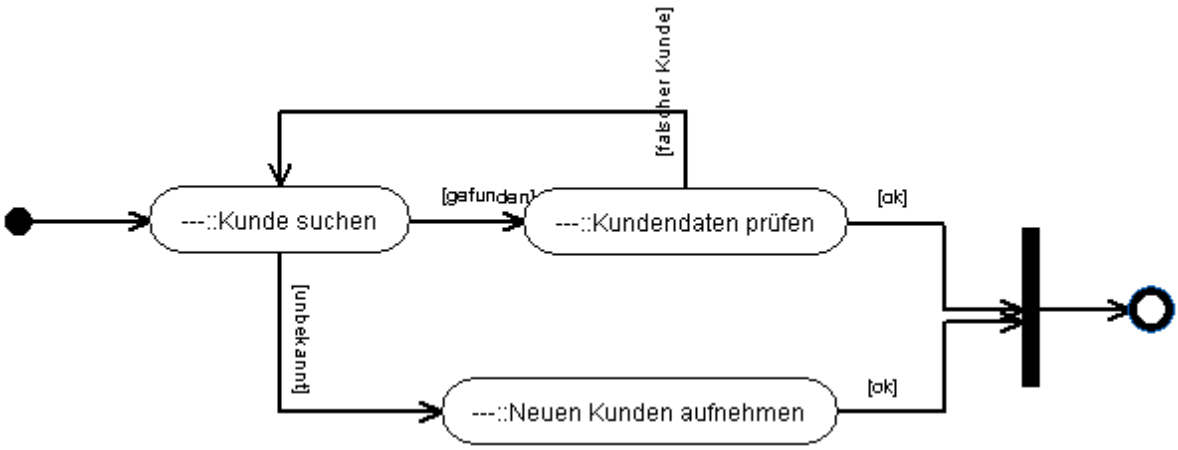
  Ablaufbeschreibung, gegliedert in Einzelschritte, jeder davon separat beschrieben

  Ausnahmen und Varianten: Abweichungen vom Normalfall, alternatives Verhalten

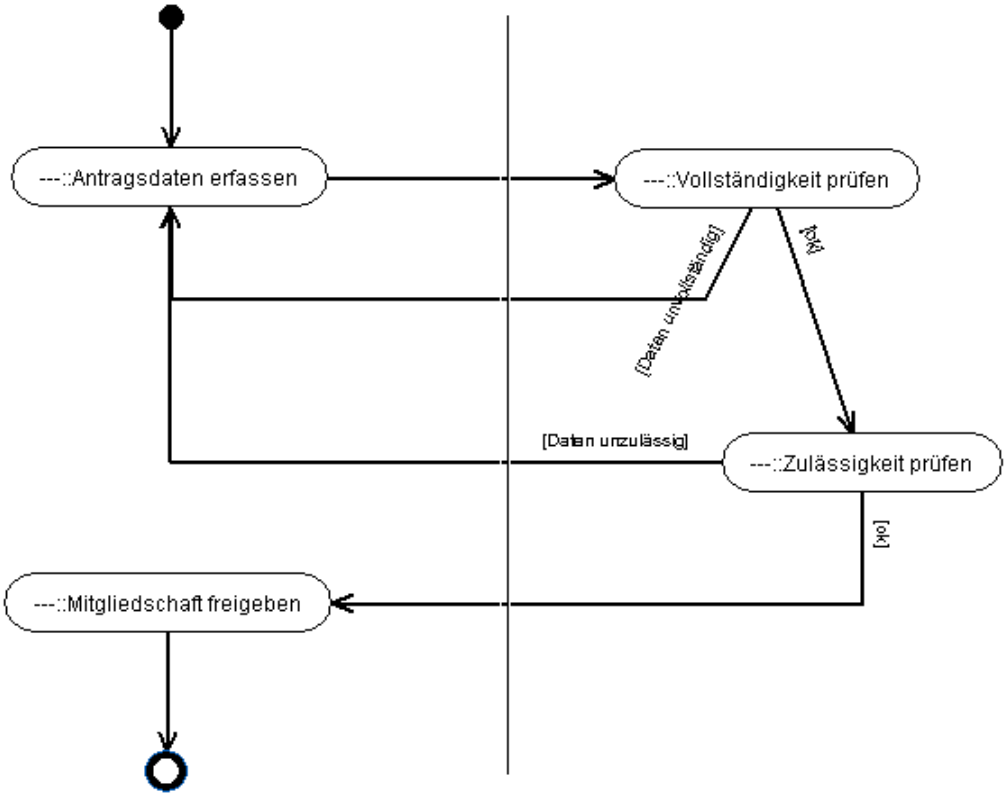
  Offene Punkte, Fragen, Dokumente, Referenzen
- Daumenregel: Pro Aufwandsjahr etwa fünf Anwendungsfälle
- Anwendungsfälle beschreiben, *was* das System leisten soll, nicht *wie* es das macht
- Prinzipiell auch nicht-softwareunterstützte Aktivitäten möglich

Aktivitätsdiagramme

- Verfeinerung der Anforderungen durch Modellierung von Einzelaktivitäten
- Einzelschritte der Anwendungsfälle sind Kandidaten für Einzelaktivitäten



- Zusätzliche Zuweisung der Verantwortlichkeiten
- Aufteilung in verschiedene „SwimLanes“



Beispiel: Versicherungsvertrag

Anwendungsfallanalyse

- Anwendungsfalldiagramm

- Beschreibung des Anwendungsfalles „Vertrag schließen“

Beschreibung

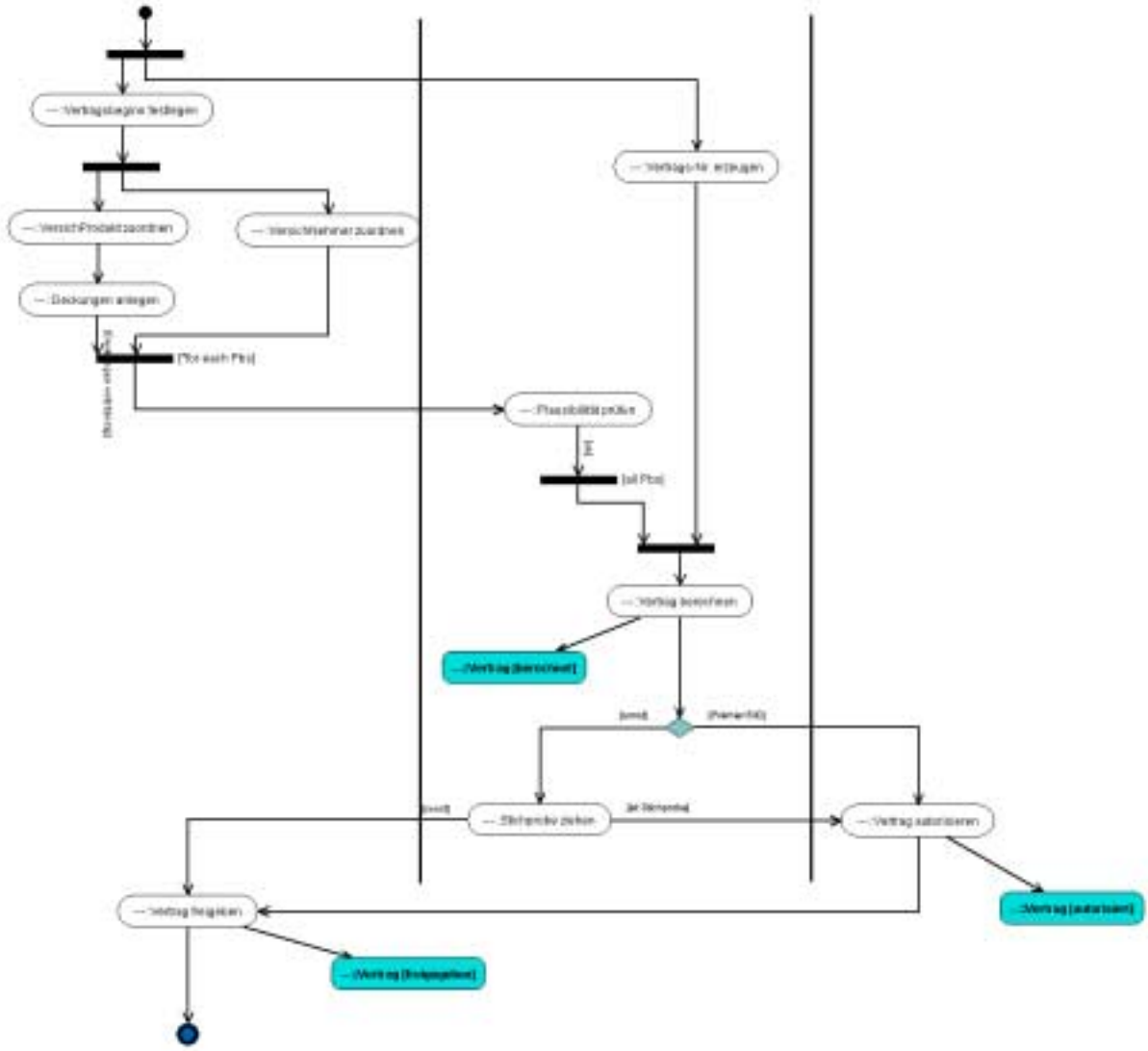
1. Beginn der Vertragslaufzeit eingeben
2. Produkt auswählen
   Der Anwender wählt ein Produkt aus, um es dem Vertrag zuzuordnen. Zur Auswahl werden ihm alle Produkte angeboten, die zum eingegebenen Vertragsbeginn gültig sind.
3. Vertrags-Nr. wird erzeugt
   Die Vertragsnummer wird automatisch erzeugt und eingeblendet.
4. `include` ⇒ Anwendungsfall „*Versicherungsnehmer zuordnen*“
5. Beitragszahler, Postempfänger und Leistungsempfänger festlegen
   [...]
6. [...]
7. `include` ⇒ Anwendungsfall „*Deckungen anlegen*“
8. Vollständigkeit und Plausibilität prüfen sowie berechnen
9. Freigeben

Variationen

4. Statt einen vorhandenen Versicherungsnehmer zu suchen und auszuwählen, wird ein neuer Versicherungsnehmer angelegt.
8. Ist die berechnete Prämie >500, muß der Vertrag speziell geprüft und autorisiert werden. `extend` ⇒ Anwendungsfall „*Vertrag autorisieren*“

Aktivitätenmodellierung

- Aktivitätsdiagramm

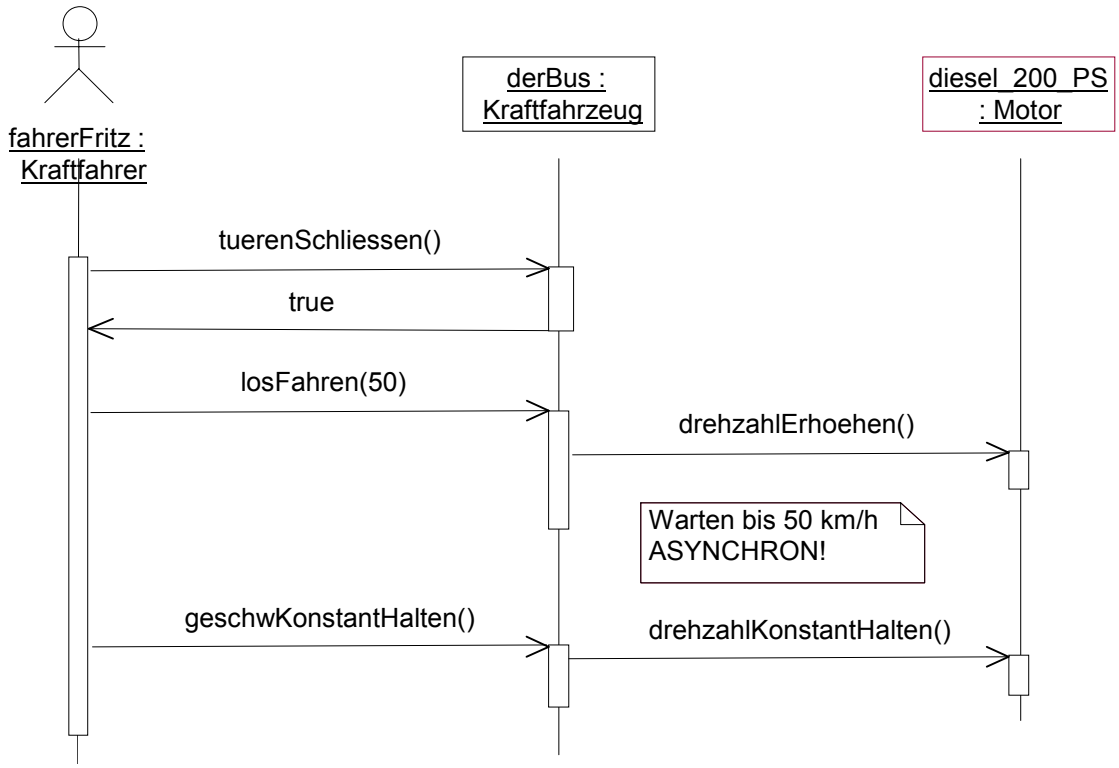


- Transitionen
- Synchronisation

Darstellung von Objektzuständen

Dynamische Sicht:

- Objektverhalten
- Kommunikation
- Folgen von Methodenaufrufen
- Kontrollfluss
- Darstellungsmittel: Sequenzdiagramme



- Darstellungsmittel: Zustandsdiagramme

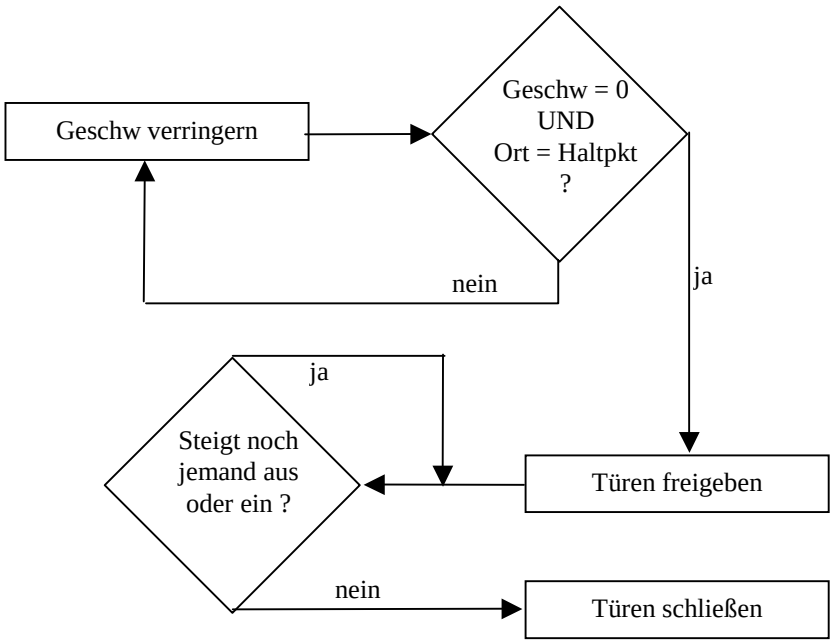
Funktionale Sicht:

- Beschreibt funktionale Anforderungen, die aus der Problembeschreibung hervorgehen. (keine Programmfunktionen)
- Algorithmen, Methoden, Funktionen
- Darstellungsmittel sind beliebig (Struktogramme, Flussdiagramme, Quelltext...).

- Algorithmen

```
Algorithmus „Bushaltestelle anfahren“:
Erreiche Haltebucht der Haltestelle,
dabei verringere Geschwindigkeit auf Null, gib
alle Türen frei,
warte bis niemand mehr ein- oder aussteigt,
schließe die Türen.
```

- Realisierung der Methoden
- Darstellungsmittel: Ablaufdiagramme



- Darstellungsmittel: Struktogramme
- Darstellungsmittel: Pseudocode

```
Pseudocode „Bushaltestelle anfahren“:
WHILE ( Geschw > 0 ) AND ( NOT isHaltestelleErreicht() )
      DO Geschw = MAX( Geschw – 10, 0 )
gibFreiTueren()
WHILE ( jemandSteigtEin() OR jemandSteigtAus() )
      DO warte( 1 )
schliesseTüren()
```

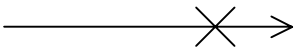
Exkurs Synchrone und asynchrone Kommunikation

- Synchrone Kommunikation:

Objekt A schickt Objekt B eine Nachricht („Brief“)und wartet auf Antwort von Objekt B.

- Brief ohne Absender (void)
- Brief mit Rückschein ohne Absender

Die Antwort von B gehört zum Nachrichtenfluß dazu, Auftrag und Antwort bilden ein untrennbares Paar.



- Asynchrone Kommunikation:

  „Senden und Vergessen“ –

  Objekt A schickt Objekt B ein Ereignis.
  A startet, ohne auf Antwort zu warten, gleich eine neue Aktion. Wann der Empfänger die Nachricht annimmt, interessiert den Absender nicht.
  Falls Objekt B eine Antwort schickt, ist die ein erneutes Ereignis. Damit B weiß, wem es die Antwort zurückschicken soll, muss in dem Ereignis, das A verschickt, die Adresse von A explizit als (Parameter) **Absender** mitgegeben werden. Im letzteren Fall oft eine andere als die von A.

Allerdings wird die Entscheidung, ob eine Kommunikation synchron oder asynchron durchgeführt wird, in der Regel erst in der Entwurfsphase getroffen.

Statisches Modell

Vorgehensweise zur Modellierung

- Identifizieren von Objekten und Klassen
- Identifizieren von Verantwortlichkeiten und Attributen
- Identifizieren von Klassen- und Objektbeziehungen
- Identifizieren von Strukturen
- Identifizieren von Methoden und Nachrichtenverbindungen
- Zerlegen in Teilsysteme

Identifizieren von Objekten und Klassen

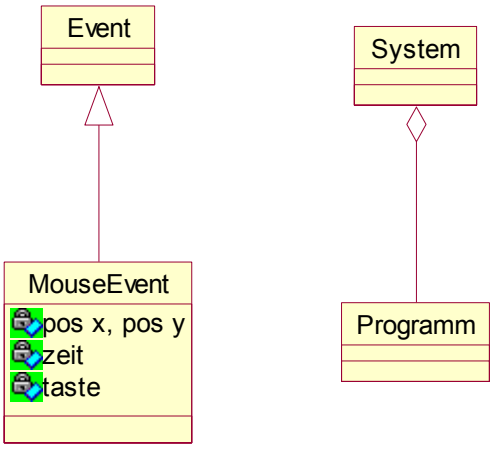
- Personen, Rollen, Orte, Dinge, Geräte, Einrichtungen
- Beschreibungen, Konzepte
- Ereignisse, Interaktionen
- Kandidaten für Objekte:

Aus der Problembeschreibung werden Objekte und Klassen extrahiert.

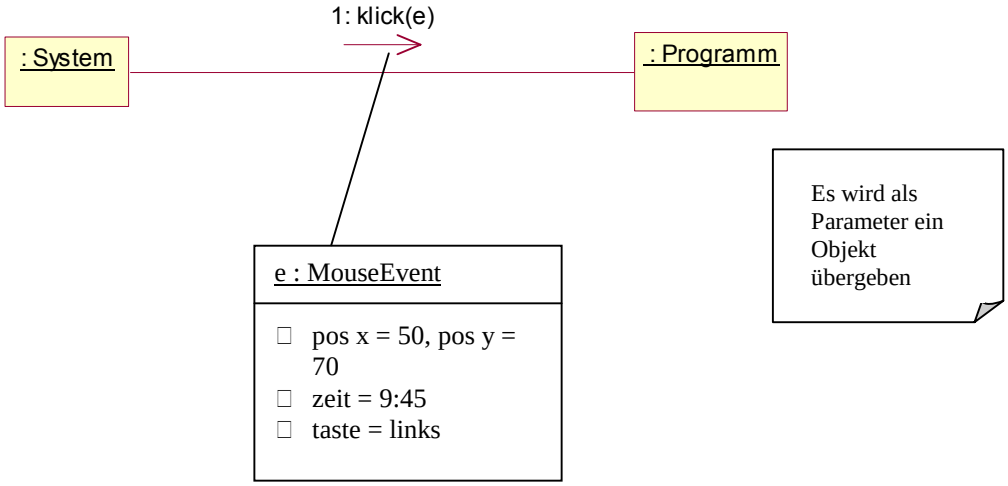
Substantive, z.B. Dinge, Prozesse, Ereignisse

Hier beachten: **Problemrelevanz!**

Insbesondere in Java:
Ereignisse = Objekte



- Organisationen, Einheiten, Systemstrukturen



- Relevanz für das zu modellierende System (Realweltausschnitt!)

Hilfsmittel zur Identifikation von Objekten und Klassen

- Objekte identifizieren und klassifizieren
- Verwendung von **C R C** - Karten

C : Class = Klasse
R : Responsibilities = Verantwortung
C: Cooperations (Collaborators) = Kooperationen (Zusammenarbeit)

Verwende Karteikarte oder Papierblatt (nicht zu groß!) mit folgender Aufteilung:

Klasse (Name, Definition): Busfahrer **Änlichkeiten mit**		**Objekte (Beispiele)** Max	
Verantwortlich für	**Typ**	**Arbeitet zusammen mit ... um ...**	
fahrKartenVerkauf()	!	Fahrkartenautomat	gibFk
losfahren()	!	Bus	
...	...	...	...

Aufgabe, die die Objekte dieser Klasse erfüllen

Klasse von der ein Objekt benötigt wird, damit die Aufgabe durchgeführt werden kann.

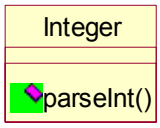
Warum nicht zu groß ?

Damit nicht zu viele Verantwortlichkeiten (also potentielle Methoden) darauf passen! (Ziel : kompakte, übersichtliche Klassen)

Sobald nicht mehr alle Verantwortlichkeiten auf eine Karte passen, sollte diese „Klasse“ auf zwei Karten (also **zwei Klassen!)** unterteilt werden.

Falls eine „Klasse“ Nur eine oder 2 Verantwortlichkeiten hat, ist es oft möglich, diese einer anderen Klasse zuzuschlagen.

- Klassen stellen Dienste bereit, die für die Gesamtfunktionalität benötigt werden
- Klassen sind typischerweise keine bloßen Behälter für Daten
- Klassen enthalten Informationen, deren Speicherung innerhalb des Systems erforderlich ist
- Klassen speichern nur diejenigen Daten, die sie für ihre Methoden brauchen.
- Klassen werden durch mehr als ein Attribut beschrieben
- Klassen werden durch mehr als ein Attribut beschrieben (im Regelfall – Ausnahme: Wrapper - Klassen in Java).



Wrapper-Klasse für den einfachen Datentyp int

parseInt:
nimmt eine Text „s“ und versucht, ihn in eine Ganzzahl zu verwandeln:
„321“ = 321
„aha!“ = <Fehler!>

- Klassen bilden zunächst nur den Problembereich ab
- Klassen enthalten zunächst keine Design- oder Implementierungskonstrukte

Identifizieren von Klassen durch Klassifizierung der Objekte

Prüffragen

a) Welche Objekte haben gleiches oder ähnliches Verhalten?

Zusammenfassung zu einer Klasse

b) Ist die Klasse irrelevant?
Die Klasse hat nichts oder wenig mit dem Problem zu tun.

c) Ist die Klasse zu unscharf?
Die Klasse besitzt keine genau definierbaren Grenzen oder umfasst zu unterschiedliche Dinge.

d) Hat das Objekt unter dem Modellierungsblickwinkel des Systems eine eigene Realität?
Manche Objekte haben nur in der Realität eine eigene Identität.

e) Ist die Klasse eher ein Attribut?
Wenn etwas eine eigenständige Existenz im Problembereich aufweist, wird es zur Klasse. (z.B. Name im System Adressenkatalog)

f) Ist die Klasse eher eine Aktion?
Das Hauptwort, das eine Aktion bezeichnet ist in der Regel kein eigenständiges Objekt, sondern eine Methode.

g) Ist die Klasse ein Implementierungsaspekt?
Bsp.: Menge (set), Liste (list), Tabelle (table), Window - Button- für solche Klassen ist die Analyse nicht zuständig. Sie werden erst in der Designphase hinzugefügt.

Identifizieren von Attributen und Verantwortlichkeiten

- Objekte einer Klasse haben gemeinsames Verhalten und Menge von Zuständen
- Objektverhalten definiert durch Methoden, die für Objekte aufgerufen werden können
- Zustand ist bestimmt durch die Werte seiner Eigenschaften
- Der Objektzustand wird bestimmt durch die Werte seiner „Eigenschaften“; dazu gehören seine Attribute, aber auch seine Beziehungen zu anderen Objekten.
- Eigenschaften sind Attribute, aber auch Beziehungen zu anderen Objekten
- Prüffrage: Wie wird das Objekt allgemein und wie im Problembereich beschrieben?
- Prüffrage: Wie ist die Systemverantwortlichkeit des Objektes?
- Prüffrage: Welche Informationen werden von den Methoden innerhalb des Objektes benötigt?
- Attributspezifikation: Name, Datentyp und Wertebereich, Kurzbeschreibung

Weitere Prüffragen:

a) Welche Daten werden innerhalb des Objekts benötigt, um seine Aufgaben zu erfüllen?
b) Welche Dienste bietet die Klasse an?

Weitere Fragen zu Attribute:

a) Ist das Attribut problemrelevant im Sinne der Analyse?
Wenn das System implementiert ist, ist dann das Attribut in irgendeiner Form an der Benutzeroberfläche sichtbar?

b) Ist das Attribut überflüssig?
Mit ihnen wird nichts getan oder sie erfüllen keinen Zweck.

b) Ist ein „Schnappschuss“ oder eine „Historie“ zu modellieren?

Schnappschuss: Das Attribut enthält einen Wert für einen Zeitpunkt, meist den Zeitpunkt „jetzt“. Ändert sich der Wert, dann steht der Alte nicht mehr zur Verfügung.
Historie: Die alten Daten bleiben erhalten und der neue Wert wird hinzugefügt.

c) Sind identifizierende Attribute notwendig?
Nur wenn ein Attribut fachlich notwendig ist- und es zufällig gleichzeitig ein identifizierendes Attribut ist – wird es in das Modell aufgenommen.
z.B.: Fahrgestellnummer eines Autos

d) Ist das Attribut ein eigenständiges Objekt?

e) Referenziert das Attribut eine andere Klasse?
Wenn ja, dann soll es als Assoziation und nicht als Attribut modelliert werden.

f) Handelt es sich bei den Attributen um interne Werte des Objekts?
Wenn ja, dann sollen sie nicht in das Modell aufgenommen werden, denn hier geht es um die Modellierung des Problemraums und nicht um Realisierungsaspekte.

g) Passen Attribute nicht zu anderen Attributen einer Klasse?
Hier müsste die Klasse evtl. zweigeteilt werden.

Übung 18: *Kaffeeautomat, Teil 1*

Das System simuliert einen Getränkeautomaten für Kalt- und Warmgetränkeautomaten, speziell Auswahl, Zubereitung, Ausgabe sowie Leeren der Kasse und Nachfüller der Zutaten. Erstelle im einzelnen (1. Anwendungsfälle, 2. Einzelaktivitäten)

Übung 19: *Kaffeeautomat, Teil 2*

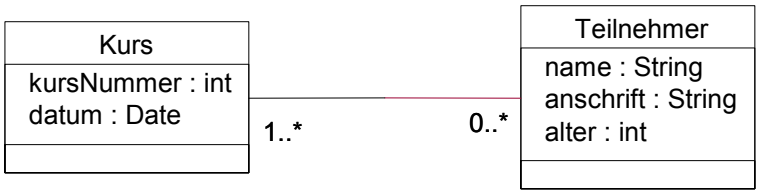
Erstelle ein Aktivitätsdiagramm für den Anwendungsfalls `Getränk beschaffen`

Identifizieren von Klassen- und Objektbeziehungen

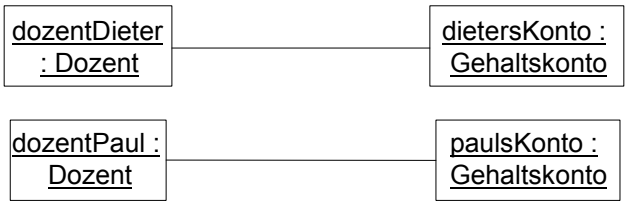
- Spezifikation: Name der Klasse, beteiligte Klassen, Kardinalitäten. Kurzbeschreibung
- Objekte sind zur Erfüllung ihrer Aufgaben auf die Kommunikation mit anderen Objekten angewiesen
- Kommunikationspfad ist Verbindung (Link) zwischen zwei Objekten
- Kardinalität der Beziehungen
- Beispiele für Klassenbeziehungen:
  - Dozent - Gehaltskonto

  - Kurstyp - Kurs

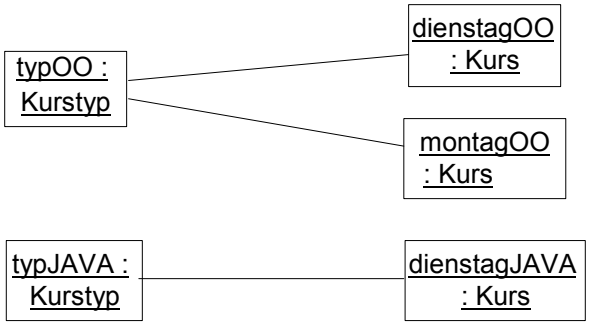
  - Kurs - Teilnehmer



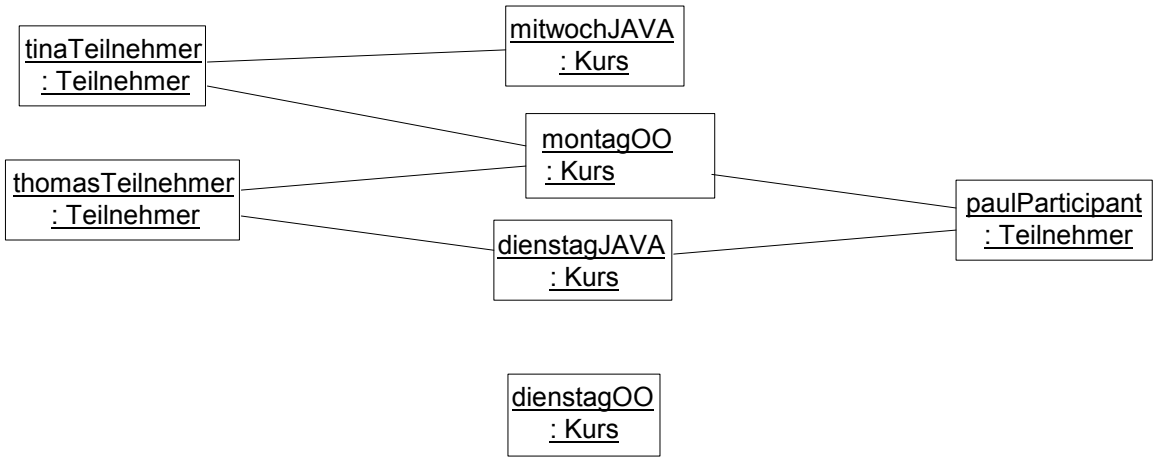
- Beispiele für Objektbeziehungen:
  - dozentDieter – dietersKonto, dozentPaul – paulsKonto



  - typOOKurs – montagOO, dienstagOO, typJAVA – dienstagJAVA



  - Teilnehmer –Kurse

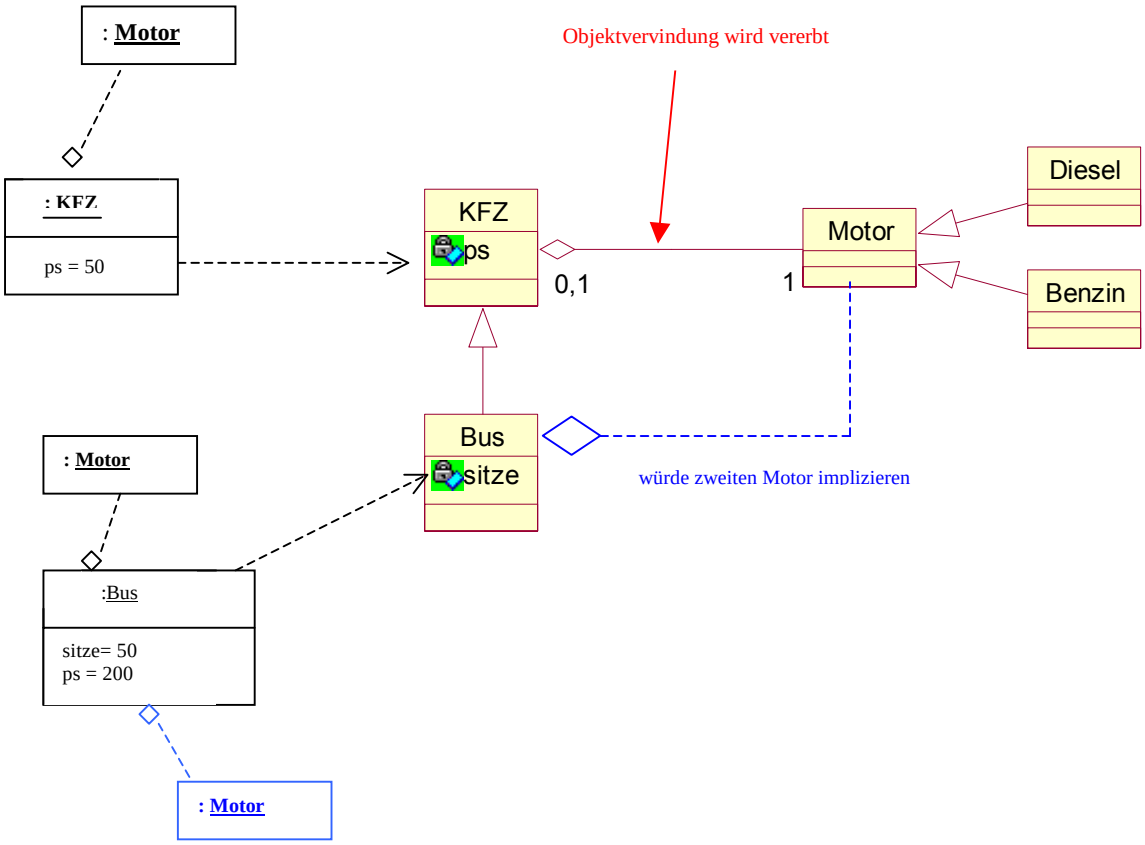


Identifizieren von Strukturen

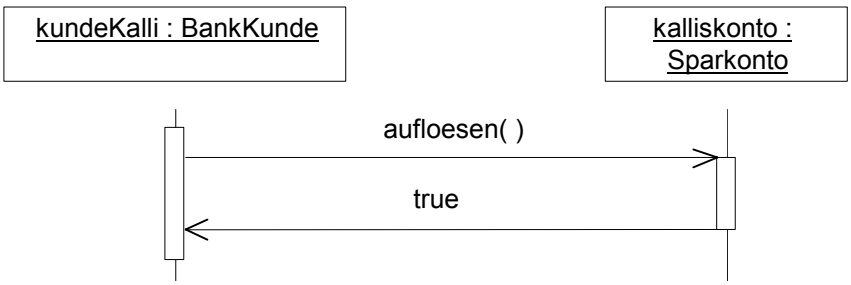
Vererbungstrukturen

- Vererbung (Generalisierung, von 2 oder mehr Klassen zu einer Basisklasse)
- Objekte der abgeleiteten Klasse erben alle Attribute, Objektverbindungen und Methoden der Basisklasse

- auch Objektverbindungen werden vererbt (zusätzlich zu Methoden und Attributen)

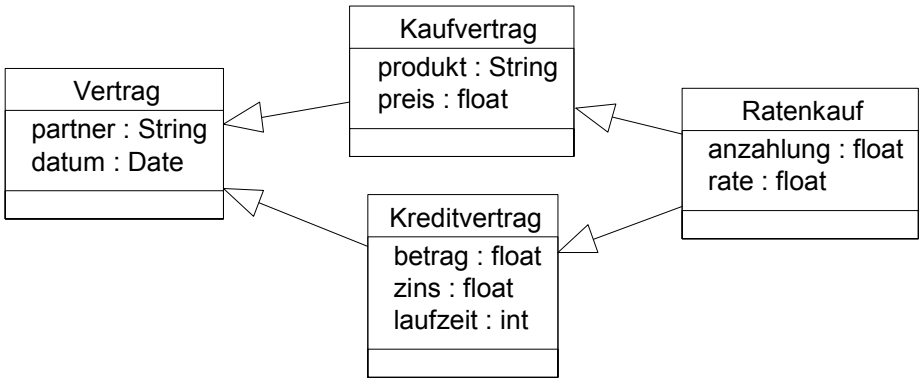


- statt Basisklasse kann immer das Objekt der abgeleiteten Klasse verwendet werden
- (Umkehrung gilt nicht)
- Überall kann statt eines Objektes der Basisklasse auch eines der abgeleiteten Klasse verwendet werden. (Die Umkehrung gilt nicht!)
- Alle Nachrichten an Basisklassenobjekte können auch von Objekten einer abgeleiteten Klasse bearbeitet werden: da jedes *Konto* die Nachricht `gibStand()` verarbeiten kann, kann auch ein davon abgeleitetes *Sparkonto* diese Nachricht verarbeiten.
- Ein abgeleitetes Objekt kann die Objektverbindungen der Basisklasse nutzen, um selbst Nachrichten zu versenden: da jeder *Bus* einen Link zu seinem Motor hat, kann auch ein davon abgeleiteter *Linienbus* diesem die Nachricht *beschleunige()* senden.
- Beispiel:



Mehrfachvererbung

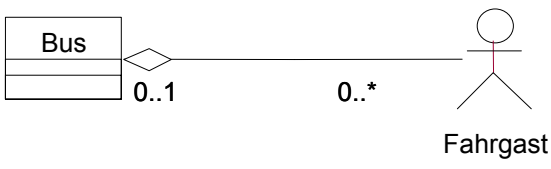
- Eine abgeleitete Klasse erbt von mehreren direkten Basisklassen
- Problem: Verschiedene Basisklassen haben gemeinsame Basisklasse, Doppelerbschaft
- Problem der Mehrfachvererbung bei Attributen beachten
- Beispiel:



Aggregationsstruktur

- Ein Objekt enthält ein anderes Objekt oder mehrere andere Objekte
- Typen: Gesamtheit-Teil, Container-Inhalt, Gruppe-Mitglied, Verwendung
- Beispiele:
  - Gesamtheit-Teil

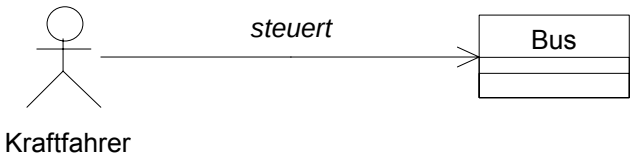
  - Container-Inhalt



  - Gruppe-Mitglied

Assoziationsstruktur

- Prüffrage: In welche Teile kann die Klasse im Problembereich zerlegt werden?
- Prüffrage: Werden die Komponenten innerhalb der Systemverantwortlichkeit benötigt?
- Prüffrage: Erfüllen diese die Kriterien an Klassen, die in das Modell gehören?
- Prüffrage: Haben die Teile keine ihrer Aufgaben an die Gesamtheit abgetreten?
- Beispiel:



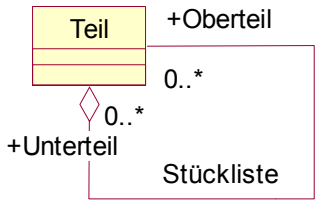
Identifizieren von Assoziationen als Aggregationen

Prüffragen:

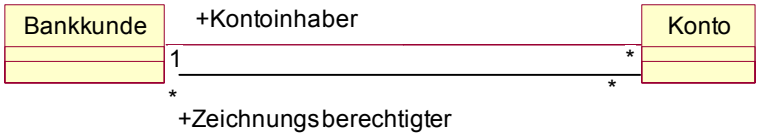
a) Welche dauerhaften Abhängigkeiten und permanente Beziehungen existieren zwischen den Objekten der beteiligten Klassen?

b) Welche Rollen spielen die beteiligten Klassen?
Ein Rollenname sollte insbesondere dann gegeben werden, wenn

- die Assoziation Objekte derselben Klasse verbindet



- Eine Klasse mit einer anderen mehrere Assoziationen hat.

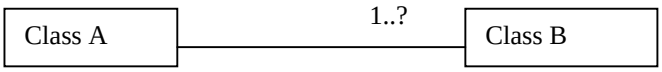


c) Ist die Assoziation irrelevant oder handelt es sich um implementierungstechnische Assoziationen?

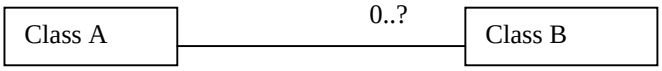
d) Handelt es sich um abgeleitete Assoziationen?
Klassen, Attribute und Assoziationen in einem Klassenmodell sollten soweit als möglich unabhängige Informationen darstellen und keine abgeleiteten.

Fragen zu Kardinalitäten

a) Liegt eine Muss – oder Kann-Beziehung vor?
Eine Muss – Beziehung besteht, wenn gilt: sobald das Objekt A erzeugt ist, muss auch die Beziehung zum Objekt B aufgebaut werden.



Bei einer Kann – Beziehung kann die Beziehung zu einem beliebigen Zeitpunkt nach Erzeugen des Objekts aufgebaut werden.



b) Ist die Obergrenze fest oder variabel ?
c) Gelten besondere Bedingungen?
Beispielsweise soll eine Assoziationsmenge in eine bestimmte Reihenfolge geordnet werden: {ordered}; oder es soll eine Objektverbindung, die einmal eingerichtet wurde, nicht mehr verändert werden:{immutable}.

Das Konto ist nicht übertragbar.

Identifizieren von Methoden

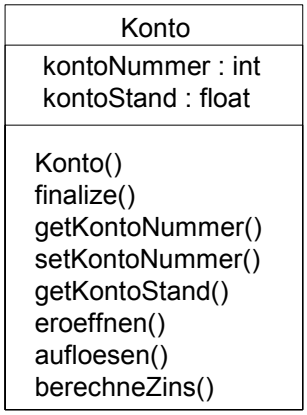
Konzept

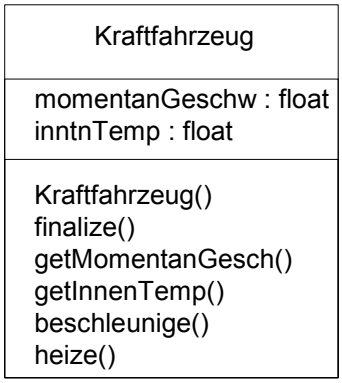
- Sämtliche Aktivitäten eines objektorientierten Systems basieren auf der Kommunikation zwischen Objekten
- Ein Objekt („Server“) stellt Dienste zur Verfügung, auf die andere Objekte („Clients“) zugreifen können.
- Ein Objekt fordert als Client die Dienste eines anderen Objektes als Server mittels Aussenden einer Nachricht an.
- Details über Methoden, Nachrichtenverbindungen und Reihenfolgen werden erst später festgelegt.

- Eine Nachricht kann auch als „Broadcast“ an alle Objekte einer Klasse gesendet werden
  Diese Art von Nachrichten unterstützen Java und C++ nicht. Diese können jedoch simuliert werden. Dazu wird in der Klasse ein Klassenattribut impliziert, das ist eine Liste aller Objekte, die angelegt werden.
- Notwendig für Kommunikation: Objektbeziehung oder Gesamtheit-Teil-Verhältnis
- Allgemein kennt der Sender den Empfänger, nicht aber der Empfänger den Sender
- Nachrichtenversendung an sich selbst ist möglich und üblich: der *Bus*, der die Nachricht *anhalten()* empfängt, wird sich selbst die Nachricht *bremsen(0)* senden

Arten von Methoden

- Unterscheide implizite und explizite Methoden
- Implizite Methoden in der Analysephase nicht dargestellt, damit überschaubar
- Explizite Methoden stellen die echte Funktionalität dar, berechnen Werte
- In Klassen die Realwelt-Schnittstellen darstellen werden Methoden definiert die auf Eingaben oder Nachrichten externer Systeme warten





Implizite Methoden

- Konstruktoren: Erzeugung von Objekten
- Destruktoren: Zerstörung von Objekten
- Zugriffsfunktionen: *setAttribut*(), *getAttribut*, *isAttribut*(), *hasAttribut*()
- Verbindungsfunktionen

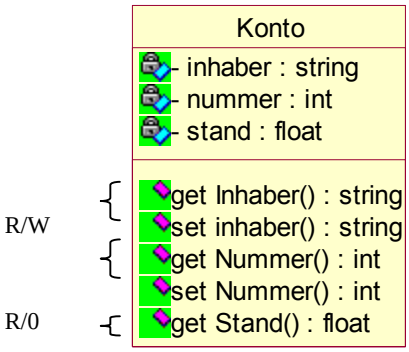
Zusammenfassung implizite Methoden:

\+ *Konstruktor*, *(Destruktor* – jedoch nicht in Java)

Objekt-Bau-Methode Objekt-Verschrottungs-Methode

+ Zugriffsmethode für Attribute

(„Getter“ , „Setter“ - Methoden)



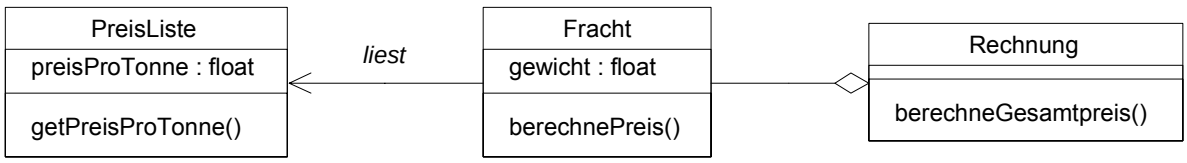
R/W= read and write
R = only read

+ Objektverbindungen (herstellen und abbauen)

Implizierte Methoden werden in der Regel nicht dargestellt.

Explizite Methoden

- Beschreiben spezifische Verantwortlichkeiten einer Klasse
- Dienste, die ein Objekt dieser Klasse anbietet und zur Verfügung stellt
- Verantwortlichkeiten, die eine Klasse erfüllen muss
- Explizite Methoden werden modelliert und dargestellt.
- Prüffragen: Welche Berechnungen sollen die Objekte bereitstellen?
  Gibt es Überwachungsaufgaben, die wahrgenommen werden müssen?



Spezifikation von Methoden

- Spezifikation einer Methode wird beim Empfänger bzw. Server vorgenommen

- Spezifikation:

  Name der Methode

  Typen und Namen der Argumente

  Typ des Wertes

  Beschreibung

- Diese beiden Spezifikationen treten in der Regel als Paar auf:

Methodenspezifikation	Spezifikation von Nachrichtenverbindungen
Was kann das Objekt (Dienstangebot)?	Was soll anderes Objekt machen (Dienstanforderung)
Objekt selbst	Objekt anderer Klasse

- Beispiel:

```
Methoden der Klasse Fracht

Methode berechnePreis()
Argumente: keine
Rückgabewert: Fließkommazahl (float)
Beschreibung: Sendet die Nachricht getPreisProTonne()
an das Objekt vom Typ PreisListe. Berechnet den Preis als
Produkt von gewicht und dem erhaltenen PreisProTonne und
gibt diesen Wert zurück.

Methode ...
```

Identifizieren von Nachrichtenverbindungen

- Nachrichtenverbindung setzt einen Kommunikationspfad (Link) voraus
- Jede Verbindung (Pfeil vom Sender zum Empfänger) repräsentiert:

  angeforderte Methode, Argumente (Werte oder Referenzen), Resultat (Rückgabepfeil)

- Für einen Link können mehrere verschiedene Nachrichten angegeben werden
- Sequenzen von Nachrichten werden durch Nummern geordnet
- Spezifikation wird in der Klasse des Senders bzw. Clients vorgenommen
- Spezifikation:

  Name

  benötigte Methode

  Typen und Namen der Argumente

  Typ des Wertes

  Beschreibung

- Beispiel:

```
Nachrichtenenverbindungen der Klasse Fracht

Verbindung zur Klasse PreisListe
Benötigte Methode getPreisProTonne()
Argumente: keine
Rückgabewert: Fließkommazahl (float)
Beschreibung: Die Anforderung eines Fracht-Objektes an das
PreisListe–Objekt fordert den Tonnagepreis an, um daraus den
Preis für dieses Fracht-Objekt zu berechnen.

Verbindung zur Klasse...
```

- Um zu verhindern, dass unnötige (weil ungenutzte) Methoden realisiert werden, ist es möglich und sinnvoll, nur diejenigen Methoden anzubieten und zu spezifizieren, die in mindestens einer Nachrichtenverbindung vorkommen. Dazu zählen auch Nachrichtenverbindungen zu sich selbst.

Klassenspezifikation

- Klasse: Name, Spezialisierungen, Generalisierungen, Beschreibung
- Teile: Namen, Kardinalitäten, Strukturtypen, Beschreibungen
- Attribute: Namen, Datentypen, Beschreibungen
- Objektverbindungen: Beteiligte Objekte, Kardinalitäten, Beschreibungen
- Methoden: Name, Argumente (Namen/Typen), Werte, Beschreibungen
- Nachrichtenverbindungen: Beteiligte Objekte, Methoden, Argumente (Namen/Typen), Werte, Beschreibungen

Übung 20: Kaffeeautomat, Teil 3 – Klassen- und Objektdiagramme für UseCase Getränkebeschaffung

1) Identifikation von Objekten und später Klassen
2) Verantwortlichkeiten
3) Beziehungen
4) Methoden und Attribute

Tipp: Klassennamen typischerweise Einzahl

Dynamisches Modell des Systems

Allgemeines

- Modelliert das Verhalten der Objekte
- Veränderungen der Attributwerte und Beziehungen im Zeitablauf
- Zustandsdiagramme modellieren Lebenszyklen von Objekten
- Zustandsübergänge sind immer Reaktionen auf Nachrichten

Ereignisse

- Ereignis ist Vorfall, der Attributwerte verändert und Zustandsänderung bewirken kann
- Ereignis ist immer mit Eintreffen einer Nachricht verbunden
- Ereignis ist immer mit einem Methodenaufruf verknüpft
- Externe Ereignisse treten auf bei Schnittstellenobjekt oder von externem System

  z.B. Taste wurde gedrückt, Timer-Objekt hat externes Ereignis erzeugt
- Interne Ereignisse treten auf bei Nachrichtenaustausch innerhalb des Systems

  z.B. Erzeugen und Löschen von Objekten

Zustände

- Zustand eines Objektes wird bestimmt durch seine Attributwerte
- Zustand = Zusammenfassung äquivalenter Attributwerte
- Zustände, die ein Objekt annehmen kann, werden in Zustandsspezifikation beschrieben
- Objekte ändern meistens ihren Zustand, um ihre Aufgaben erfüllen zu können

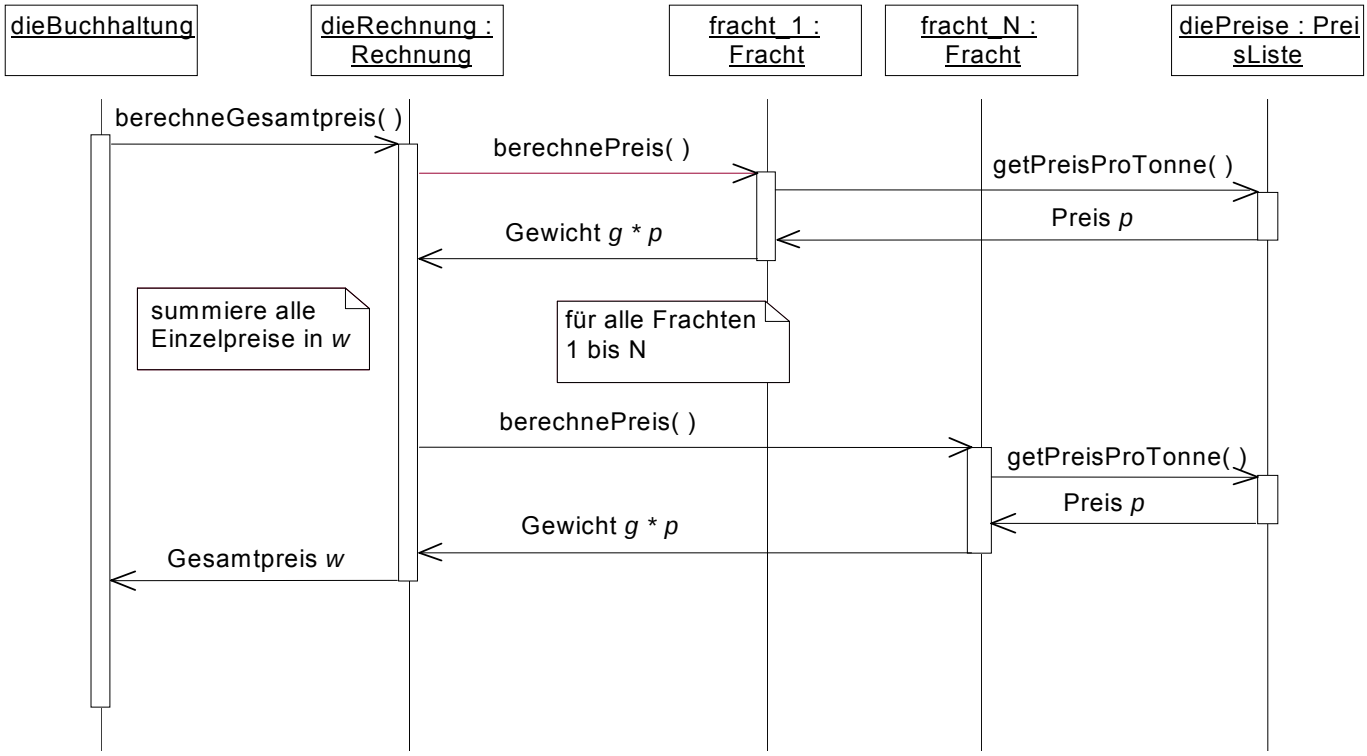
Szenarios

- Hypothetische Aufeinanderfolge von Ereignissen zur Darstellung der kausalen Zusammenhänge
- Jedes vorstellbare Ereignis, das eintreten kann, in mindestens einem Szenario erfasst
- Szenarios können als Skript oder Freitext formuliert werden.

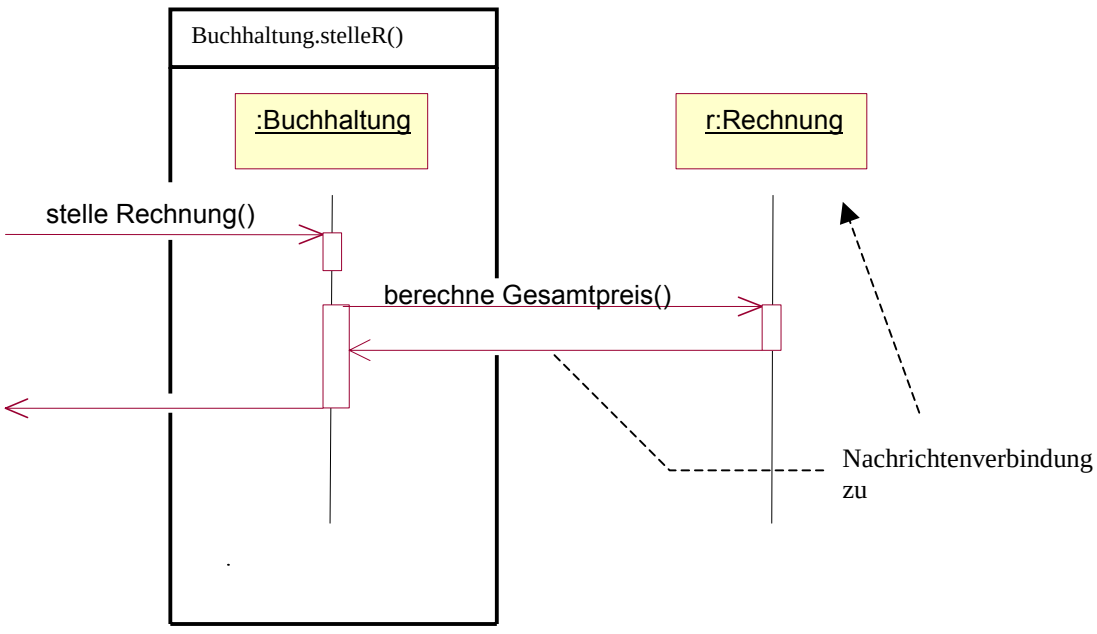
Ereignisfolgen

- Formalere und detailliertere Darstellung der Objektinteraktionen aus den Szenarios
- Erleichtern die Validierung der modellierten Struktur- und Verhaltensaspekte
- Pro Szenario ein Ereignisfolgediagramm
- Objekte durch vertikale Linien, Nachrichten durch Pfeile vom Sender zum Empfänger

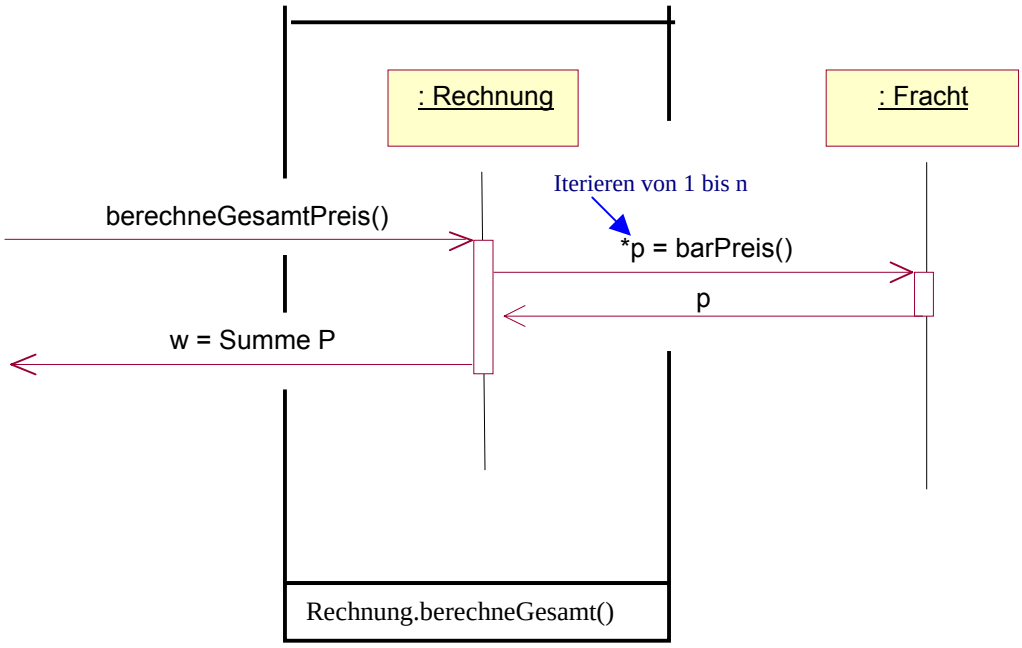
❖ Beispiel: Vollständiges, tiefes Sequenzdiagramm für die Methode `berechneGesamtpreis()` der Klasse *Rechnung*



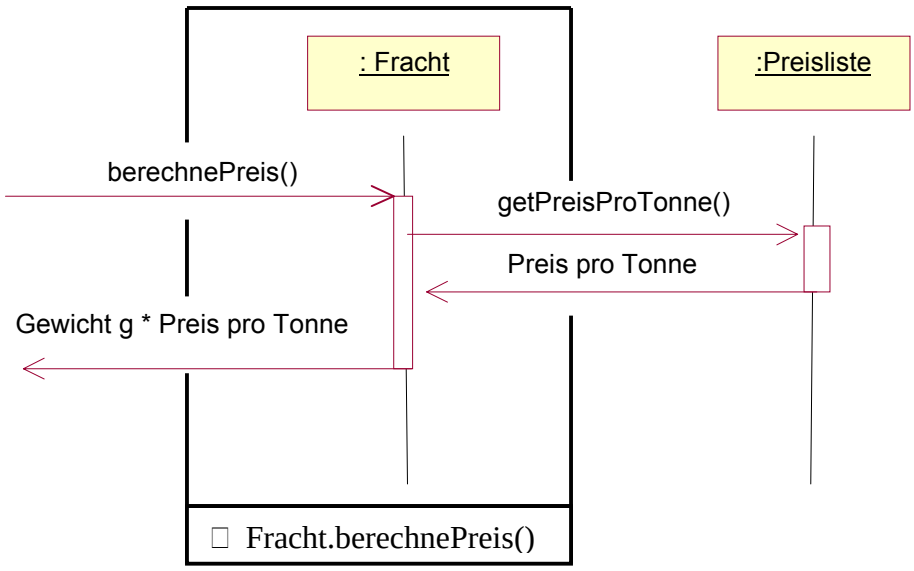
❖ Besser: Pro Methode kann ein Sequenzdiagramm erstellt werde:



❖ Die Behandlung einer Nachricht bei untergeordneten Objekten wird separat beschrieben:

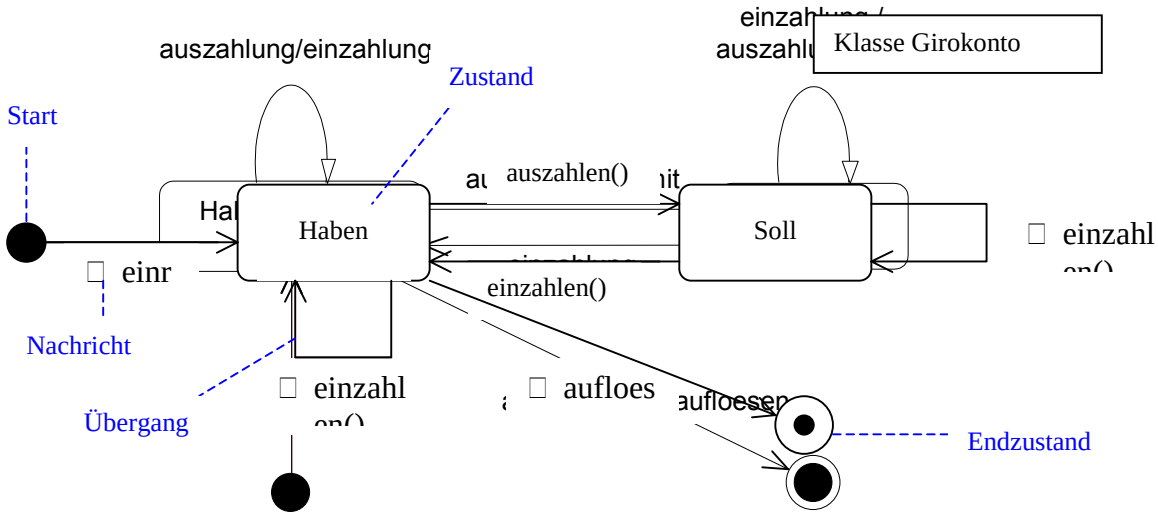


❖ Typischerweise pro Methode eine/Nachricht jeder Klasse ein Sequenzdiagramm (max.).



Zustandsdiagramme

- Darstellung von Ereignissen, Zuständen, Aktivitäten und ihre Aufeinanderfolge für eine Klasse, für ein Teilsystem oder das ganze System
- Jeder Weg durch das Diagramm beschreibt ein Verhaltensmuster für ein Objekt
- Jeder Weg durch das Diagramm beschreibt ein Szenario/Sequenzdiagramm
- Diagramm beschreibt den Lebenszyklus eines Objektes
- Dem Verhalten in einem Szenario oder Ereignisfolgediagramm entspricht genau ein Weg durch das Zustandsdiagramm
- Pro Klasse (maximal) ein Zustandsdiagramm
- Pro Zustandsdiagramm (genau) eine Klasse
- Zustände können in „Unterzustände“ zerlegt werden, z.B. Haben zerlegen in HabenBis2000 und HabenAb2000
- Zustände können nicht überlappen!
- Ein Objekt kann sich niemals in 2 Zuständen gleichzeitig befinden (disjunkt).
- Zustandsdiagramme entstehen aufbauend auf die Ereignisfolgediagramme
- Beispiel Zustandsdiagramm



Funktionales Modell des Systems

- Algorithmische Beschreibung der identifizierten Methoden und Aktivitäten
- Entwurf erfolgt auf Klassenebene

- Hilfsmittel zur Spezifikation:
  Strukturierte Sprachen, Entscheidungsbäume und –tabellen, Ablaufpläne, Struktogramme, Pseudocode, mathematische Gleichungen

Übung 21: *Kaffeeautomat, Teil 4*

Erstelle ein das Klassenstrukturdiagramm für Mixer, Rezeptliste, Zutaten

Übung 22: *Kaffeeautomat, Teil 5*

Erstelle ein Kollaborationsdiagramm für den Test, ob ein bestimmtes Getränk möglich ist

Übung 23: *Kaffeeautomat, Teil 6*

Erstelle das dazugehörige Objektdiagramm

Übung 24: *Kaffeeautomat - Teil 7 : Modul Geld*

Entwirf ein Kollabroationsdiagramm für das Modul Geld!

Übung 25: *Kaffeeautomat - Teil 8 : Modul Geld*

Entwirf das Klassenstrukturdiagramm für das Modul Geld!

Übung 26: *Entwicklung einer Basisklasse Liste (+Schnittstelle) für das Konzept*

Generalisiere Rezeptliste, Zutatenbehälter zu einer gemeinsamen Basisklasse Liste

Tipp : Verwende eine Schnittstelle

Übung 27:

Entwirf ein Kollaborationsdiagramm für die Nutzung dieser Klasse

Übung 28:

Stelle das Klassendiagramm dazu auf.

Objektorientiertes Design

- Nahtloser Übergang
- OOA: Modell zur Beschreibung der Systemverantwortlichkeiten
- OOD: Architekturmodell des zu implementierenden Systems
- Allgemeine taktische Vorgehensweise

Vorgehensweise

- Modellierung spezifischer Implementierung
  unter Berücksichtigung von Hard- und Softwareumgebung

Beispiel: Berücksichtigung von Soft- und Hardwareumgebung

- Zielsprache: JAVA – Interface
  C++ - Mehrfachvererbung
- Zielplattform: verschiedene GUIs*
  (Unix, Windows, Mac)
  *Graphical User Interface
- Verwendete Bibliotheken:

  C++: "MVSC++"

  MFC - Microsoft Foundation Classes (MS)
  OWL - Object Windows Libary (Borland)

  JAVA: "Borland Jbuilder" – **plattformunabhängig!**

  JAVA- AWT : Abstract Windowing Toolkit (Sun)
  JFC - JAVA: Foundation Classes (SUN) → "Swing Classes"
  JBCL - JAVA: Borland Custom Control Library ("Borland Controls für JAVA")
  AFC - Application Foundation Class (MS; Win 95 Controls für JAVA)

  Nur "Win 32"! – **plattformabhängig!**

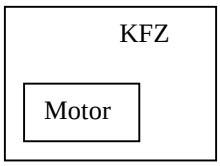
  WFC: Windows Foundation Classes (Win 32 – Controls)

- Statisches Modell wird erweitert um
  lösungsspezifische Attribute, Objektverbindungen, Methoden oder Klassen

Beispiel: Attribute für Objektbeziehungen

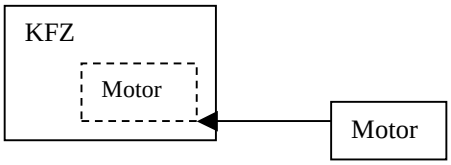
JAVA / C++:

1. enthalten als Element



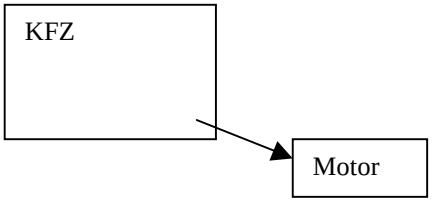
Komposition

2. enthalten als Referenz



Aggregation

Nur C++
3. Zugriff über Zeiger



Assoziation

- Iteration zwischen Problembereich (OOA) und Lösungsbereich (OOD)

Prüffragen:

- Welche Attribute sind nötig um Objektverbindungen zu implementieren?
- Welche Attribute sind nötig um Gesamtheit-Teil-Beziehungen zu implementieren?
- Sollen zur Effizienzsteigerung ableitbare Attribute modelliert werden?
- Welche zusätzlichen Klassen sind notwendig: Hilfsklassen, Benutzerschnittstelle?
- Sind persistente Objekte gefordert? Wenn ja, für welche Klassen?
- Findet Kommunikation nur synchron oder auch asynchron statt?
- Asynchrone Kommunikation ist aufwendig, also nur dort implementieren, wo notwendig
- Ist das System zentral oder verteilt zu realisieren?

  Wie werden im verteilten Fall Nachrichten weitergeleitet?

  Welche Architektur ist geeignet?

  Plattformunabhängigkeit: RMI, Beans

Sprachenunabhängigkeit: COM, DCOM, ActiveX

Plattform- und Sprachenunabhängigkeit: RPC, DCE, CORBA

Exkurs Verteilte und Komponententechniken
RMI – Remote Method Invocation

"Aufruf von Methoden für Objekte auf anderen (entfernten Rechnern)"

JAVA:

1) Rechner A (Server) erzeugt ein Objekt und exportiert es/stellt es zur Verfügung
2) Rechner B (Client) fragt beim Server an, erhält eines Referenz auf dieses Objekt und kann nur dessen Dienste nutzen.
   (JAVA) Beans – JAVA – Komponententechnologie:
   Softwarekomponenten /- bausteinen, die **ohne Neukompilieren** verwendbar sind.
3) Active X / COM : Component Object Model
   (~MS / Windows Pendant zu JAVA Beans)
4) DXOM – Distributed COM (~MS/ Win RMI)
5) CORBA - Common Object Request Broker Architecture (~ RMI / DCOM also plattform- und sprachunabhängig)

❖ Fehlermöglichkeiten, Ausnahmebehandlungen insbesondere bei verteilten Systemen

Exkurs: Fehlermöglichkeiten /- behandlung

“ Ausnahmebehandlung” (Exception Handling):
1) Versuche, eine Aktion durchzuführen
2) Fehler tritt im Programm auf
3) System/Programm “wirft” daraufhin Ausnahmen
4) Spezieller Programmteil fängt sie auf

Bsp.: (JAVA)

```
1) try
   {
      methodenAufruf(); // hier kann etwas passieren
       }
3) catch (Exception e )
   {
      // Fehlerfall – Aktionen
   }

   methodenAufruf()
    {
1) if (Fehler)
2)  throw new Exception();
    }
```

Anwendung

Definition

```
int division ( int a, int b) throws Exception
{
  if (b= = 0)
    throw new Exception();
  else
    return a/b;
}
```

```
try
{
  division(30,3);  // geht ok
  division (2,0);  // ruft Ausnahme hervor
  division (10,4); // wird nicht erreicht!
}

catch
{
  system.out.println (“Division durch Null!”);
}
  system.out.println (“Ende!”);
```

Also: Ausnahmebehandlung = programmgesteuerte Fehlerbehandlung

- Datenschutz: Authentisierung, Autorisierung, Verschlüsselung

Authentisierung - Wer greift zu?
Autorisierung - Was ist erlaubt?
Verschlüsselung - Alle anderen haben keinen Zugriff.

- Datensicherheit: Redundanz, Transaktionsverfahren

Redundanz - “mehrfaches Speichern derselben Daten”
Transaktionen - Mehrere Aktionen sind nur als “Gesamtpaket” sinnvoll, Einzelschritte sonst rückgängig machen.

Ergebnisse der Designphase

Beschreibung der Architektur

- Klassen und Objektdiagramme der logischen Architektur

- Moduldiagramme der physikalischen Architektur
- Einteilung der Klassen in Klassenkategorien
- Einteilung der Module in Subsysteme

Beschreibung der gemeinsamen taktischen Vorgehensweise

- Fehlerbehandlung
- Speicherverwaltung
- Datenspeicherung
- Taskmanagement
- Transaktionskonzept
- Sicherheitskonzept

Entwicklungsplan

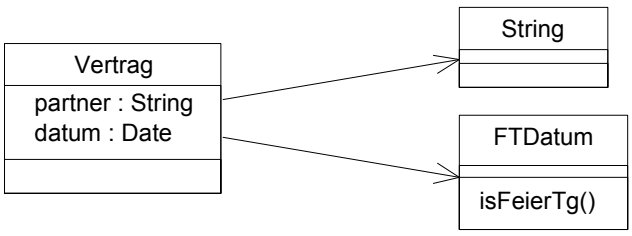
- Versionsplanung
- Einordnung der Szenarios und Funktionspunkte in Versionsfolge
- Aufgabenverteilung
- Risikoeinschätzung und Terminplanung
- Verifizierung der Architektur mit Hilfe von Prototypen

Elementare Systembausteine

PBK – Problembereichskomponente (Analyse)
MCK – Mensch-Computer-Komponente (Benutzerschnittstelle)
DMK – Datenmanagementkomponente (Persistenz)
TMK – Taskmanagementkomponente

Problembereichskomponente

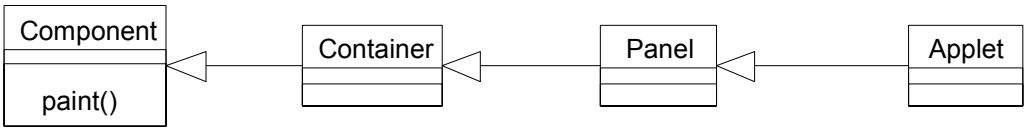
- Klassen und Strukturen gemäß der Analysephase
- Fortgeschrieben und ergänzt in der Designphase



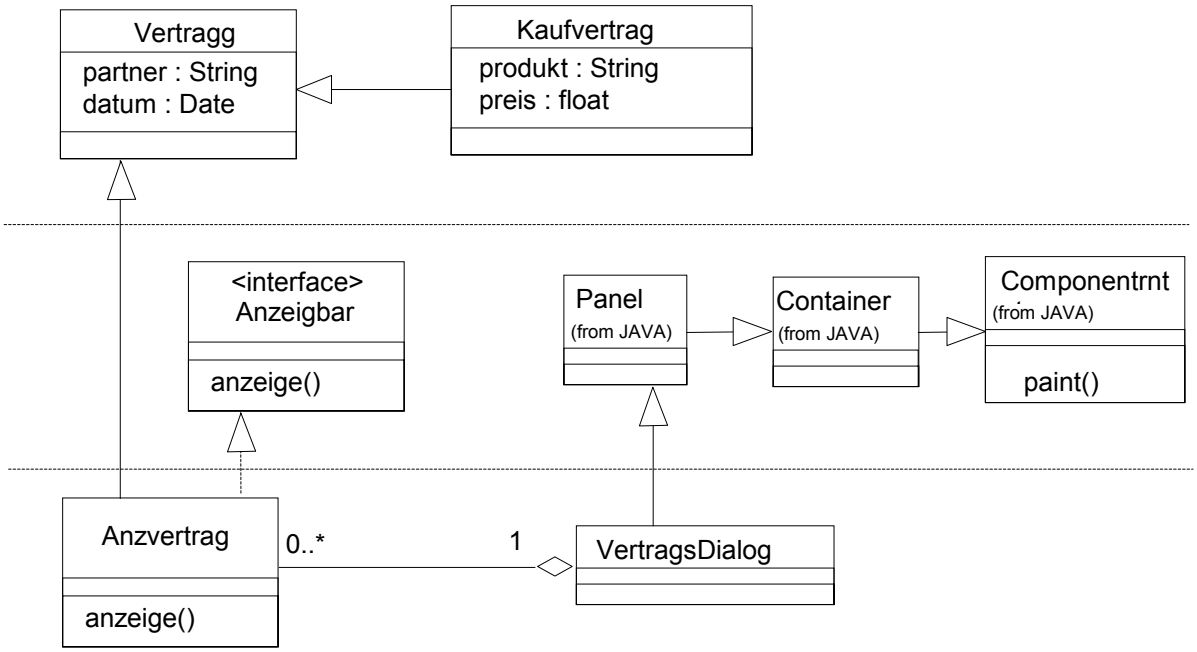
Kommunikationskomponente

- Bedienung des Systems durch den Benutzer
- Präsentation von Resultaten und Informationen

- Klassen und Objekte auf der Systemgrenze, stark werkzeug- und bibliotheksabhängig

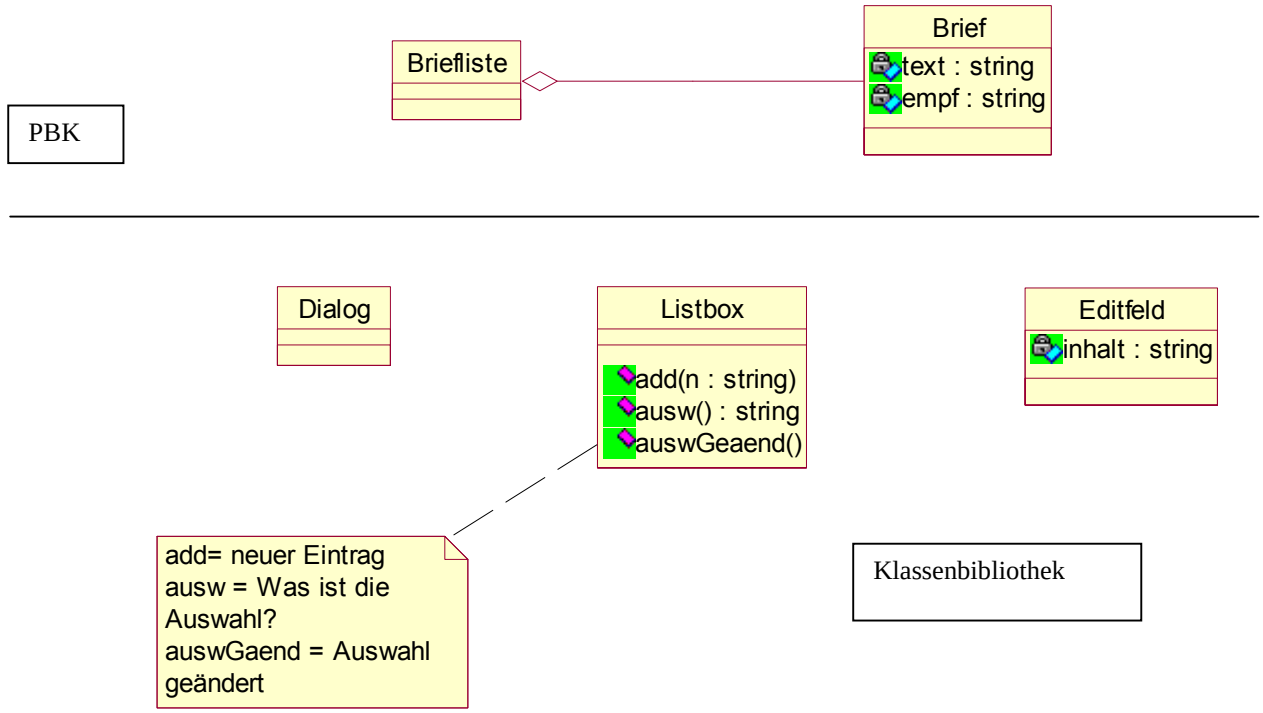


- eigenständiges, klar abgegrenztes Subsystem
- Beispiel:



Übung 29: *PBK/MCK*

Gegeben sind die folgenden Klassen:



Aufgabe:
Nutze diese Klassen, um folgende Bedienoberfläche zu gestalten:

1) Dialog mit einer Liste und einen Eingabefeld
2) Liste zeigt Empfaenger aller Briefe, Eingabefeld zeigt Brieftext zu ausgewählten Empfängern.
3) Änderung der Auswahl (in Liste) überträgt Notwendiges vom und ins Eingabefeld.

Welche Methoden werden in der PBK zusätzlich benötigt?
Welche zusätzlichen Klassen werden für die MCK benötigt?
Können Klassen unverändert genutzt werden?

Tipp: Erstelle Kooperationsdiagramme für die Nachrichten zur

- ✦ Initialisierung des Dialogs sowie beim
- ✦ Wechsel der Auswahl der Listbox.

Datenmanagementkomponente

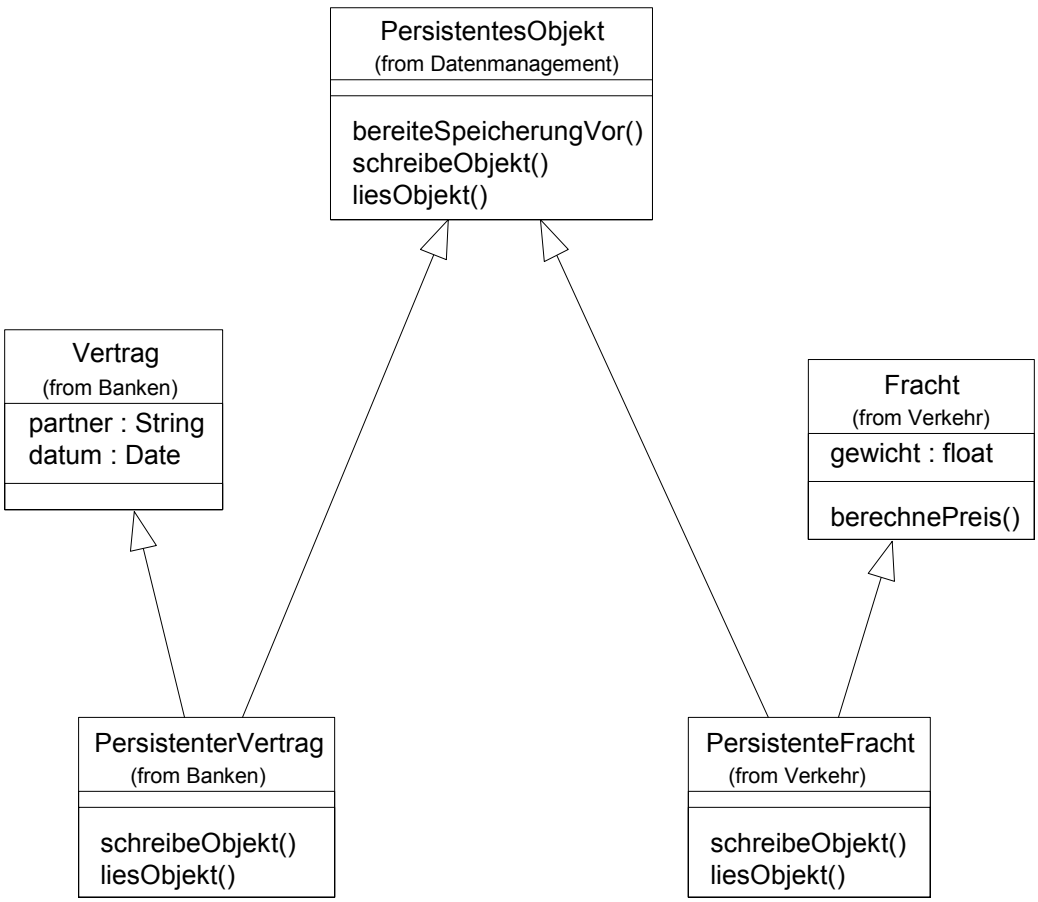
- ❖ Technik ähnlich wie bei Entwurf der MCK:

- Führe eine “Zwischenschicht” zwischen der PBK und der verwendeten Bibliothek ein.
- Datenbankschemata erstellen
- Datenaspekte, Voraussetzung für Speicherung und Wiederauffinden von Objekten
- Einfluss haben Integrität und Konsistenz, aber auch Zugriffsgeschwindigkeit
- Ebenfalls isoliert zu betrachtendes Subsystem
- Technik Textdatei:

  unstrukturiertes Speichern in Datei

  in der Regel proprietäres Format

  zusätzliche abstrakte Basisklasse oder Schnittstelle (JAVA) für „persistente Objekte“

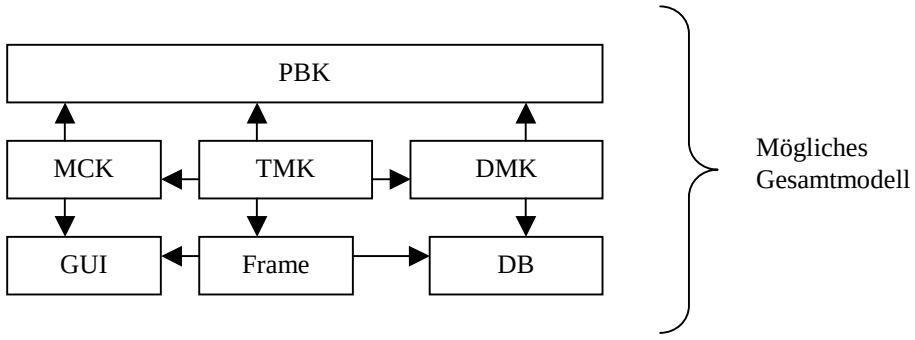


- Technik Relationale Datenbank:
  Speicherung in einer oder mehreren Tabellen
  Schutzmechanismen werden genutzt
  generische Klasse für Datenbankcollection
  Abbildung in die Relationen eines Datenbankschemas
  Verschiedene Verfahren:
  für jede Klasse eine Tabelle (ObjID, Attribut, ...)
  für jedes Attribut jeder Klasse eine Tabelle (ObjID, Attribut)
  insgesamt nur eine Tabelle (Klasse, ObjID, Attributname, Attribut)
- Technik Objektorientierte Datenbank:
  Nutzung der entsprechenden Erweiterung einer objektorientierten Programmiersprache
  Persistente Objekte unterscheiden sich nicht mehr von transienten Objekten

Taskmanagementkomponente

- Koordination der Problembereichsobjekte und ihrer Methoden
- Modellierung des nebenläufigen Verhaltens verschiedener Objekte sowie der asynchronen Objektkommunikation
- Klassen zum Starten und Beenden von Tasks und Prozessen
- Aktivitäten und Aktionen, Interprozesskommunikation und Prioritäten
- Ablaufsteuerung
- Meistens “vom System” erledigt oder durch Programmrahmen (“Framework”) sichergestellt.

Mögliches Gesamtsystem



Abkapselung von Systemteilen und Definition von Schnittstellen

- Modularisierung hat als Ziel kohäsive und lose gekoppelte Module
- Komplexität durch Modularisierung wesentlich verringert
- Module als Klassengruppen unter dem Aspekt der Wiederverwendbarkeit

Übung 30: *Kaffeeautomat - MCK für Teil Getränkeauswahl*

Entwirf die MCK für den Teil “ Getränkeauswahl”.
Verwende hierzu die bekammten Klassen Dialog, Listbox sowie die klasse Button .
Button kennt die Methode pressed, die er beim Drücken erhält.

Tipp: Kooperationsdiagramm für Initialisierung des Dialogs und Getränkeauswahl aus Liste mit Knopfdruck.

Ziel: Klassendiagramm der MCK

Frage: Welche zusätzlichen Klassen erhält die PBK?

Marschweg:
1. Kooperationsdiagramm init()
   (Spezialisierungen?, Zusätzliche Klassen?)
2. Kooperationsdiagramm pressed()
   (Spezialisierungen?, Neue Methoden?)
3. Klassendiagramm MCK
   (Aggregationen?, Assoziationen?)

Objektorientierte Programmentwicklung

Wartbarkeit

Formen von Wartung

- Fehlerbehebung und Mängelbeseitigung
- Weiterentwicklung und Erweiterung der Funktionalität

Bessere Wartbarkeit durch Objektorientierung

- Robustheit
- Erweiterbarkeit
- Wiederverwendbarkeit
- Abbildung ähnlich der realen Welt
- Verflechtung von Analyse und Design
- Kompatibilität

Wiederverwendbarkeit

Formen allgemeiner Wiederverwendbarkeit

- Code
- Komponenten aus Bibliothek
- Design-Muster
- Anforderungsspezifikationen

Formen objektorientierter Wiederverwendbarkeit

- Basisklassen
- Spezialisierte Klassen
- Mechanismen, sog. Design-Patterns
- Application-Frameworks
- Business-Objekte
- Dokumentkomponenten

Komponenten und Tools für objektorientierte Entwicklung

Klassenbibliotheken

- Problembezogen wiederverwendbare Klassen, z.B. Business-Objekte
- Anwendungsbezogen wiederverwendbare Klassen, z.B. Framework-Objekte
- Erweiterte Basisklassen, z.B. Grafikklassen, Bedienelemente
- Basisklassen, z.B. Datenstrukturen

Entwicklungs-Tools

- Standardisierte Notation, z.B. Unified Modelling Language (UML)
- Unterstützung des gesamten Entwicklungsprozesses (OOA – OOD – OOP)
- Grafisches Entwicklungssystem, textueller Editor mit Sprachenunterstützung
- Browser für Klassen- und Modulhierarchien
- Quellcodegenerator für die Zielsprache
- Reverse Engineering für Round-Trip-Entwicklung
- GUI-Builder, Compiler, Debugger
- Klassenbibliothekar für Verwaltung eigener und vorgefertigter Bibliotheken

Projektmanagement-Tools

- Konfigurationsmanagement mit Quelltext- und Versionskontrolle
- Projektübergreifende Informationen
- Projektplanung und –steuerung

Methodenüberblick

- Objektorientierte Analyse und objektorientiertes Design
  Peter Coad, Edward Yourdon, 1991
- Objektorientiertes Modellieren und Entwerfen
  Rumbaugh und andere, 1991
- Objektorientiertes Software-Engineering
  Jacobson und andere, 1992
- Objektorientierte Analyse und Design
  Grady Booch, 1994

Objektorientierte Analyse und objektorientiertes Design
Peter Coad, Edward Yourdon, 1991

Ziele

- Objektorientierte Analyse
- Objektorientiertes Design
- Top-down-Ansatz
- Integrierte Vorgehensweise durch objektorientierten Systementwurf
- Schließen der Lücke zwischen Analyse und Design

Objektorientierte Analyse

- Identifikation von Klassen und Objekten des abzubildenden Realweltausschnittes
- Identifikation von Strukturen
  - Generalisierung
  - Spezialisierung
  - Vererbungsprinzip
- Definition von Subjekten als Zusammenfassung von Objekten
- Attribute und Instanzenverbindungen definieren
  - Beachtung von Generalisierung und Spezialisierung
  - Attribute nur in der kleinsten gemeinsamen Oberklasse notieren
  - Instanzenverbindungen bedeuten, dass ein Objekt ein anders benötigt, um seine Aufgabe erfüllen zu können
- Definition von Methoden und Nachrichtenverbindungen
  - Erzeugen und Löschen eines Objektes
  - Lesen und Schreiben der Daten eines Objektes
  - Berechnungen

Objektorientiertes Design

- Problembereichskomponente

  Verfeinerung der Ergebnisse der Analyse,

  z.B. im Hinblick auf Speicherverwaltung und Wiederverwendbarkeit

- Kommunikationskomponente

  Schnittstelle zwischen DV-System und Anwender erarbeiten

- Taskmanagement-Komponente

  Berücksichtigung zeitkritischer Vorgänge

  Hardwareeinsatz

- Datenmanagement-Komponente

  Zugriff auf und Manipulation von Daten

  datenbankunabhängiger Entwurf

Objektorientiertes Modellieren und Entwerfen
Rumbaugh und andere, 1991

Modelle

- Objektmodell
- Dynamisches Modell
- Funktionalitätsmodell

Objektmodell

- Identifizieren von Klassen und Objekten
- Anfertigen eines Data Dictionary
- Identifizieren von Beziehungen und Aggregationen
- Identifizieren der Attribute von Objekten und Beziehungen
- Einführung von Vererbungshierarchien zur Vereinfachung
- Sicherstellung von Zugriffpfaden für gewöhnliche Anfragen
- Wiederholung der bisherigen Schritte und Verfeinerung des Modells
- Gruppierung von Klassen in Modulen

Dynamisches Modell

- Entwerfen von Szenarien typischer Interaktionsfolgen
- Identifizieren von Ereignissen
- Entwurf einer Ereignisfolge für jedes Szenario
- Erstellen von Zustandsdiagrammen für jede Objektklasse
- Zuordnen von Ereignissen, Zuständen und Zustandsübergängen

Funktionalitätsmodell

- Identifikation der Ein- und Ausgabedaten
- Entwerfen von Datenflußdiagrammen
- Spezifikation der benötigten Funktionen
- Identifizieren von Beschränkungen
- Spezifizieren von Optimierungskriterien

Objektorientiertes Software-Engineering
Jacobson und andere, 1992

Modelle

- Anforderungsmodell
- Analysemodell

Anforderungsmodell

- Funktionale Anforderungen
  - Statische Struktur und statisches Verhalten
  - Dynamisches Verhalten
- Darstellungsform
  - Anwendungsfälle
  - Schnittstellenbeschreibungen
  - Problemdomäne
- Inhalte
  - Akteure
  - Anwendungsfälle
  - Systemabgrenzungen
  - Domänen-Objekte und Beziehungen

Analysemodell

- Fachliche Systemstruktur
  - Statische Struktur und statisches Verhalten
- Darstellungsform
  - Anwendungsfälle
  - Schnittstellenbeschreibungen
  - Problemdomäne
- Inhalte
  - Entity-Objekte
  - Interface-Objekte
  - Control-Objekte
  - Beziehungen zwsichen diesen Objekten

Objektorientierte Analyse und Design
Grady Booch, 1994

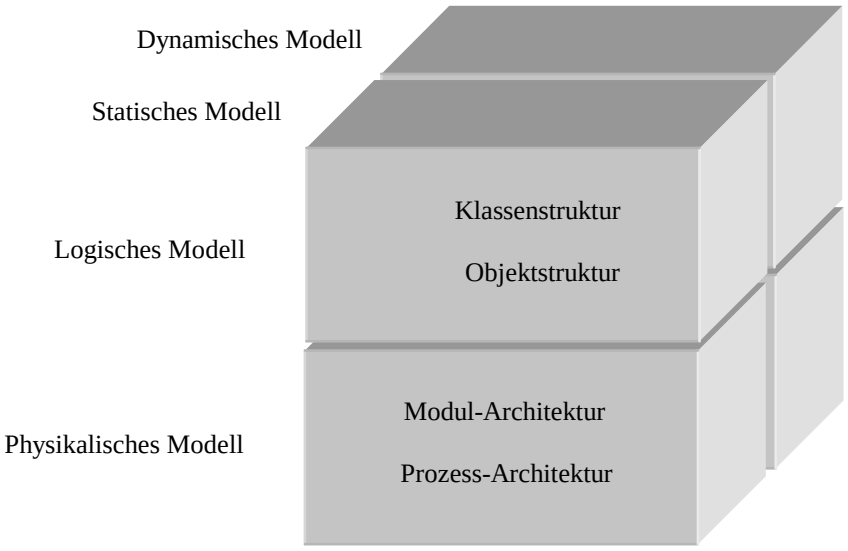
Macro Development Process

- Konzeptualisierung: Festlegen der Kernanforderungen
- Analyse: Entwicklung eines Modells für das gewünschte Sytemverhalten
- Design: Erzeugung einer Architektur für die Implementierung
- Evolution: Implementierung mit schrittweiser Verfeinerung
- Wartung: Verwalten der Entwicklung nach der Auslieferung

Micro Development Process

- Identifizieren von Klassen und Objekten einer bestimmten Abstraktionsebene
- Festlegen der Semantik dieser Klassen und Objekte
- Festlegen der Beziehungen zwischen diesen Klassen und Objekten
- Spezifizieren der Schnittstellen und Implementierung dieser Klassen und Objekte

Dreidimensionales Modell



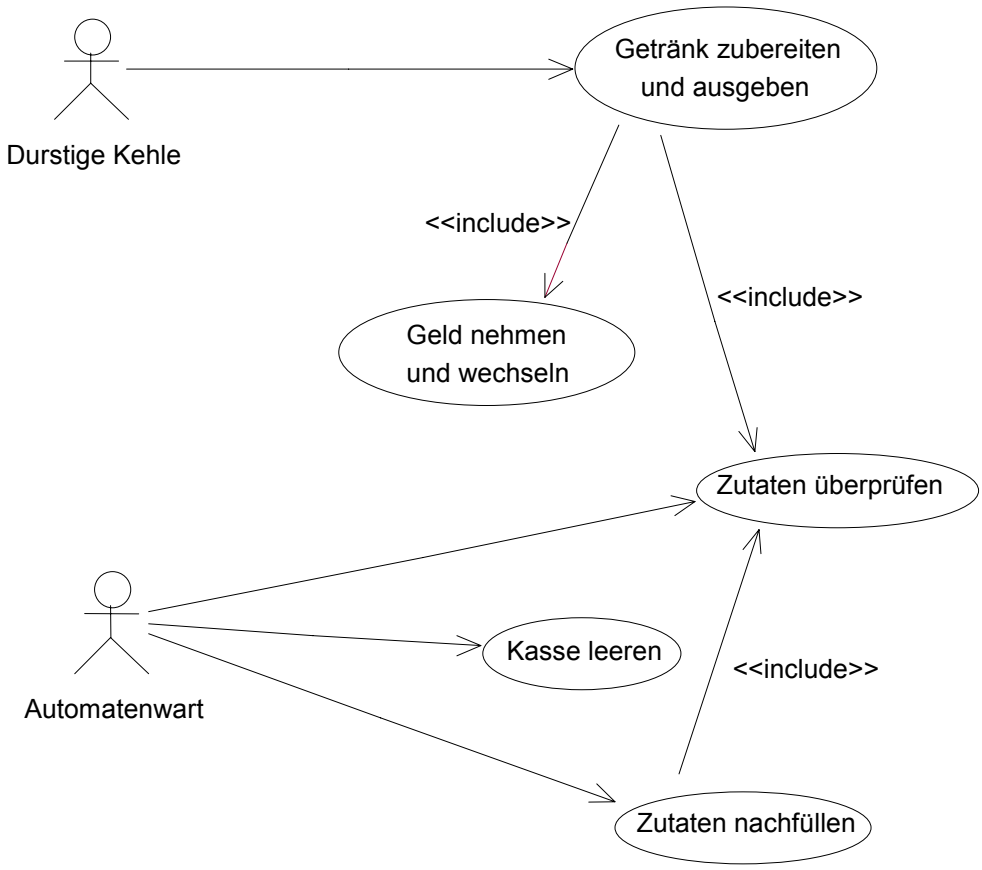
Beispiel: Getränkeautomat

Systembeschreibung

Das System beschreibt die Funktionsweise eines Getränkeautomaten für Warmgetränke (Kaffee und Kakao) einschließlich seiner Wartung (Nachfüllen von Zutaten)

- Münzeinwurf und – rückgabe
- Getränkeauswahl und – zubereitung
- Getränkeausgabe
- Leeren der Kasse und Auffüllen von Zutaten

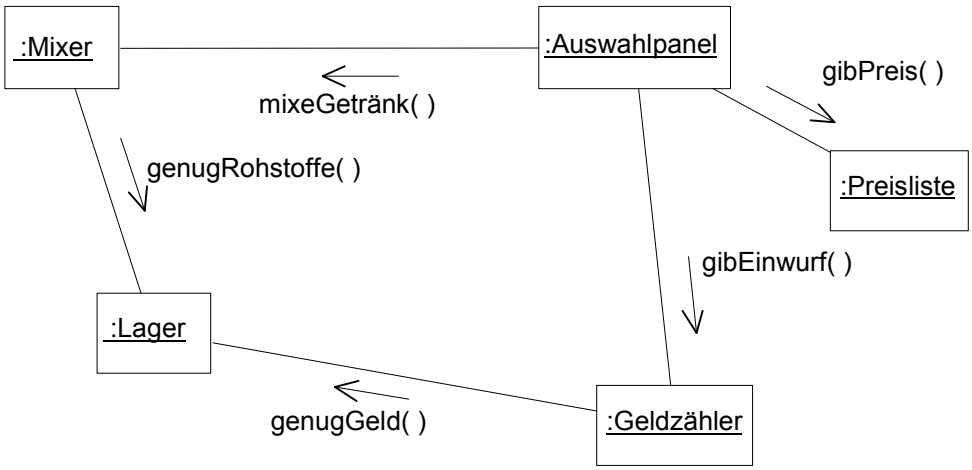
Anwendungsfalldiagramm



Kandidaten für Objekte, erster Ansatz

- Durstige Kehle, Automatenwart
- Rohstoffe, Becher, Wasser, Milch, Kaffepulver, Kakaopulver
- Geld, Groschen, Fuffziger, Markstücke
- Preisliste
- Automat, Kasse, Getränk
- Auswahlknöpfe
- Mixer, Lager, Geldzähler

Kollaborationsdiagramm



Kandidaten für Objekte, zweiter Ansatz

- Rohstofflager

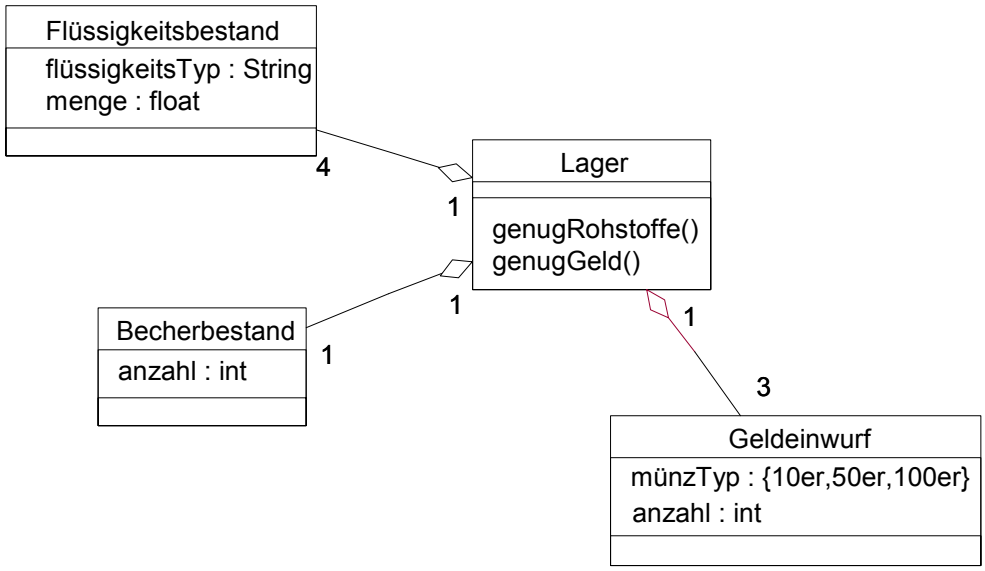
  Attribute: Becher, Wasser, Milch, Kaffepulver, Kakaopulver
- Geldzähler

  Attribute: Groschen, Fuffziger, Markstücke
- Preisliste
- Auswahlknöpfe
- Mixer

Entwurf der Klasse Lager

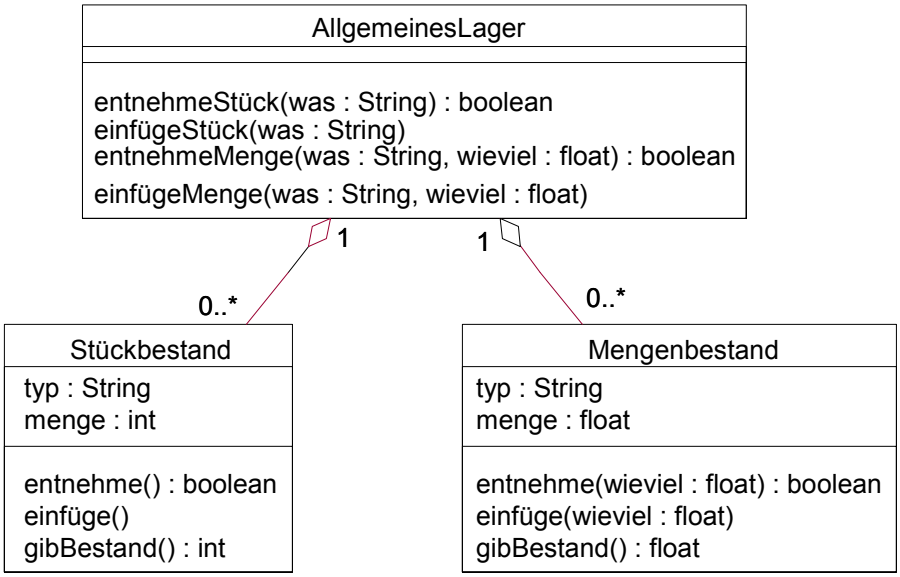
- Attribut: Becher, Ganzzahl, Zahl der vorhandenen Becher
- Attribut: Wasser, Fließkommazahl, Menge des vorhandenen Wassers
- Attribut: Milch, Fließkommazahl, Menge der vorhandenen Milch
- Attribut: Kaffepulver, Fließkommazahl, Menge des vorhandenen Kaffepulvers
- Attribut: Kakaopulver, Fließkommazahl, Menge des vorhandenen Kakaopulvers
- Methode: gibMengeZahl, Parameter: Typ, Rückgabe: Ganzzahl,
  liefert die Zahl der vorhandenen Becher
- Methode: gibMengeGramm, Parameter: Typ, Rückgabe: Fließkommazahl,
  liefert die vorhandene Menge der als „Typ“ gewählten Zutat

Klassendiagramm der Klasse Lager und ihrer Hilfsklassen

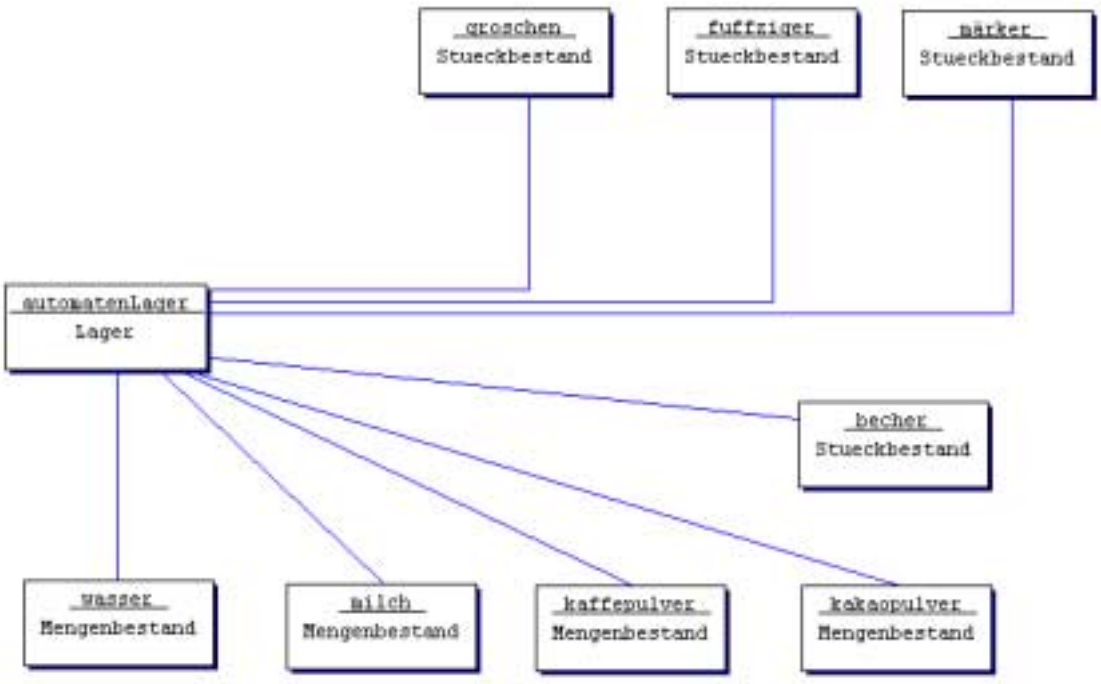


- Streng genommen unterscheiden sich *Becherbestand* und *Geldeinwurf* nur sehr wenig
- Führe daher für beide eine gemeinsame Basisklasse ein
- Modelliere diese sofort so, dass sie später auch für andere Anwendungen nutzbar ist
- Modelliere entsprechend eine Basisklasse für den *Flüssigkeitsbestand*

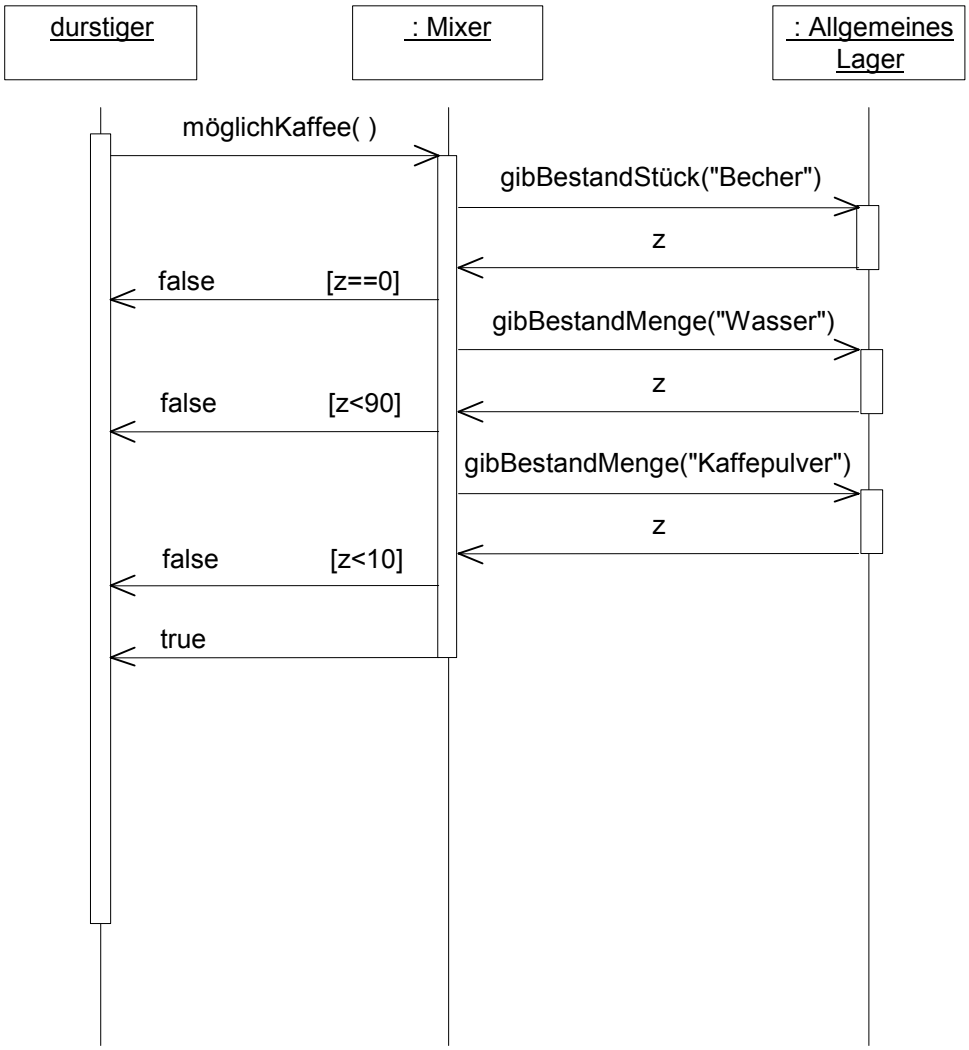
Klassendiagramm der allgemeinen Klasse Lager und ihrer Hilfsklassen



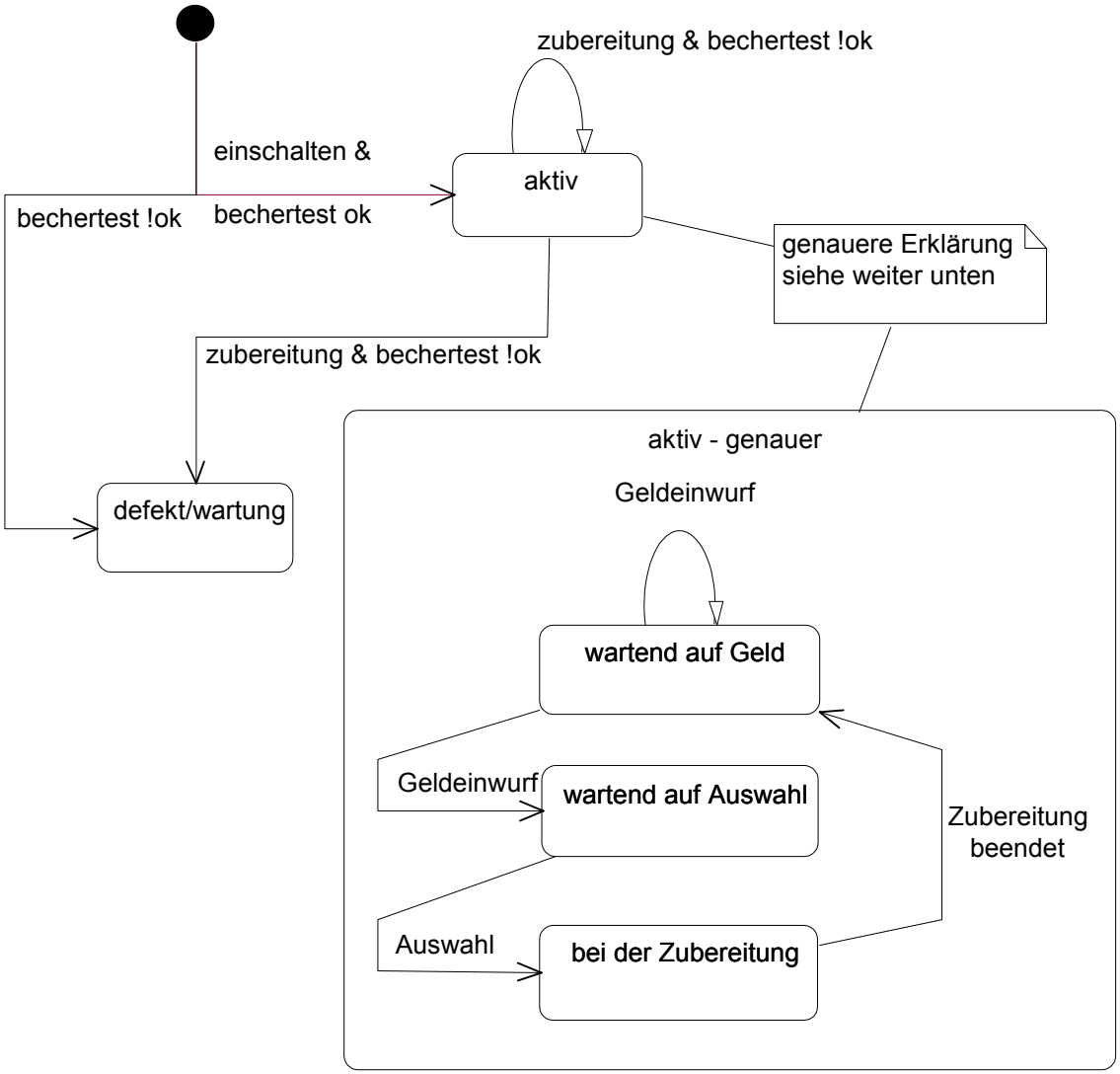
Objektdiagramm für das Lager des Getränkeautomaten



Sequenzdiagramm für die Nachricht* möglichKaffee() *an den Mixer

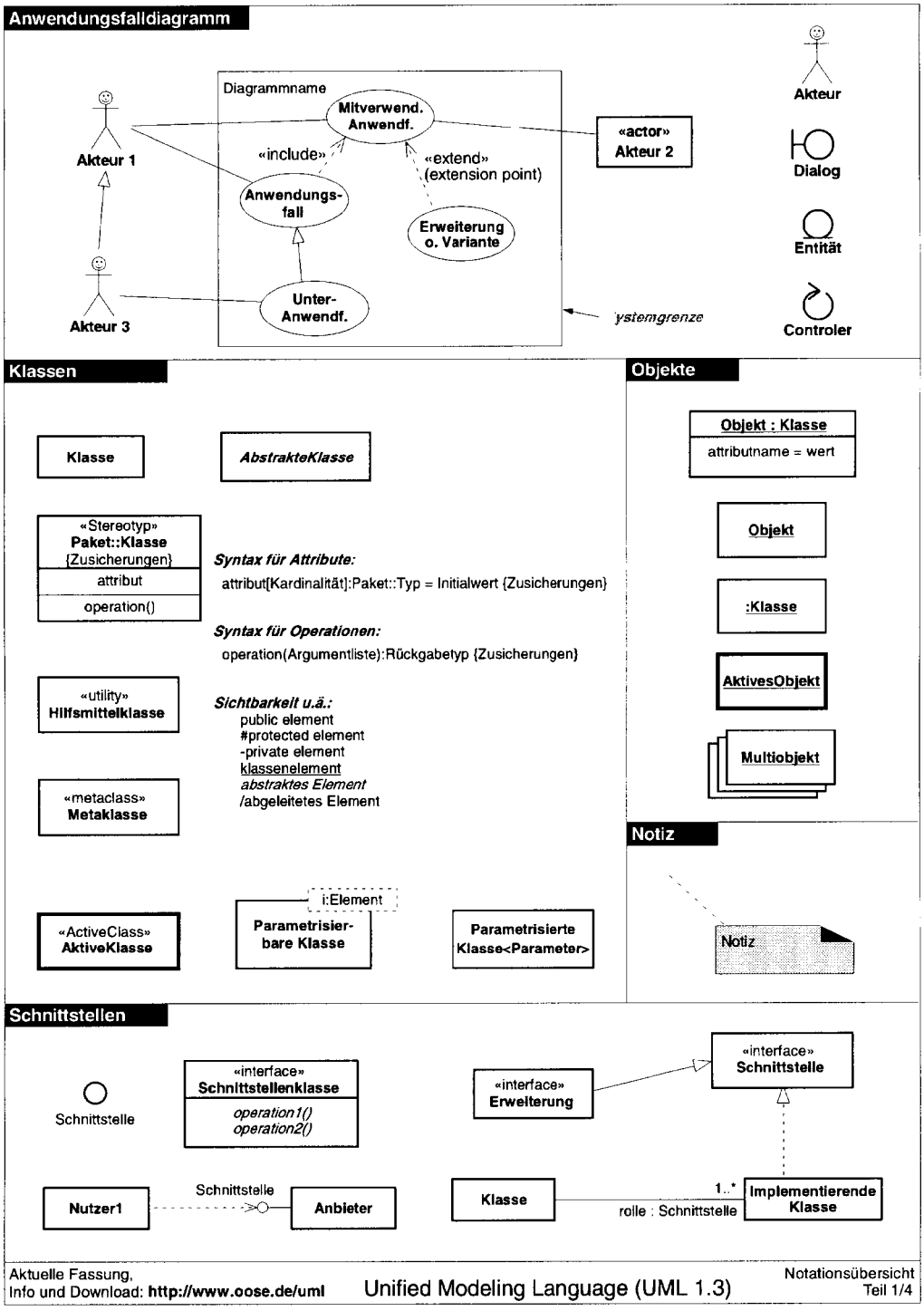


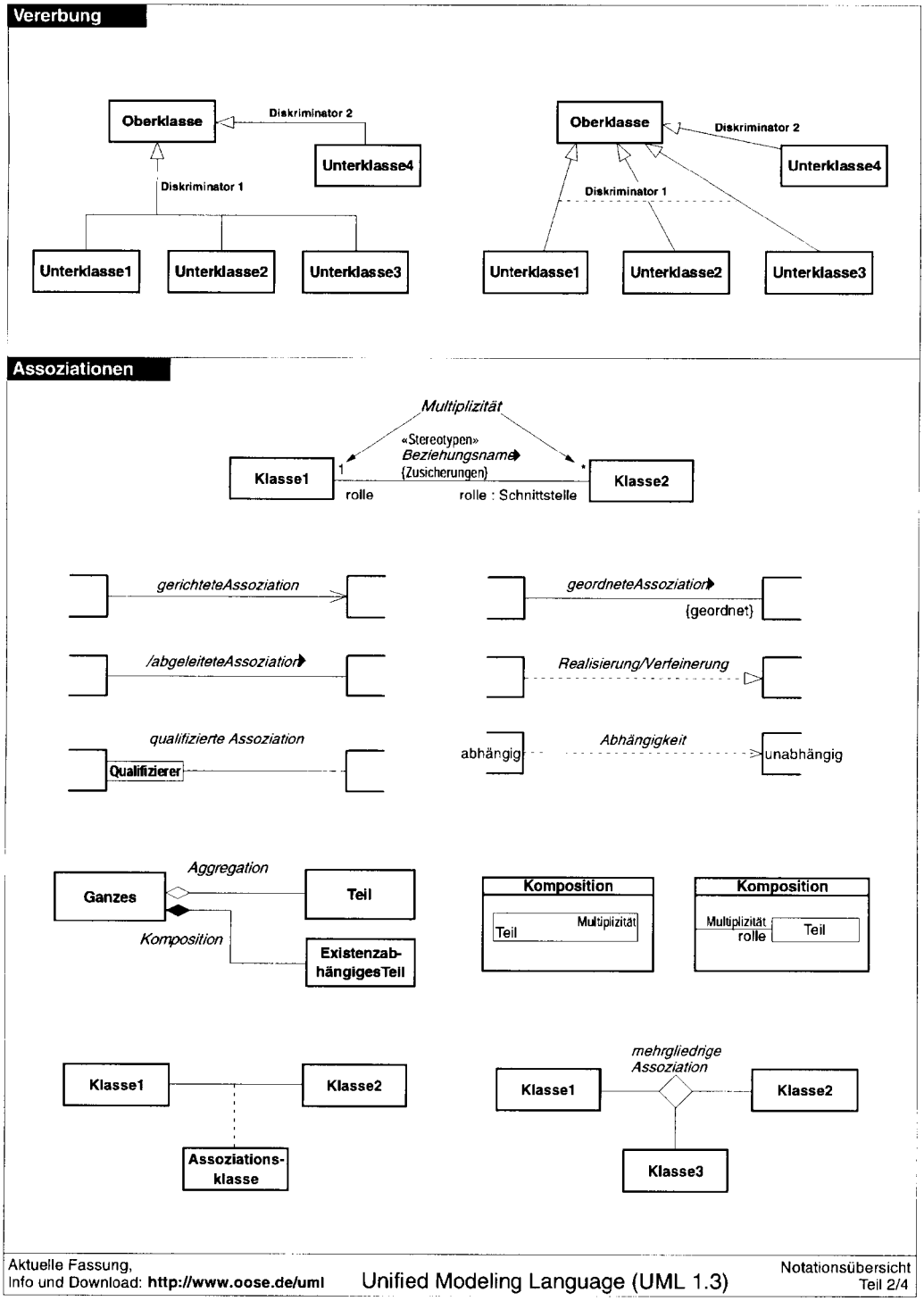
Zustandsdiagramm für den Getränkeautomaten

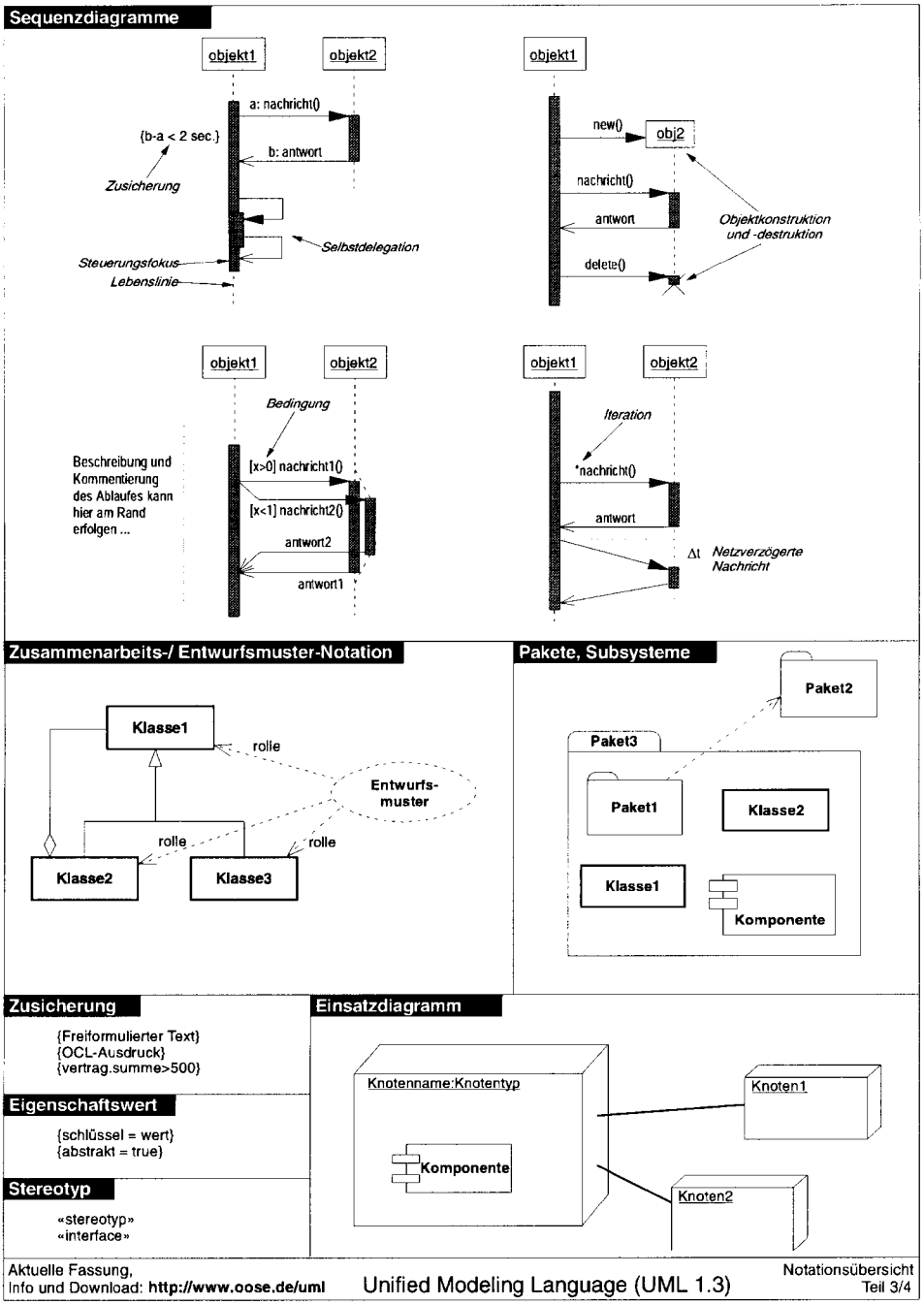


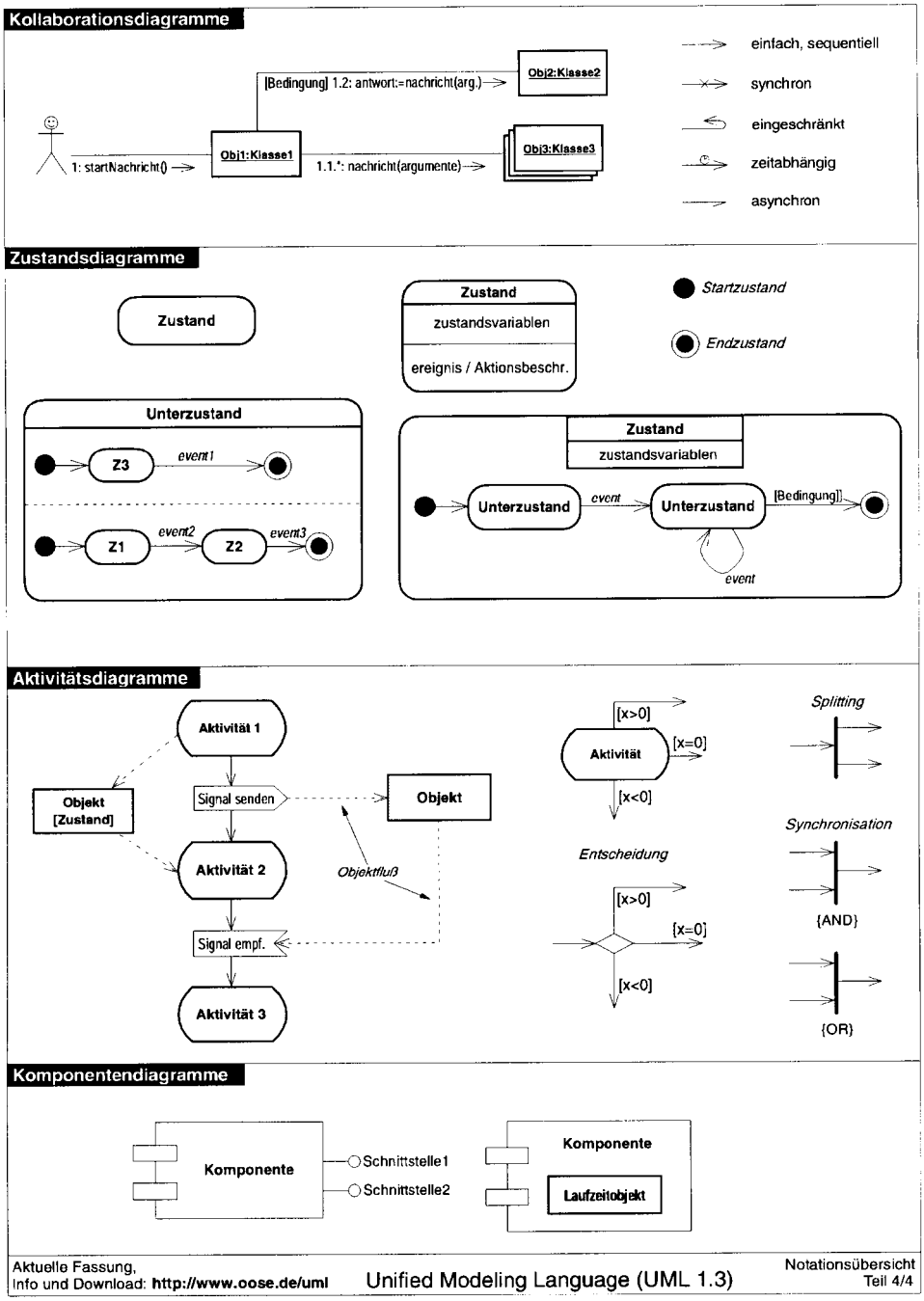
Unified Modelling Language

Notationsübersicht Unified Modelling Language (UML), entnommen aus [13]




Vererbung
Oberklasse
Diskriminator 2
Unterklasse4
Diskriminator 1
Unterklasse1
Unterklasse2
Unterklasse3
Oberklasse
Diskriminator 2
Unterklasse4
Diskriminator 1
Unterklasse1
Unterklasse2
Unterklasse3
Assoziationen
Multiplizität
«Stereotypen»
Beziehungsname
{Zusicherungen}
Klasse1
1
*
Klasse2
rolle
rolle : Schnittstelle
gerichteteAssoziation
geordneteAssoziation
{geordnet}
/abgeleiteteAssoziation
Realisierung/Verfeinerung
qualifizierte Assoziation
Qualifizierer
abhängig
Abhängigkeit
unabhängig
Aggregation
Ganzes
Teil
Komposition
Existenzab-
hängiges Teil
Komposition
Teil
Multiplizität
Komposition
Multiplizität
rolle
Teil
Klasse1
Klasse2
Assoziations-
klasse
mehrgliedrige
Assoziation
Klasse1
Klasse2
Klasse3
Aktuelle Fassung,
Info und Download: http://www.oose.de/uml
Unified Modeling Language (UML 1.3)
Notationsübersicht
Teil 2/4

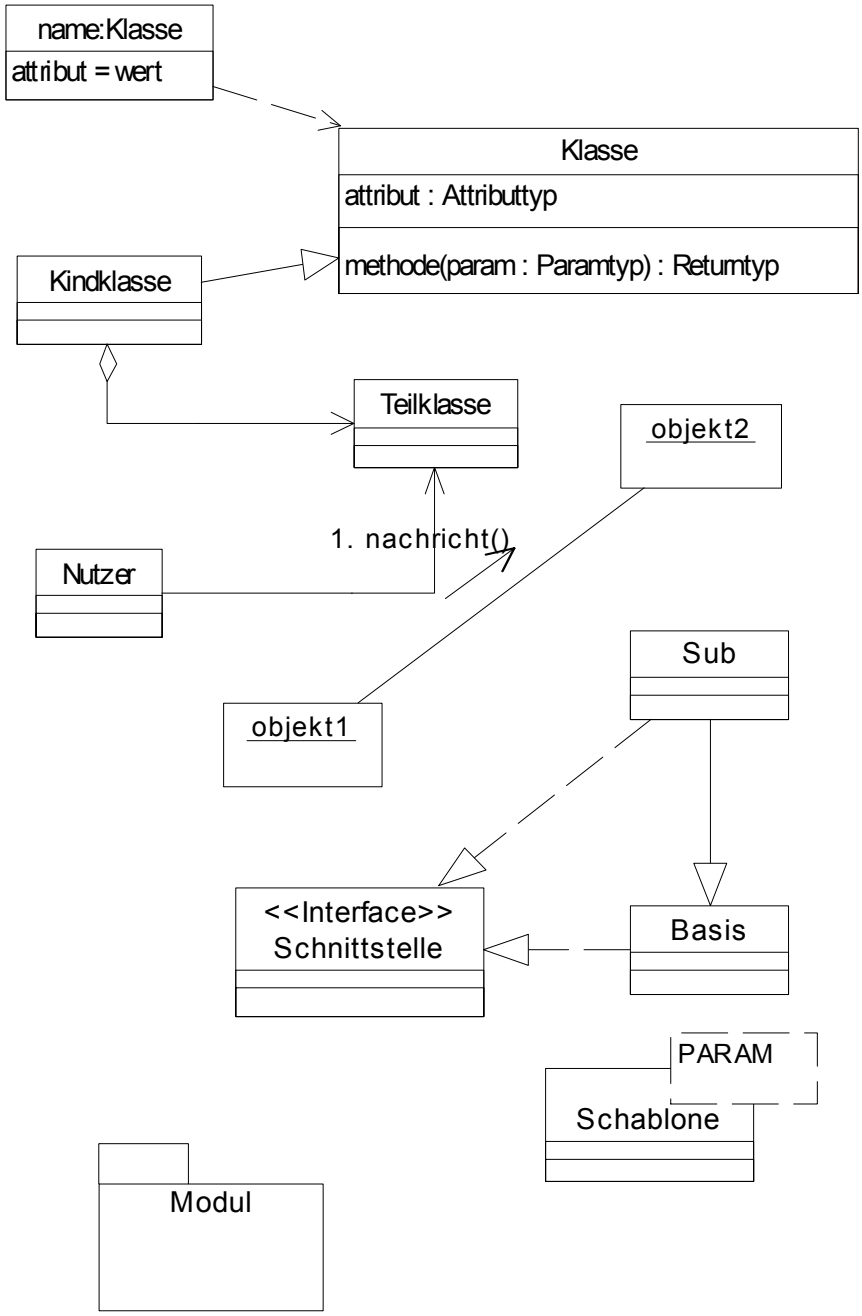

Sequenzdiagramme
objekt1
objekt2
a: nachricht()
{b-a < 2 sec.}
b: antwort
Zusicherung
Selbstdelegation
Steuerungsfokus
Lebenslinie
objekt1
new()
obj2
nachricht()
antwort
Objektkonstruktion und -destruktion
delete()
objekt1
objekt2
Bedingung
Beschreibung und Kommentierung des Ablaufes kann hier am Rand erfolgen ...
[x>0] nachricht1()
[x<1] nachricht2()
antwort2
antwort1
objekt1
objekt2
Iteration
*nachricht()
antwort
Δt
Netzverzögerte Nachricht
Zusammenarbeits-/ Entwurfsmuster-Notation
Klasse1
rolle
Entwurfs-muster
rolle
rolle
Klasse2
Klasse3
Pakete, Subsysteme
Paket2
Paket3
Paket1
Klasse2
Klasse1
Komponente
Zusicherung
{Freiformulierter Text}
{OCL-Ausdruck}
{vertrag.summe>500}
Eigenschaftswert
{schlüssel = wert}
{abstrakt = true}
Stereotyp
«stereotyp»
«interface»
Einsatzdiagramm
Knotenname:Knotentyp
Komponente
Knoten1
Knoten2
Aktuelle Fassung,
Info und Download: http://www.oose.de/uml
Unified Modeling Language (UML 1.3)
Notationsübersicht
Teil 3/4


Kollaborationsdiagramme
einfach, sequentiell
synchron
eingeschränkt
zeitabhängig
asynchron
Obj2:Klasse2
[Bedingung] 1.2: antwort:=nachricht(arg.)
Obj1:Klasse1
1: startNachricht()
1.1.*: nachricht(argumente)
Obj3:Klasse3
Zustandsdiagramme
Zustand
Zustand
zustandsvariablen
ereignis / Aktionsbeschr.
Startzustand
Endzustand
Unterzustand
Z3
event1
Z1
event2
Z2
event3
Zustand
zustandsvariablen
Unterzustand
event
Unterzustand
[Bedingung]
event
Aktivitätsdiagramme
Aktivität 1
Objekt [Zustand]
Signal senden
Objekt
Aktivität 2
Objektfluß
Signal empf.
Aktivität 3
[x>0]
Aktivität
[x=0]
[x<0]
Entscheidung
[x>0]
[x=0]
[x<0]
Splitting
Synchronisation
{AND}
{OR}
Komponentendiagramme
Komponente
Schnittstelle1
Schnittstelle2
Komponente
Laufzeitobjekt
Aktuelle Fassung,
Info und Download: http://www.oose.de/uml
Unified Modeling Language (UML 1.3)
Notationsübersicht
Teil 4/4

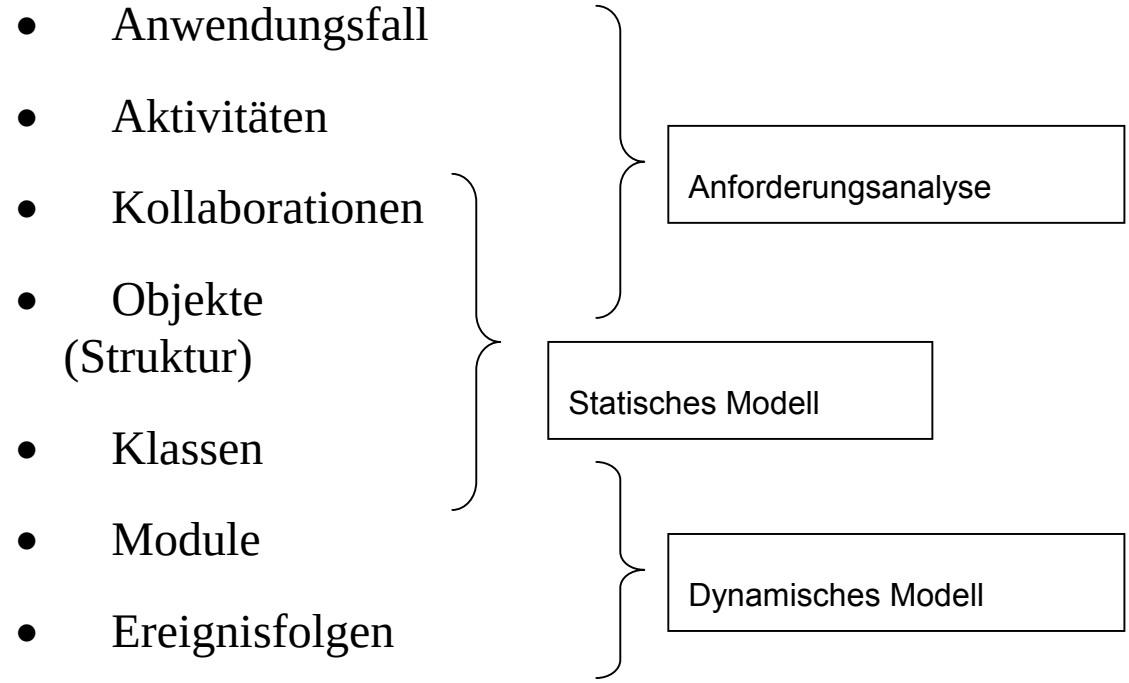
Übersichten

Begriffe der Objektorientierung und ihre UML-Darstellung

- Objekt
- Klasse
- Vererbung
- Aggregation
- Assoziation
- Nachrichten
- Polymorphie
- Schnittstelle
- Generizität
- Subsysteme



UML-Diagramme und ihre Anwendung



Einfaches Vorgehensmodell und zugehörige Diagramme

• Anwendungsfälle & Text	⇨	• USE CASE
• Einzelaktivitäten	⇨	• ACTIVITY
• Objekte/Zusammenwirk.	⇨	• OBJECT/COLL.
• Klassen/Pakete	⇨	• CLASS/PACK
• Ereignisfolgen	⇨	• SEQUENCE
• Zustände	⇨	• STATE

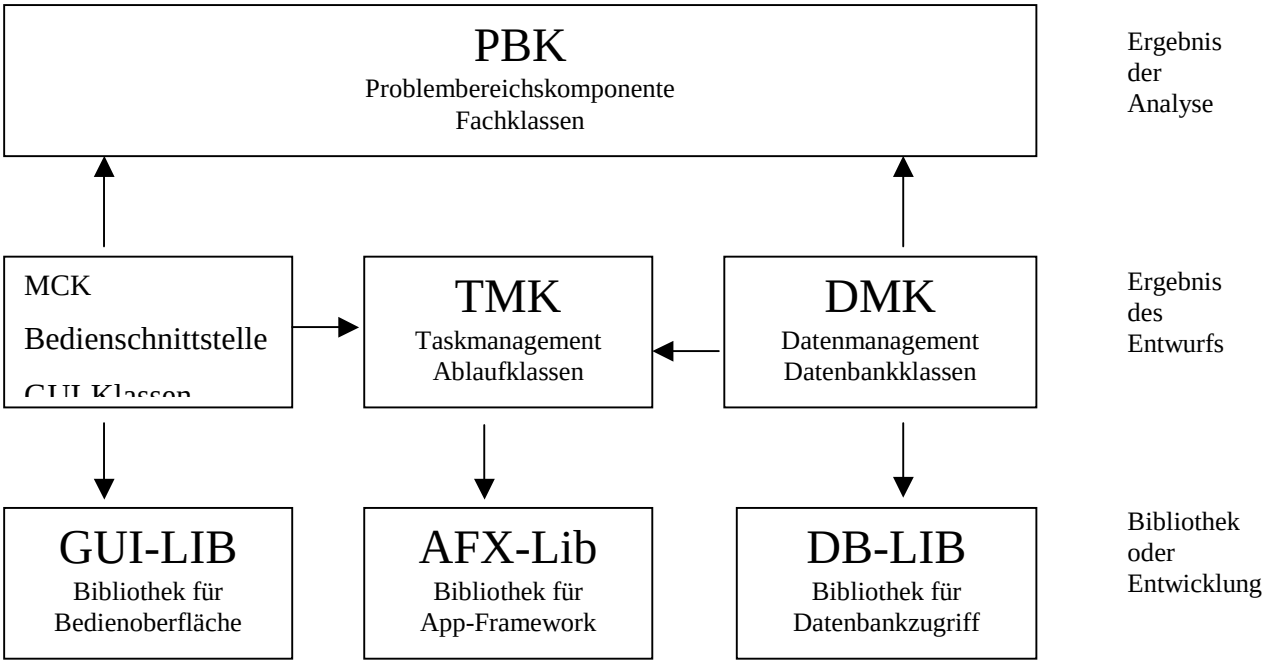
Einfaches Vorgehensmodell mit Erläuterungen

- ***Anwendungsfälle (Use Cases)***
  - ***Ermitteln und beschreiben***
  - ***Akteure sind nicht Teil des Systems***
  - ***Anwendungsfalldiagramm***
- ***Aktivitäten (Activities)***
  - ***Modellieren und beschreiben***
  - ***Eventuell mit zusätzlichen Zuständen***
  - ***Aktivitätsdiagramm***
- ***Kooperation von Objekten (mit Akteuren)***
  - ***Beschreiben mit Hilfe von Assoziationen***
  - ***Über diese Nachrichtenverbindungen versendete Nachrichten***
  - ***Diese werden später Methoden***
  - ***Kollaborationsdiagramm***
- ***Nachrichtensequenzen***
  - ***Beschreiben die Reihenfolge von Methodenaufrufen***
  - ***Sequenzdiagramm***
- ***Objekte***
  - ***Identifizieren und strukturell beschreiben***
  - ***Objektdiagramm***

    (zweckentfremdetes Klassen- oder Kollaborationsdiagramm)

- ***Klassifizierung***
  - ***Einteilung der Objekte in Klassen***
  - ***Ableitungen, Generlaisierugnen, Spezialisierungen***
  - ***Klassendiagramm***
- ***Szenarien und Methodenbeschreibung***
  - ***Detaillierter Ablauf der Methodenrealisierung***
  - ***Sequenzdiagramm***
- ***Lebenszyklus***
  - ***Ablaufwege durch Objektzustände beschreiben***
  - ***Zustandsdiagramm***
- ***Pakete***
  - ***Module und Komponenten definieren***
  - ***Moduldiagramm***
- ***Verteilung***
  - ***Objekte und Prozesse auf Rechner und Knoten***
  - ***Verteilungsdiagramm***

- ***Darstellung eines Gesamtsystems mit seinen Entwicklungsphasen***



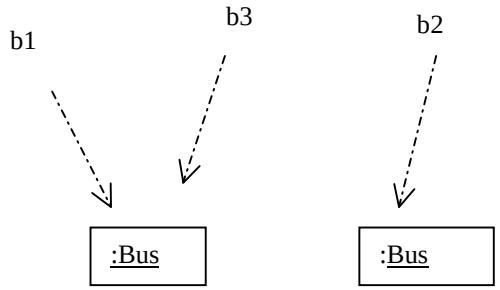
Sichten auf das System

- ***Use Case View – Anwendungsfälle***
- ***Logical View – Klassen und Pakete***
- ***Component View – Realisierungspakete***
- ***Deployment View Geräte und Rechner***

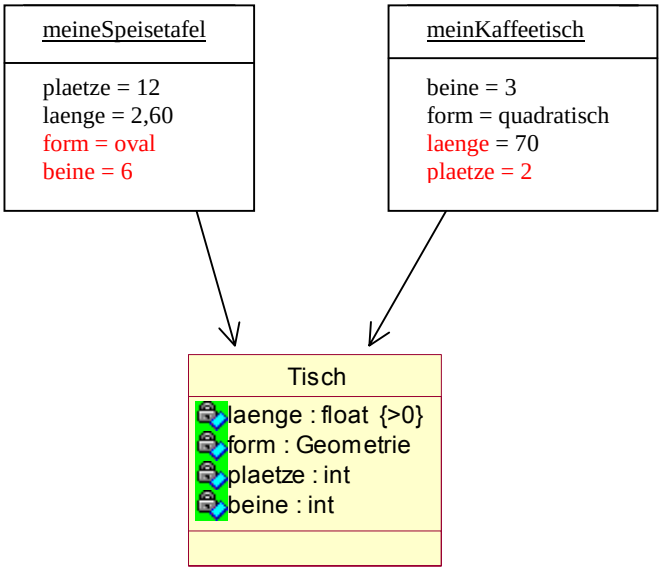
Lösungen zu den Übungen

Lösung 1:

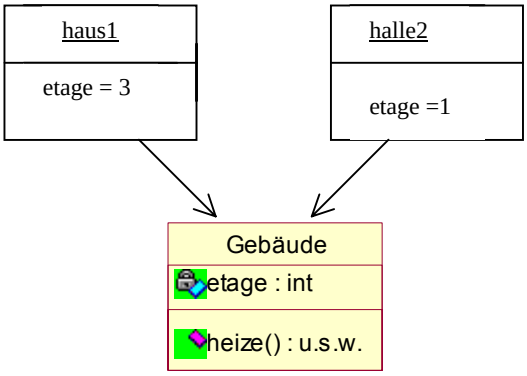
```
if (b1==b2)..;//falsch, zwei verschiedene Objekte
if (b1==b3)..;//wahr, zwei Referenzen auf ein Objekt
```



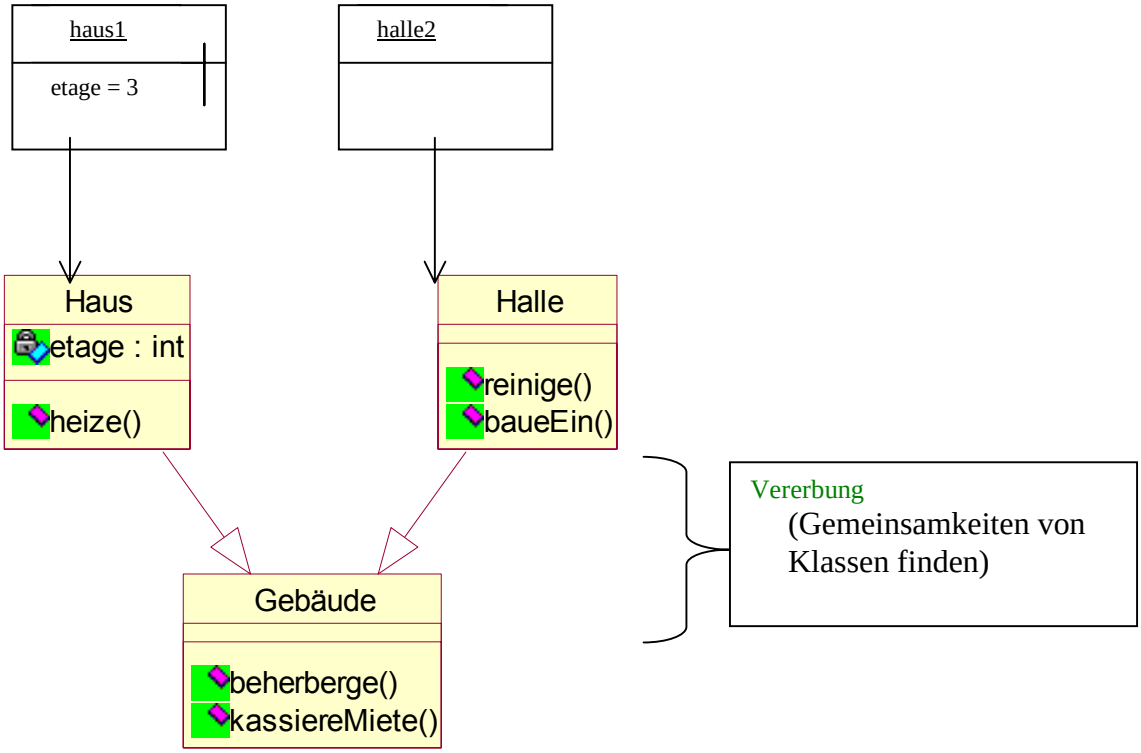
Lösung 2:



Lösung 3:

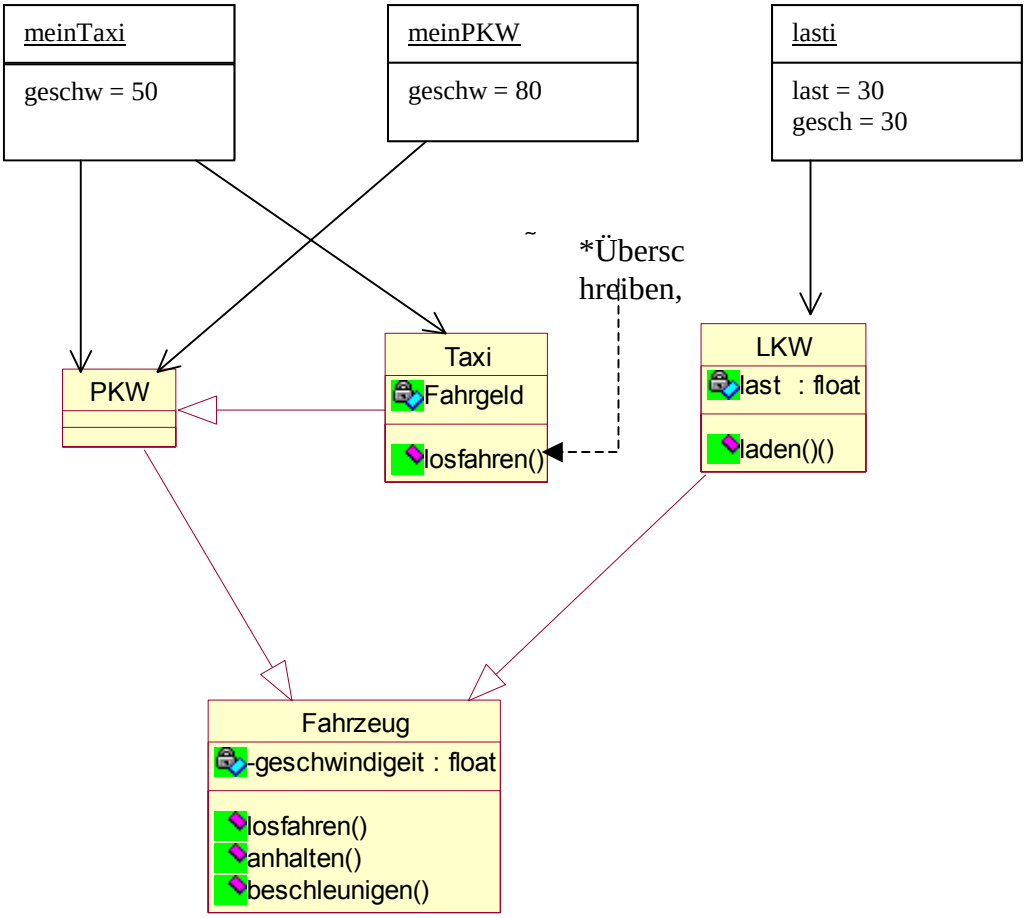


ist möglich, aber fraglich wegen eventueller unerwarteter Erweiterungen



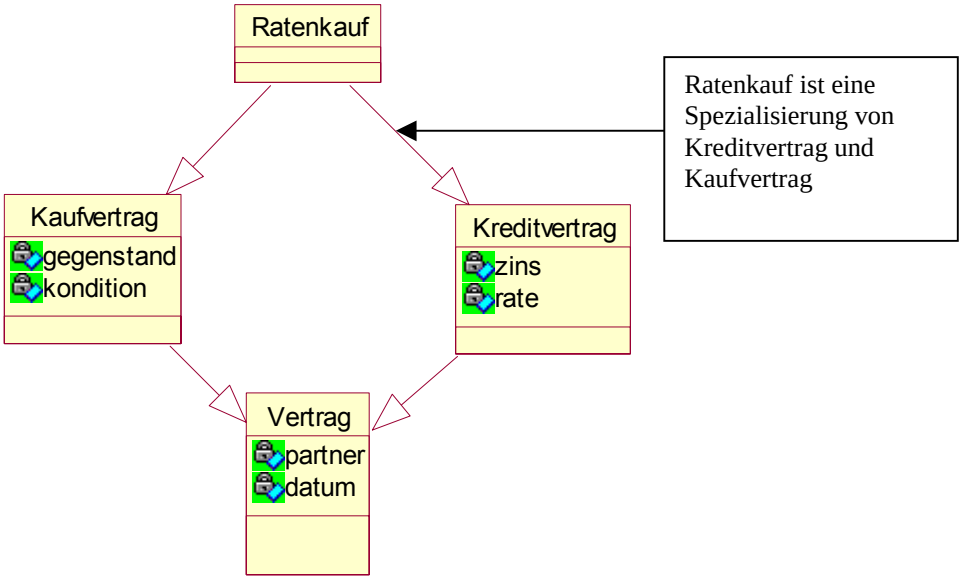
ist besser, denn die Methoden `beherberge()` und `kassiere()` der Klasse `Gebäude` werden jetzt „vererbt“ auf die abgeleiteten Klassen `Halle` und `Haus`.

Lösung 4:

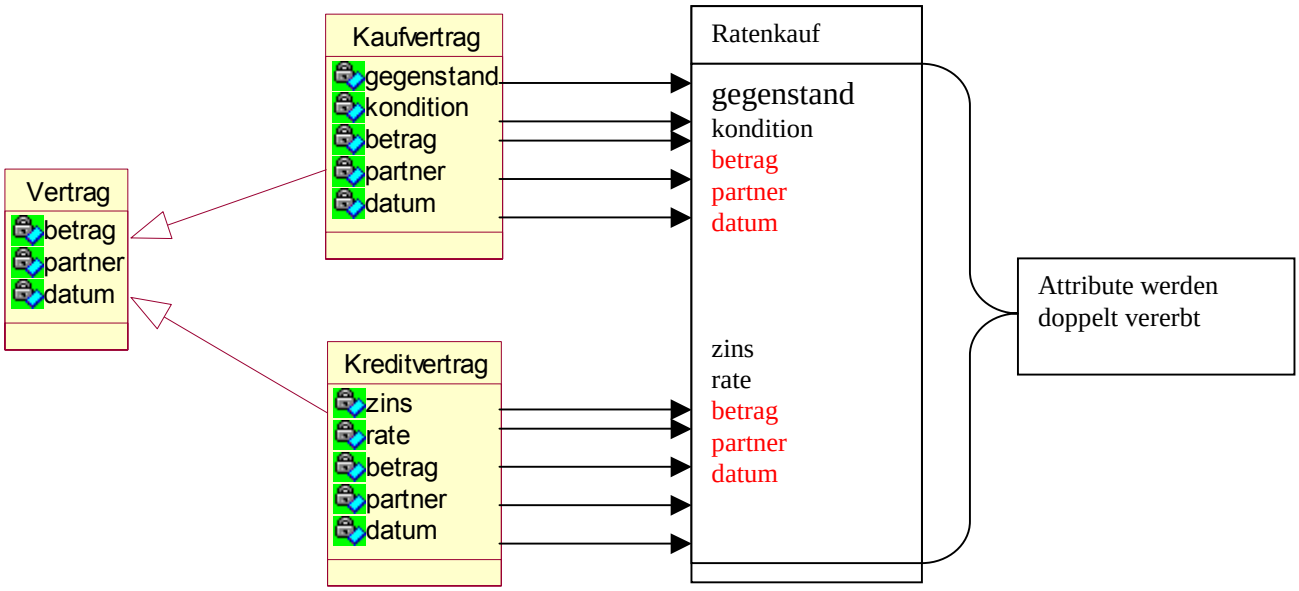


Die Methode losfahren wurde in der Klasse Taxi überschrieben, das heißt, dass sie nicht geerbt wurde, sondern neu definiert. Derselbe Methodenname wird verwendet.

Lösung 5:

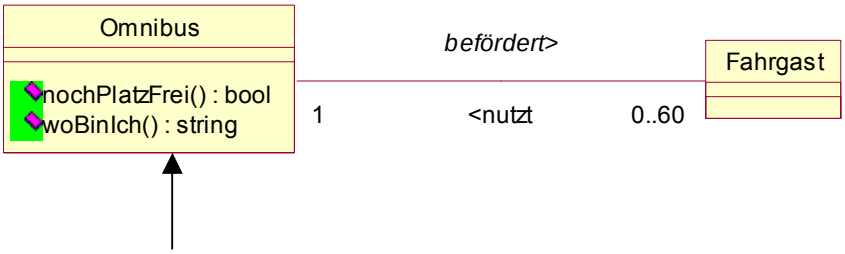


Allerdings tritt hier ein Problem auf:



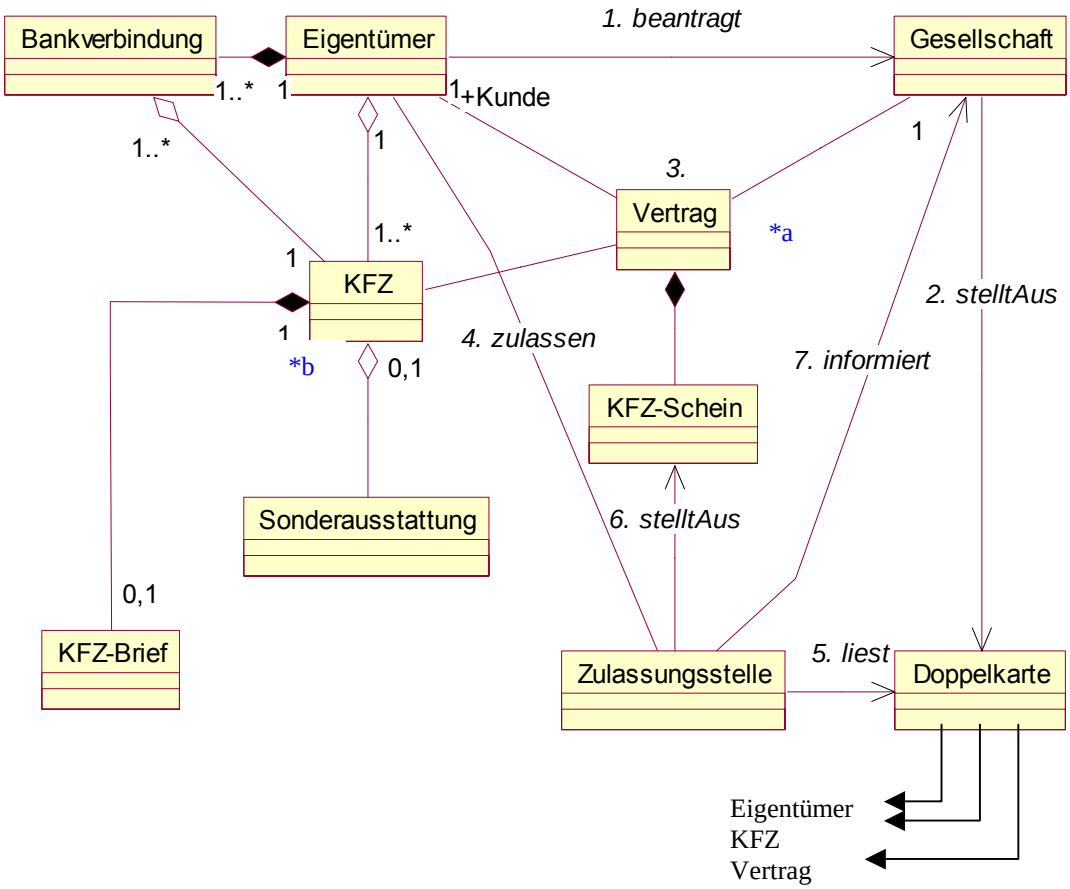
C++: Vorsicht! „virtuelle“ Erbschaft
Java: Mehrfachvererbung ist nicht möglich

Lösung 6:



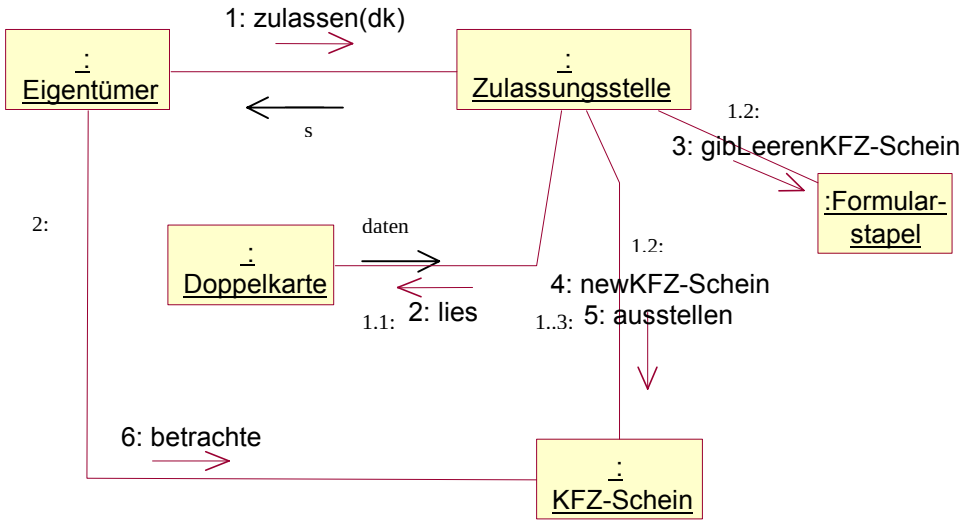
bei Anfrage können diese Ergebnisse an Fahrgast abgegeben werden

Lösung 7:

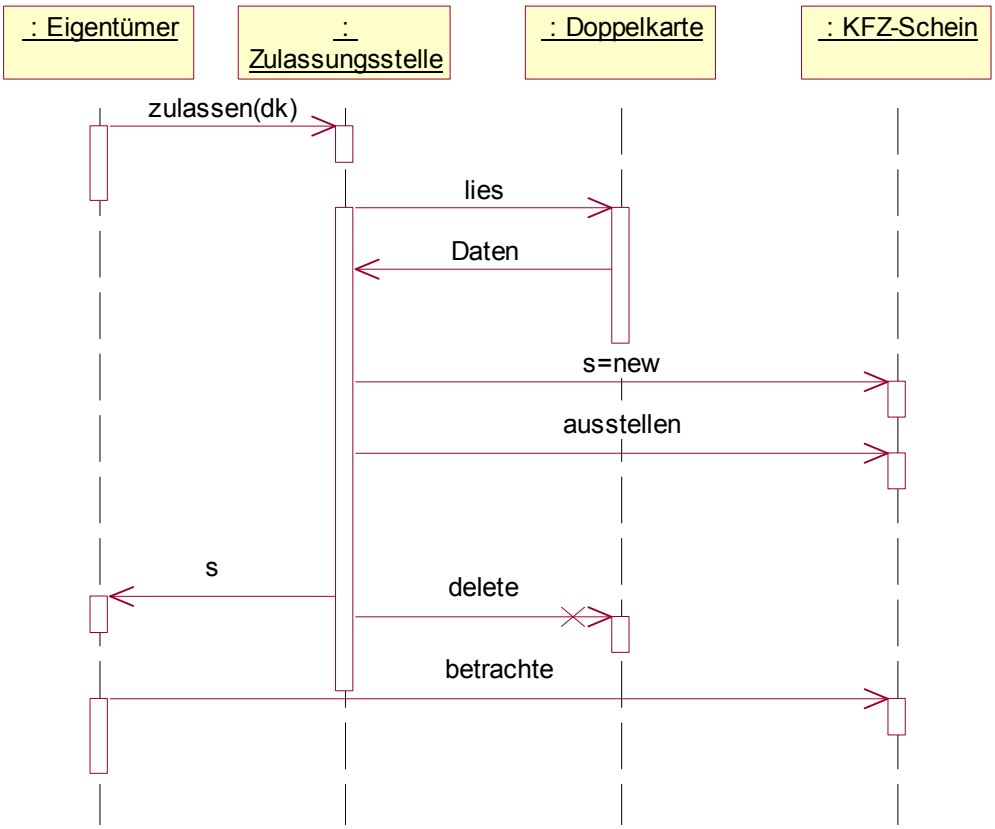


*a attributierte Assoziation, die aufgelöst wurde
*b mehrgliedrige Assoziation, da KFZ auch in Verbindung mit Vertrag

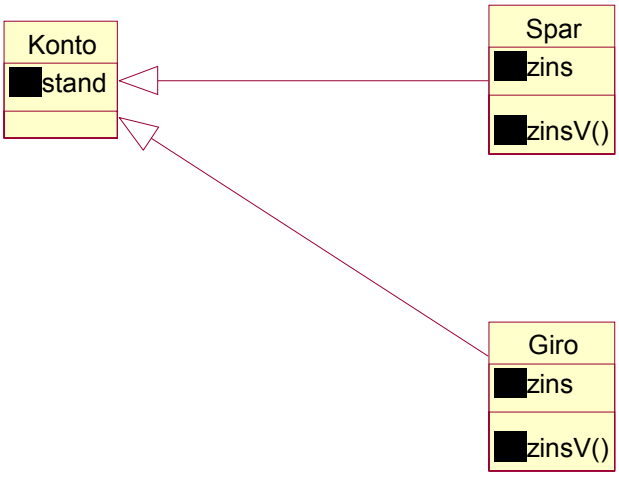
Lösung 8:



Lösung 9:



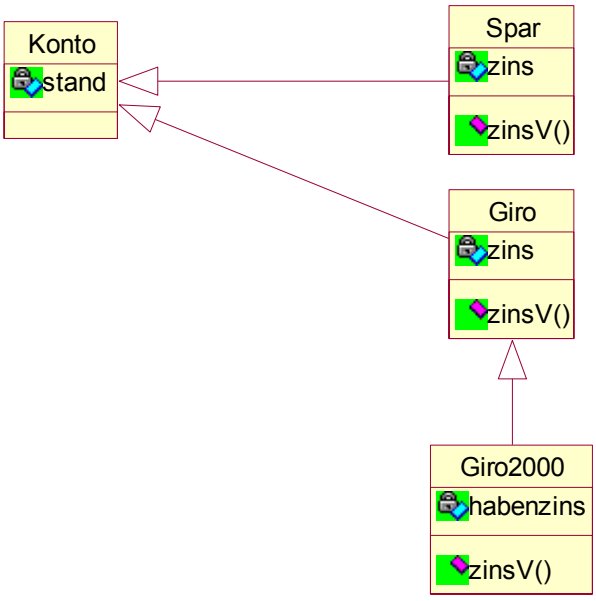
Lösung 10:




```
spar.zinsV()
{
  stand = stand +stand * zins;
}
```

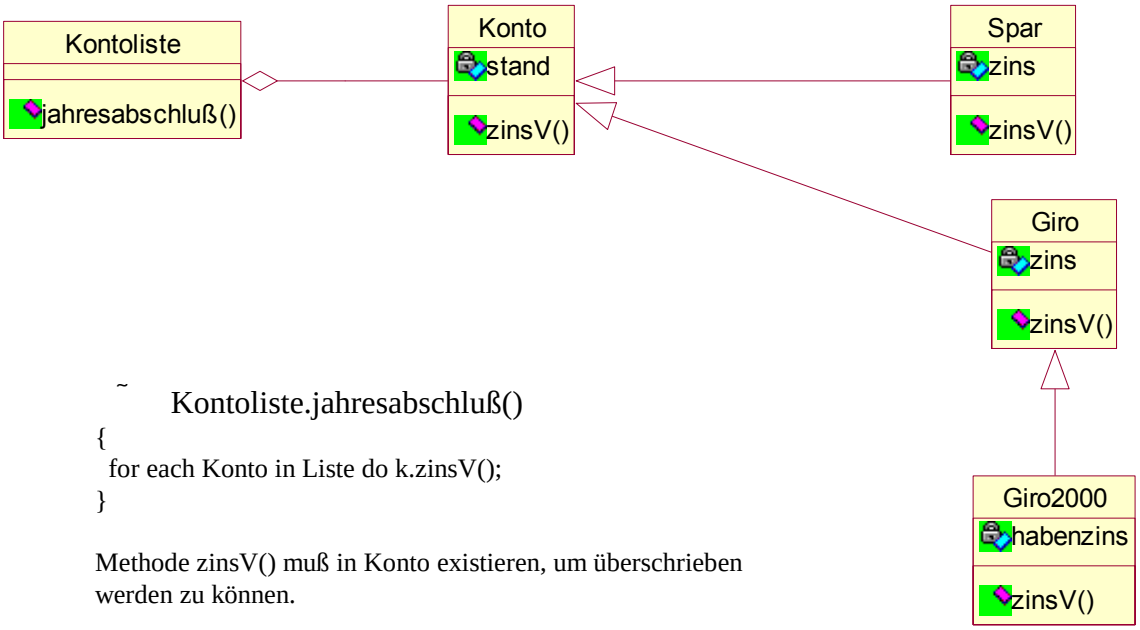
```
Giro.zinsV()
{
  if stand <0
  stand = stand - |stand| * zins
}
```

Lösung 11:




```
Giro2000.zinsV()
{
  if (stand >2000)
  stand = stand * habenzins;
  else super.zinsV();
}
```

Lösung 12:

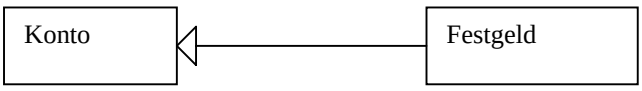


~ Kontoliste.jahresabschluß()
{
for each Konto in Liste do k.zinsV();
}

Methode zinsV() muß in Konto existieren, um überschrieben werden zu können.

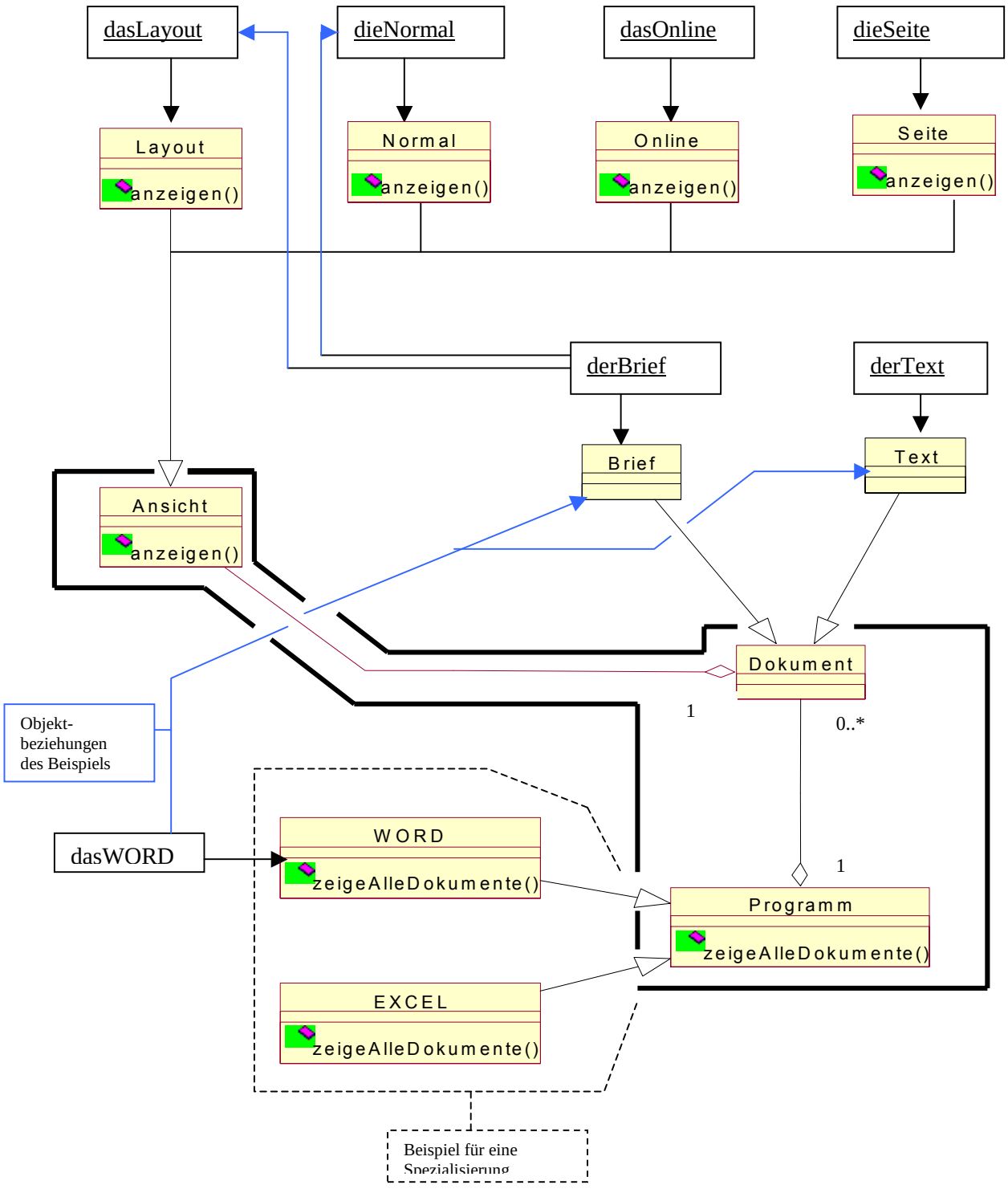
Konto.zinsV()
{} // **LEER!**

Lösung 13:



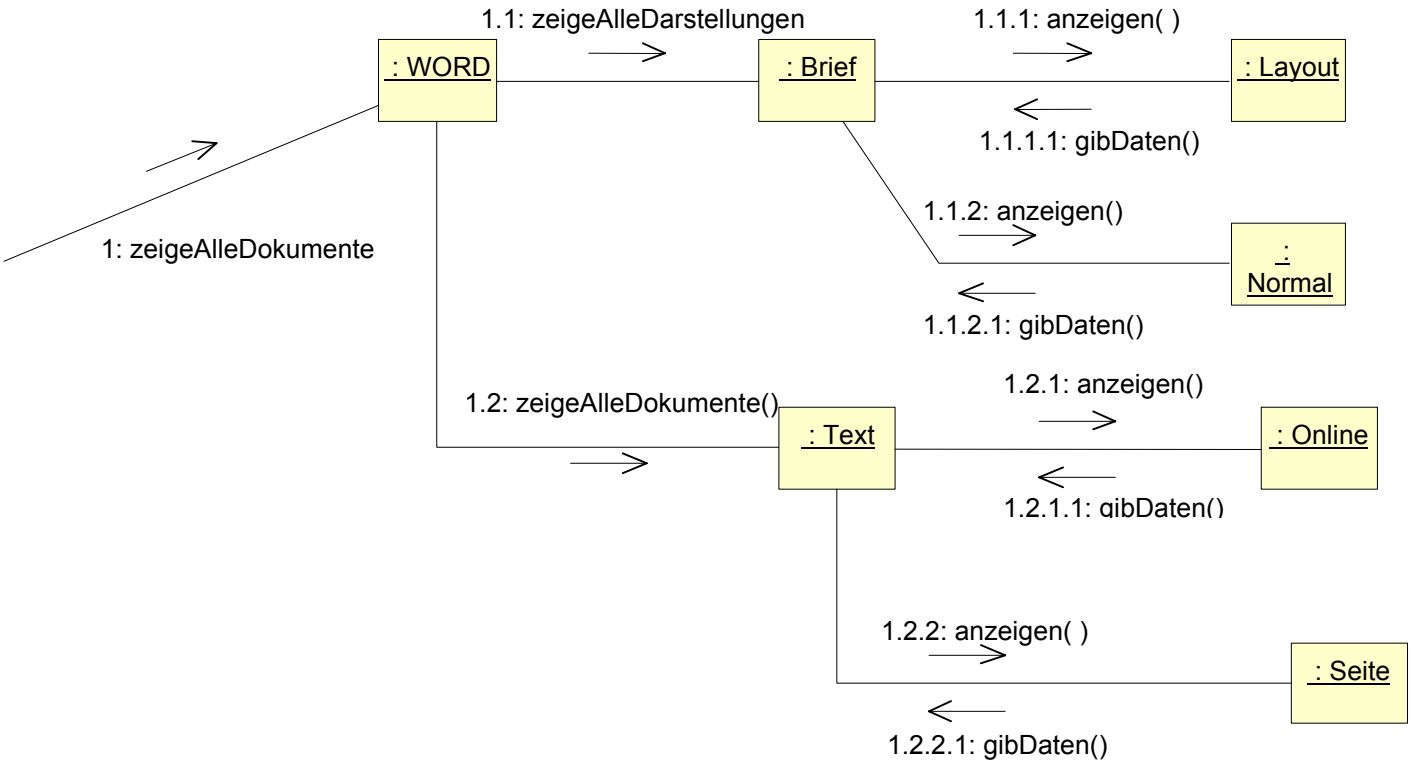
Die Klasse Festgeld ist ein direkter Nachfolger von Konto, weil keine Spezialisierung der anderen.

Lösung 14:



Es fehlt: Assoziation von Ansicht zum Dokument – Methode `gibDaten()` von Dokument

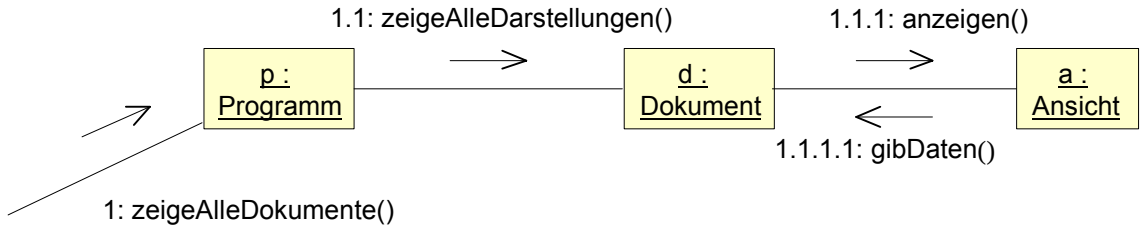
Kollaborationsdiagramm



Methoden in Basisklassen:

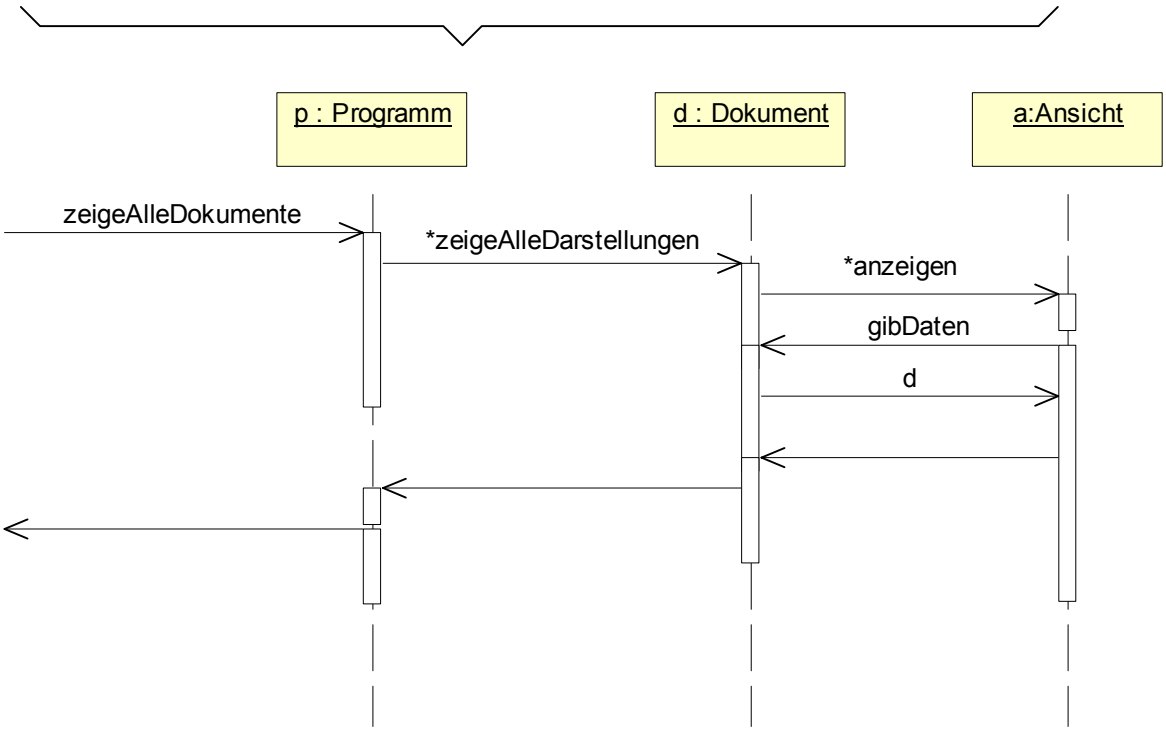
```
Programm.zeigeAlleDokumente()
{
  for each Dokument d do d.zeigeAlleDarstellungen; //polymorph
}

Dokument.zeigeAlleDarstellungen()
{
  for each Ansicht a do a.anzeigen();  //polymorph
}
```

Vereinfacht:



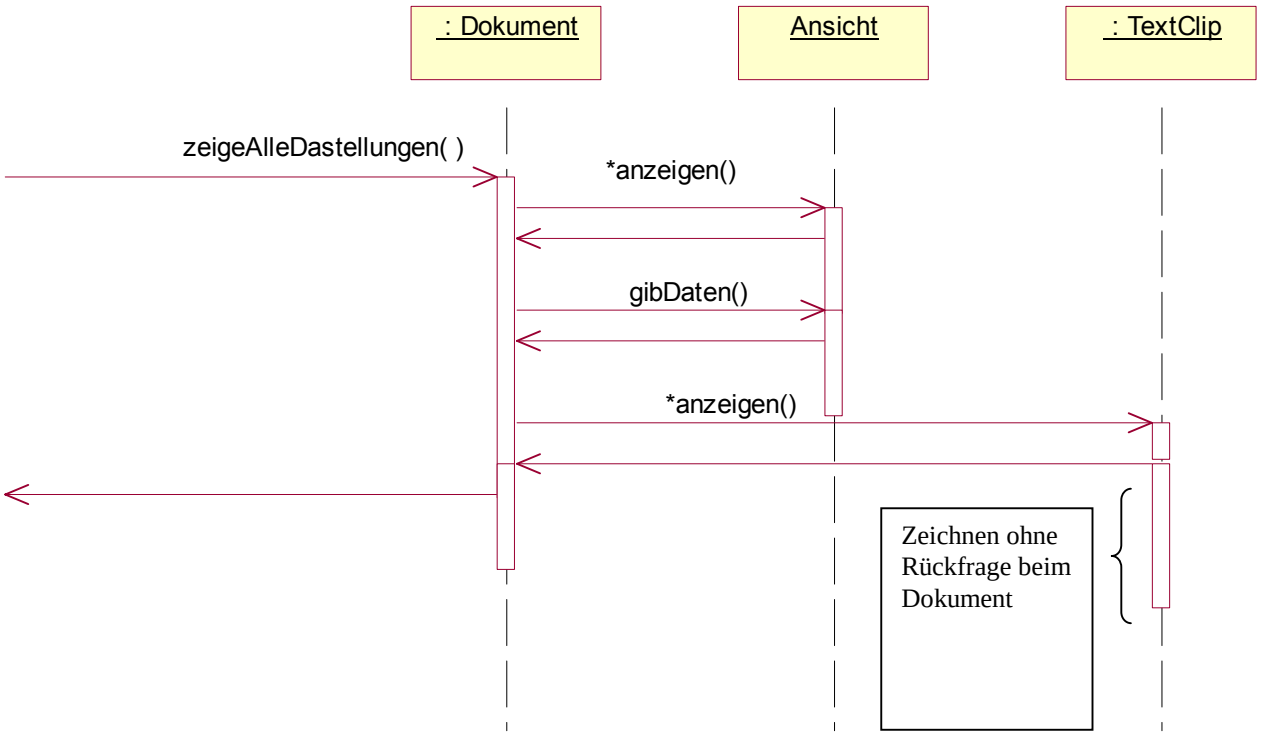
Sequenzdiagramm

Dokument und Ansicht bedingen sich gegenseitig (nicht nur in eine Richtung)!
In der Praxis ist dies aber oft Normalfall.
(Insbesondere die MFC nutzen diese Art der Architektur)

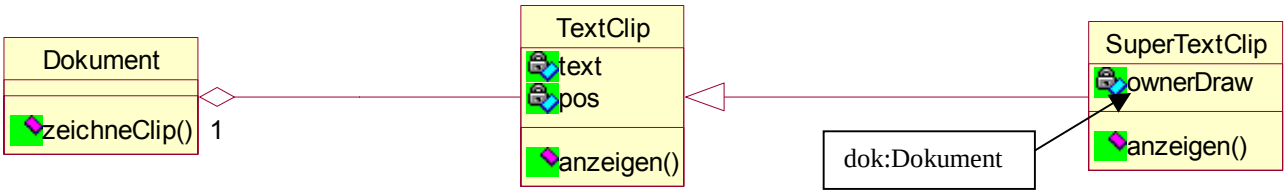


* = Iteration über alle –Dokumentobjekte
- Ansichtsobjekte

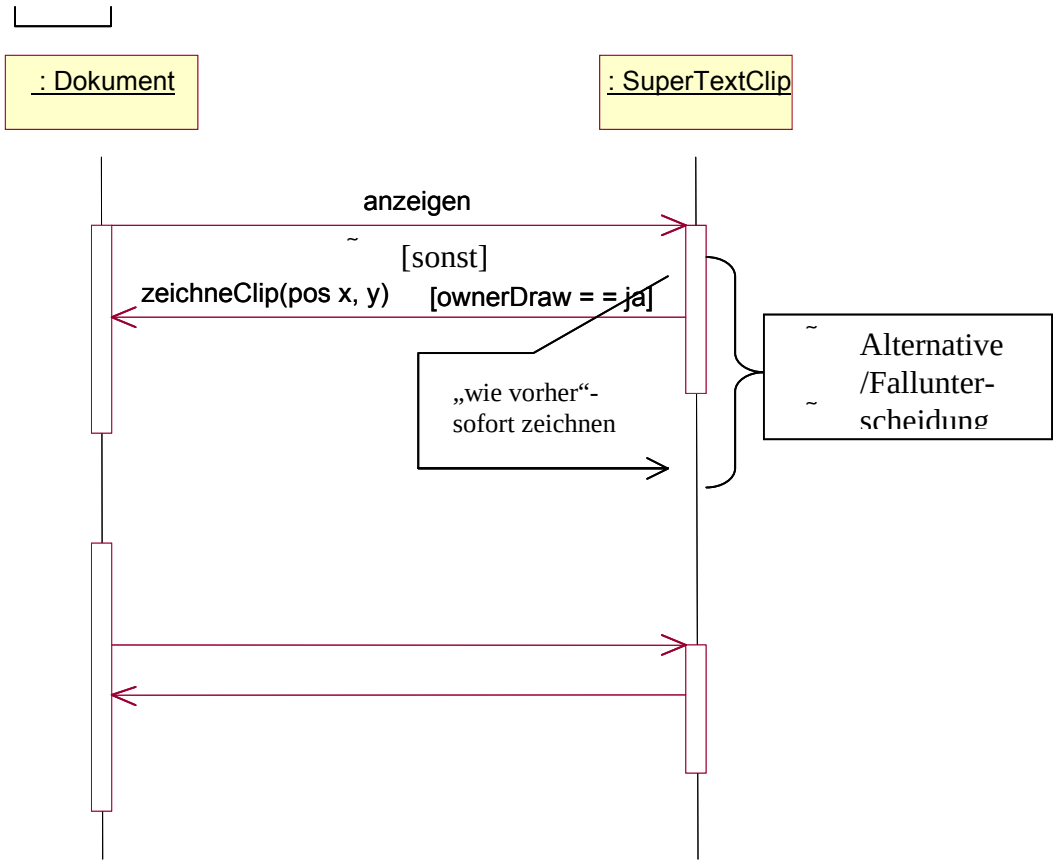
Lösung 15:


: Dokument
Ansicht
: TextClip
zeigeAlleDastellungen( )
*anzeigen()
gibDaten()
*anzeigen()
Zeichnen ohne Rückfrage beim Dokument

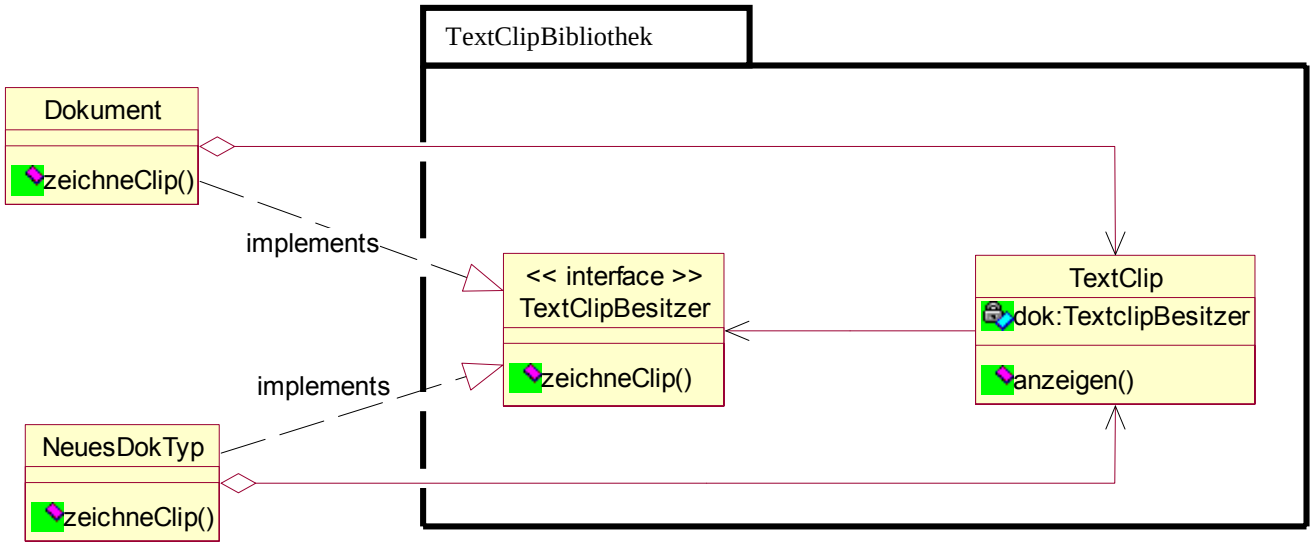
Lösung 16:




```
Erweiterung von :
SuperTextClip.anzeigen()
{
  if (ownerDraw = = false) „geerbte.“anzeigen
  else dok.zeichneClip (pos, text);          // brauche also
Assoziation!(d.h. Attribut dok:Dokument)
}

Beispiel für
Dokument.zeichneClip (text, pos)
{
  „zeigeBitmap“(pos,text)     // z.B. als Dateiname
interpretiert
}
```



Lösung 17:

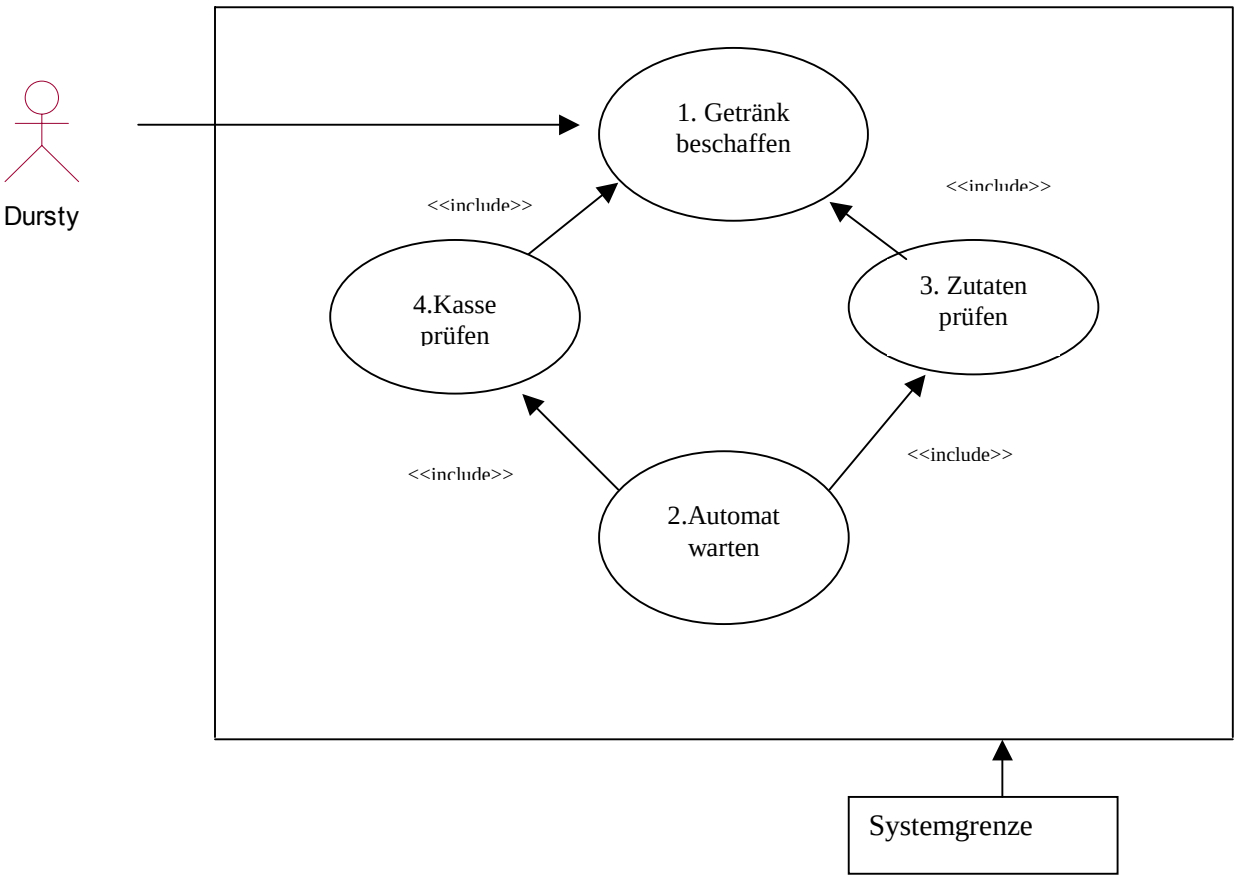


TextClip ist nun völlig unabhängig von Dokument. Das Dokument muss lediglich die Schnittstelle TextClipBesitzer implementieren.

Lösung 18:

Erinnerung:

- Anwendungsfall: abgeschlossener Geschäftsvorfall
- nicht zu fein modellieren, **keine** Abläufe
- <<include>> nur für Mehrfachverwendung (nicht zur funktionalen Zerlegung)

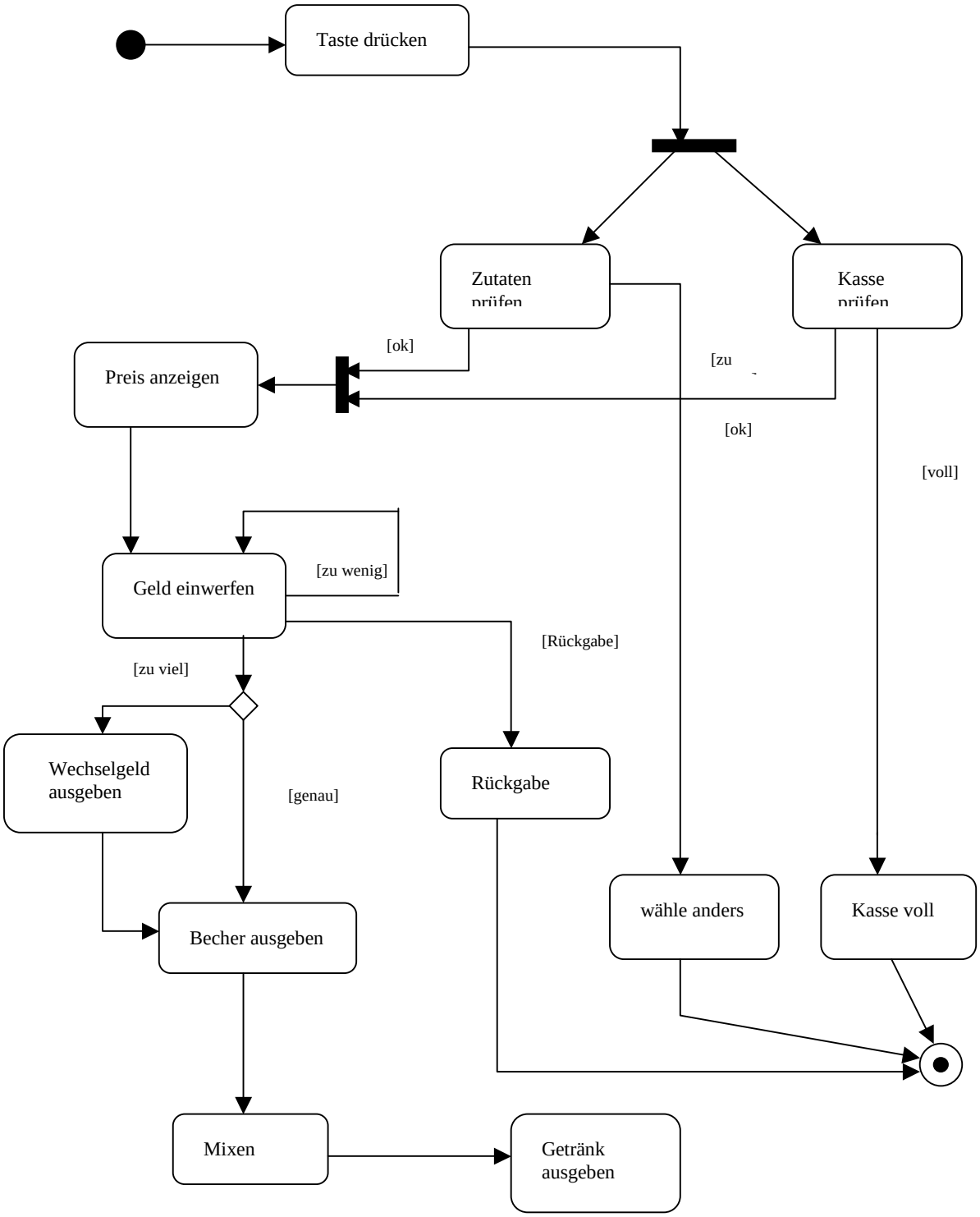


Anwendungsfall 1: Getränk beschaffen

Vorbedingung: -Dursty vor Automat
-Automat aktiv
Einzelaktivitäten: 1. Getränkeauswahl: Taste drücken
2. Zulässigkeit prüfen
2a. Zutaten prüfen
2b. Kasse prüfen
3. Preisanzeige oder Ausnahme
4. Geldeinwurf (bis genug) oder Ausnahme
5. Getränkezubereitung: Becherauswurf, Mixen, Ausgabe in Becher
Nachbedingung, Ergebnisse: Getränk ausgeben

Ausnahmen: 3a.. zuwenig Zutaten: Meldung „anderes Getränk“
3b. Kasse voll: Meldung „Automat nicht bereit“
4a. Rückgabetaste: Ende Anwendungsfall

Lösung 19:


Taste drücken
Zutaten prüfen
Kasse prüfen
[ok]
[zu
Preis anzeigen
[ok]
[voll]
Geld einwerfen
[zu wenig]
[Rückgabe]
[zu viel]
Wechselgeld ausgeben
[genau]
Rückgabe
wähle anders
Kasse voll
Becher ausgeben
Mixen
Getränk ausgeben

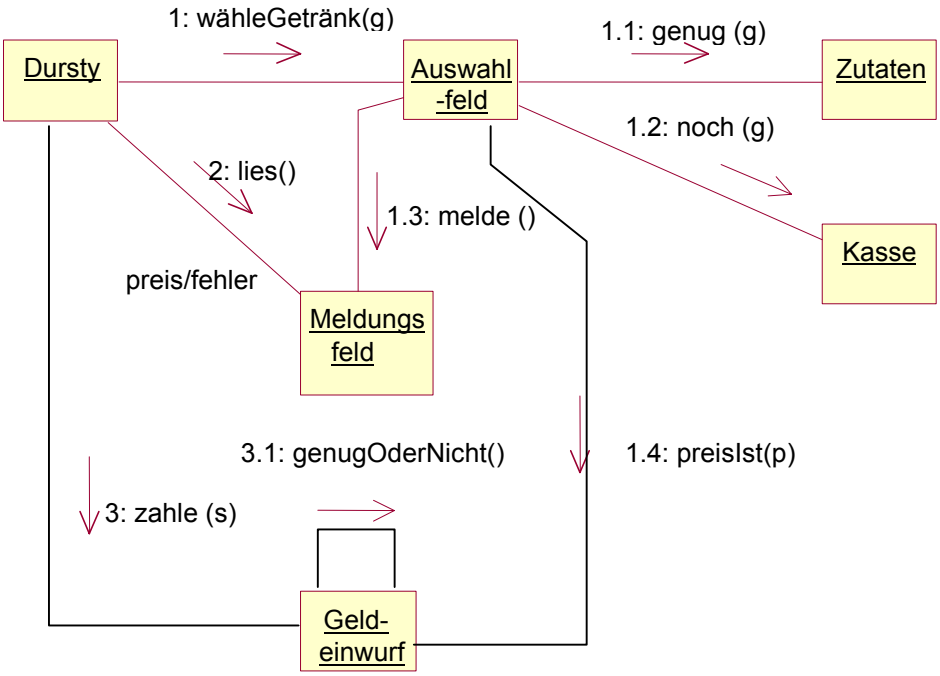
Lösung 20:

a) Kandidaten für Objekte

Dursty Automat Kasse Zutaten Auswahlfeld Wärter Geldeinwurf Getränke-Ausgabe Geldrückgabe Geld Meldungsfeld Mixer

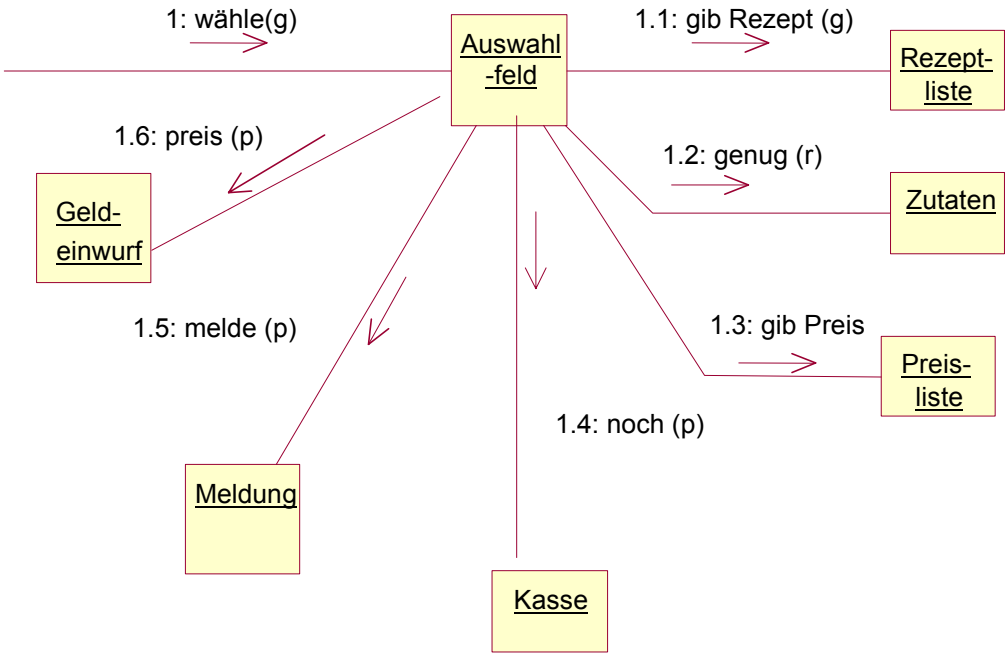
b) Kollaborationsdiagramm Use Case „Getränkebeschaffung“

1. Ansatz:



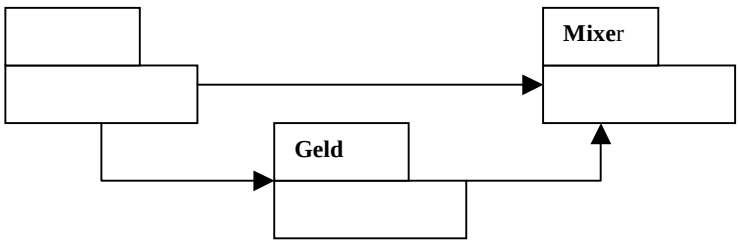
Nachteil: Auswahlfeld muss so alle Preise und Rezepte kennen
Tipp: Klassen lose gekoppelt
Idee: Füge die Objekte Preisliste und Rezeptliste ein

2.Ansatz



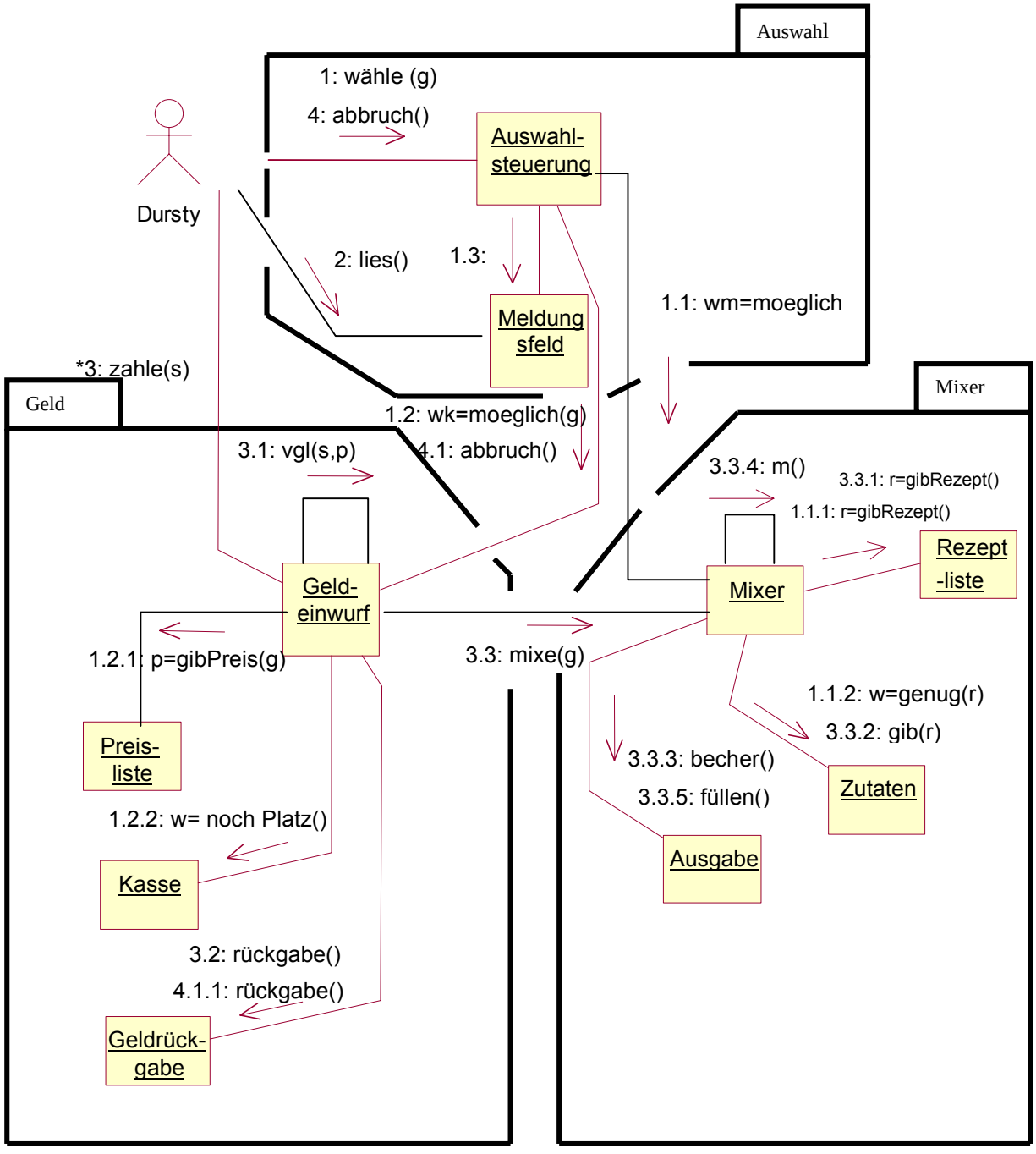
Nachteil: Auswahlfeld ist jetzt eine übermächtige Steuerungsinstanz
Idee: Entkopple die vorhandenen Objekte möglichst weit und delegiere Aufgaben

3. Ansatz:
Modulabhängigkeiten:

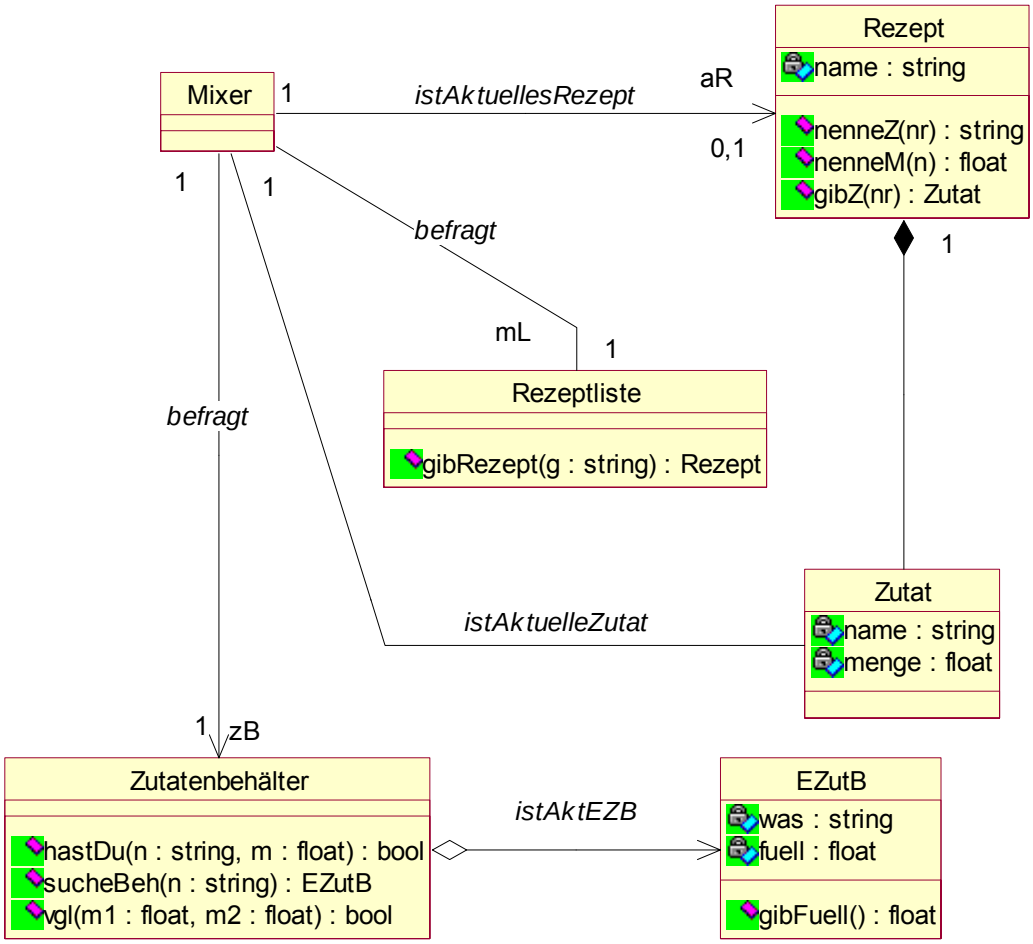


Das Modul Geld stellt nur Dienste zur Verfügung, benutzt keine Methode von Auswahlfeld.
Dasselbe Prinzip gilt für die anderen Module auch.

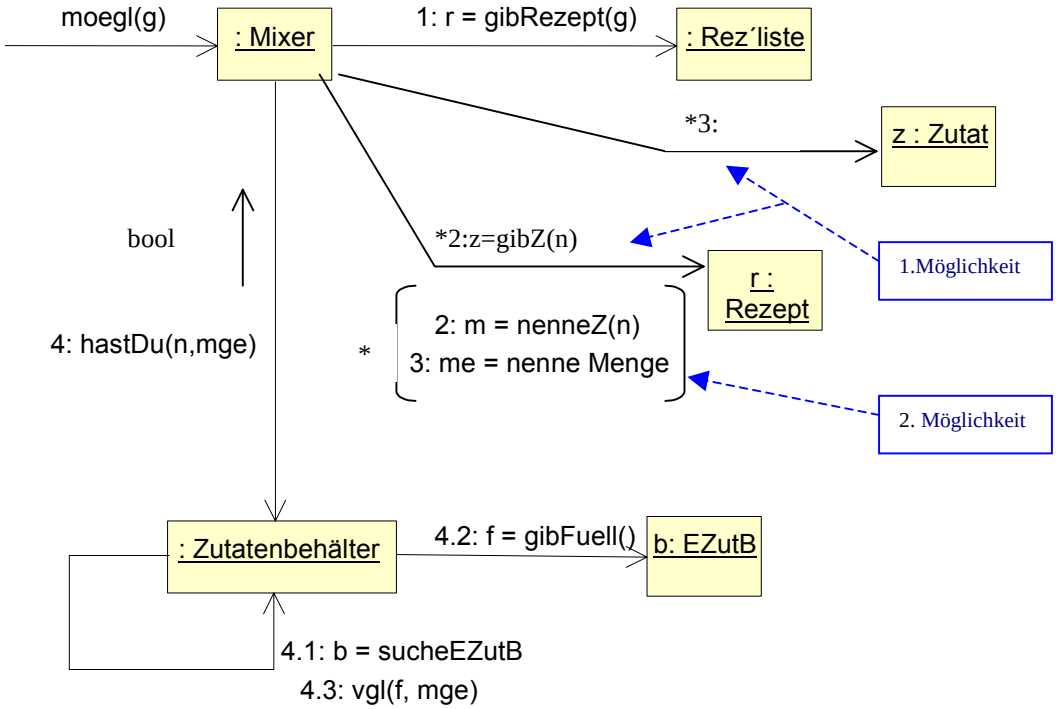
Nebenbemerkung: Dursty ist ein Akteur und kein Objekt im System



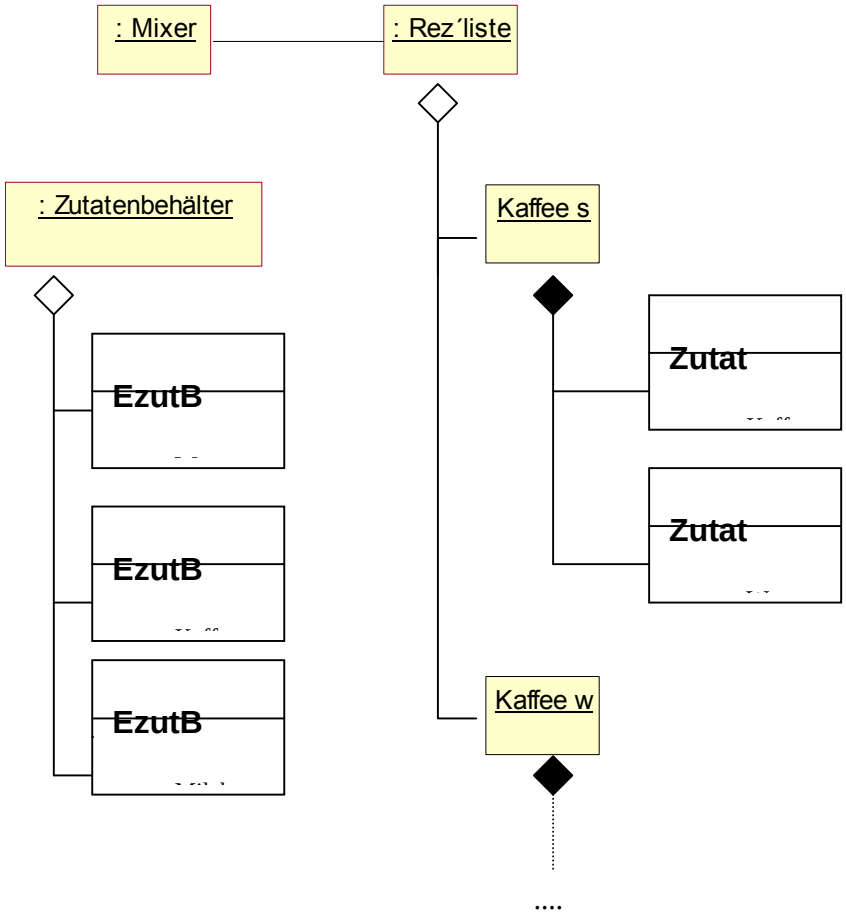
Lösung 21:


Rezept
name : string
nenneZ(nr) : string
nenneM(n) : float
gibZ(nr) : Zutat
Mixer
1
istAktuellesRezept
aR
0,1
1
1
befragt
1
mL
1
Rezeptliste
gibRezept(g : string) : Rezept
befragt
1
Zutat
name : string
menge : float
istAktuelleZutat
1
zB
Zutatenbehälter
hastDu(n : string, m : float) : bool
sucheBeh(n : string) : EZutB
vgl(m1 : float, m2 : float) : bool
istAktEZB
EZutB
was : string
fuell : float
gibFuell() : float

Lösung 22:

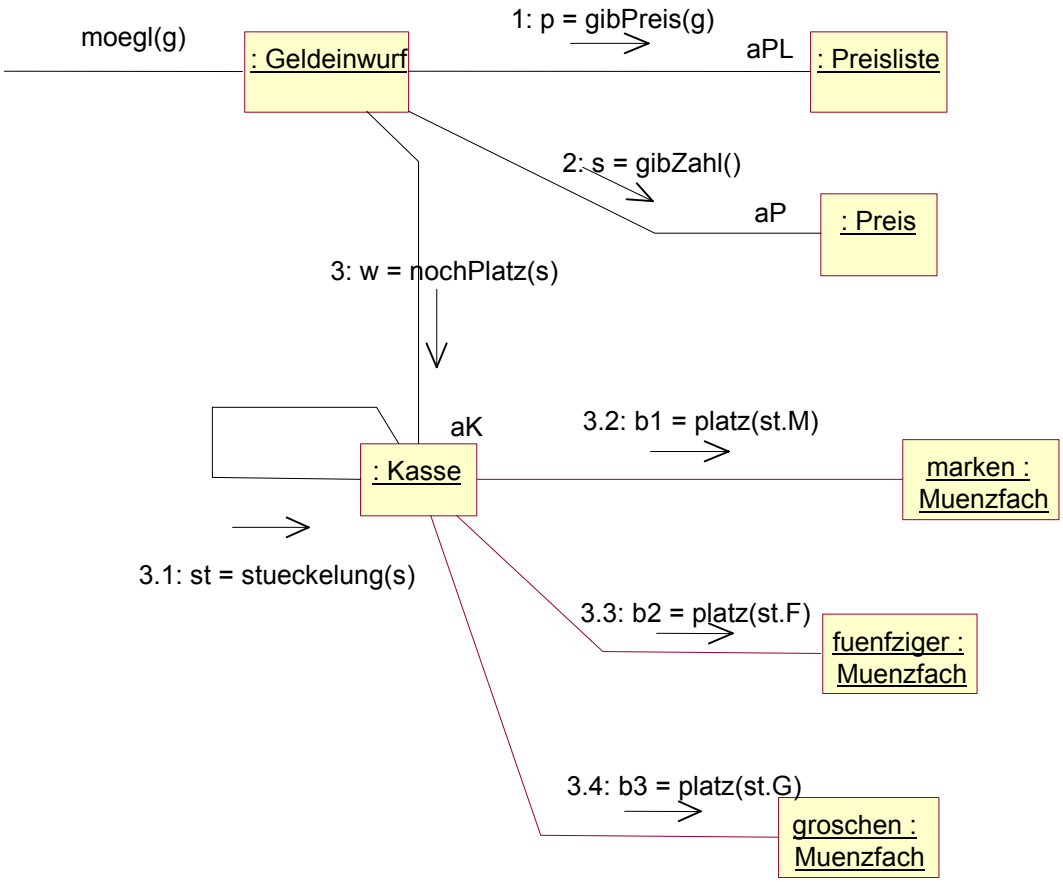

moegl(g)
: Mixer
1: r = gibRezept(g)
: Rez´liste
*3:
z : Zutat
bool
*2:z=gibZ(n)
r : Rezept
1.Möglichkeit
4: hastDu(n,mge)
*
2: m = nenneZ(n)
3: me = nenne Menge
2. Möglichkeit
: Zutatenbehälter
4.2: f = gibFuell()
b: EZutB
4.1: b = sucheEZutB
4.3: vgl(f, mge)

Lösung 23:



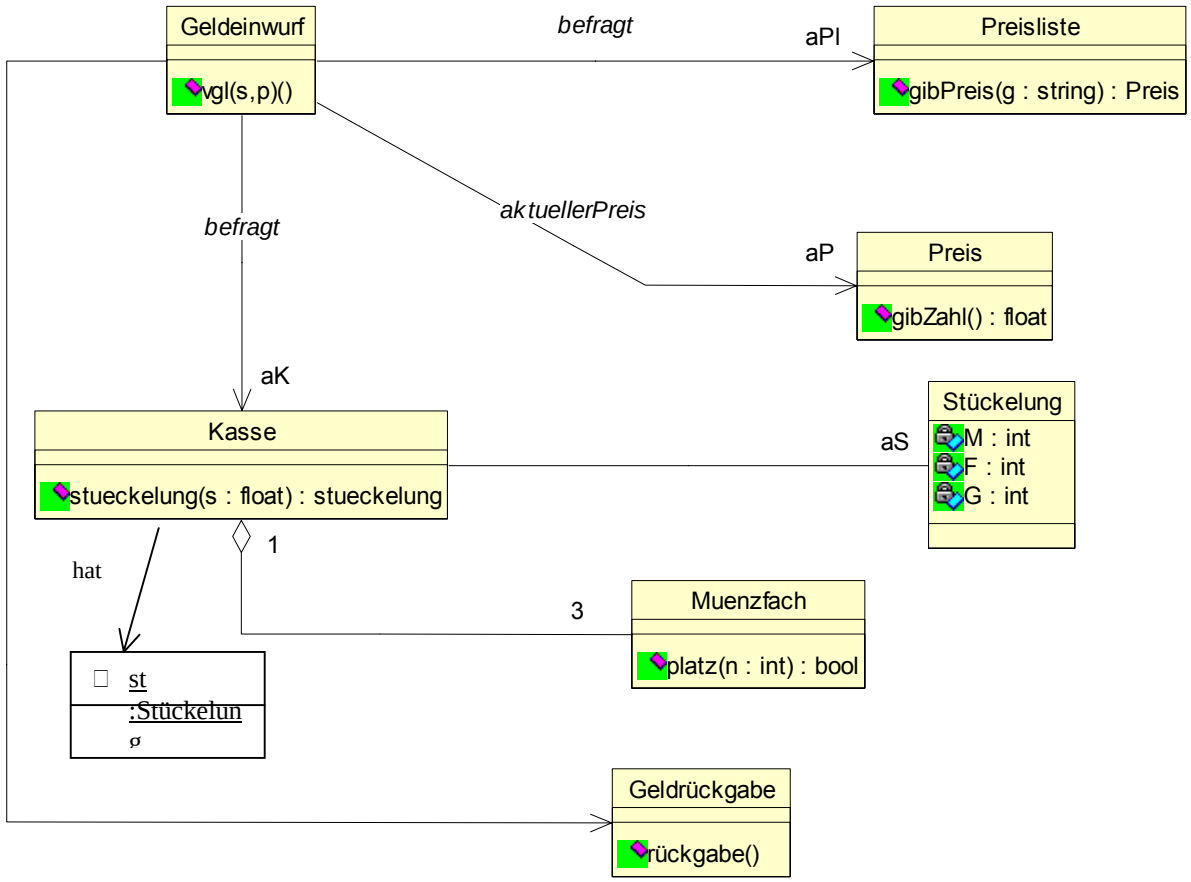

```
bool Mixer.moegl(g:string)
{
  Rezept aR = ml.gibRezept(g);
  if aR = = null) return false;
  int z = 0;
  string aktZutName “ “;
  float aktZutMenge = 1;
  while ( aktZutMenge != 0)
  {
     aktZutname = aR.gibZutName(z)
     aktZutMenge = aR.gibZutMenge(z);
     EzutBeh aB =  mZ.gibBeh (aktZutName);
     if (aB = = null) return false;
     if ( aktZutMenge > aB.gibFüllstand())  return false;
     z++;
  }
  return true;
}
```

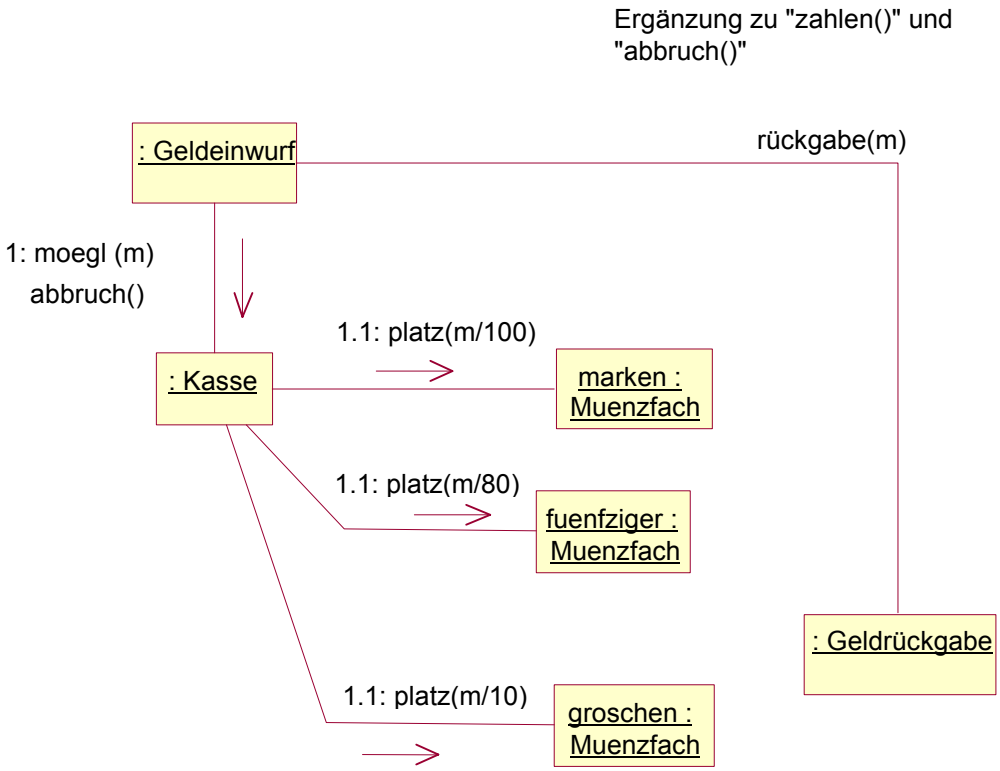
Lösung 24:


moegl(g)
: Geldeinwurf
1: p = gibPreis(g)
aPL
: Preisliste
2: s = gibZahl()
aP
: Preis
3: w = nochPlatz(s)
aK
: Kasse
3.2: b1 = platz(st.M)
marken : Muenzfach
3.1: st = stueckelung(s)
3.3: b2 = platz(st.F)
fuenfziger : Muenzfach
3.4: b3 = platz(st.G)
groschen : Muenzfach

Lösung 25:

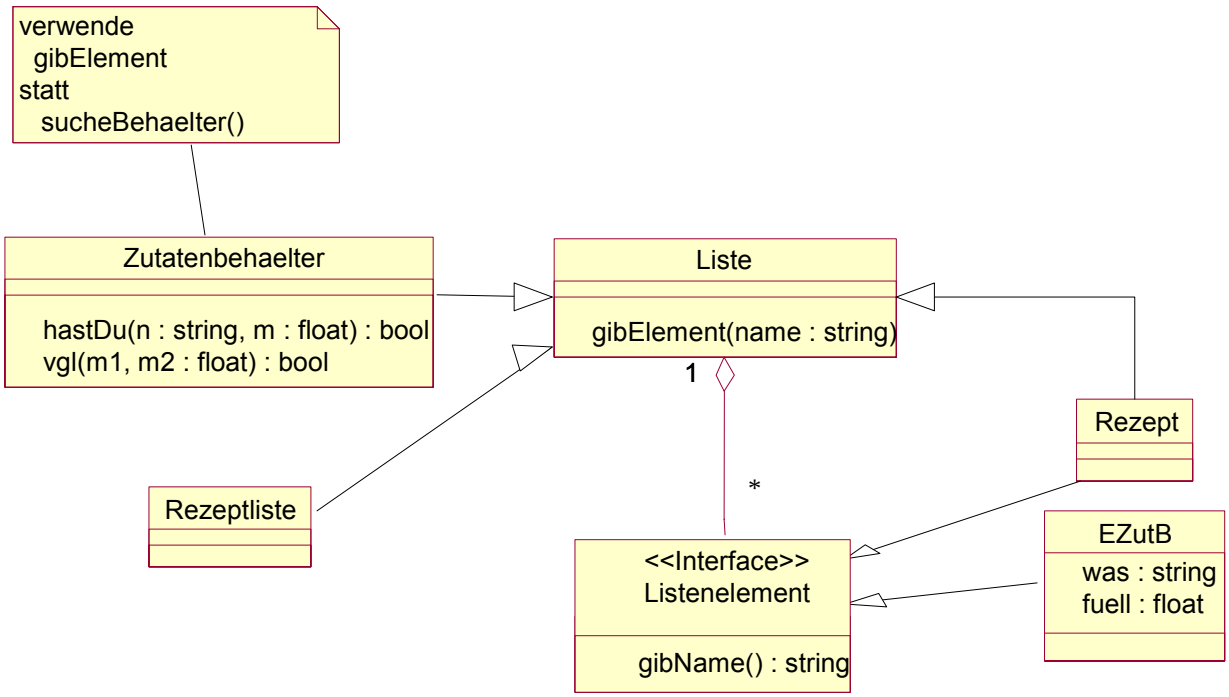
Klassenstrukturdiagramm hieraus



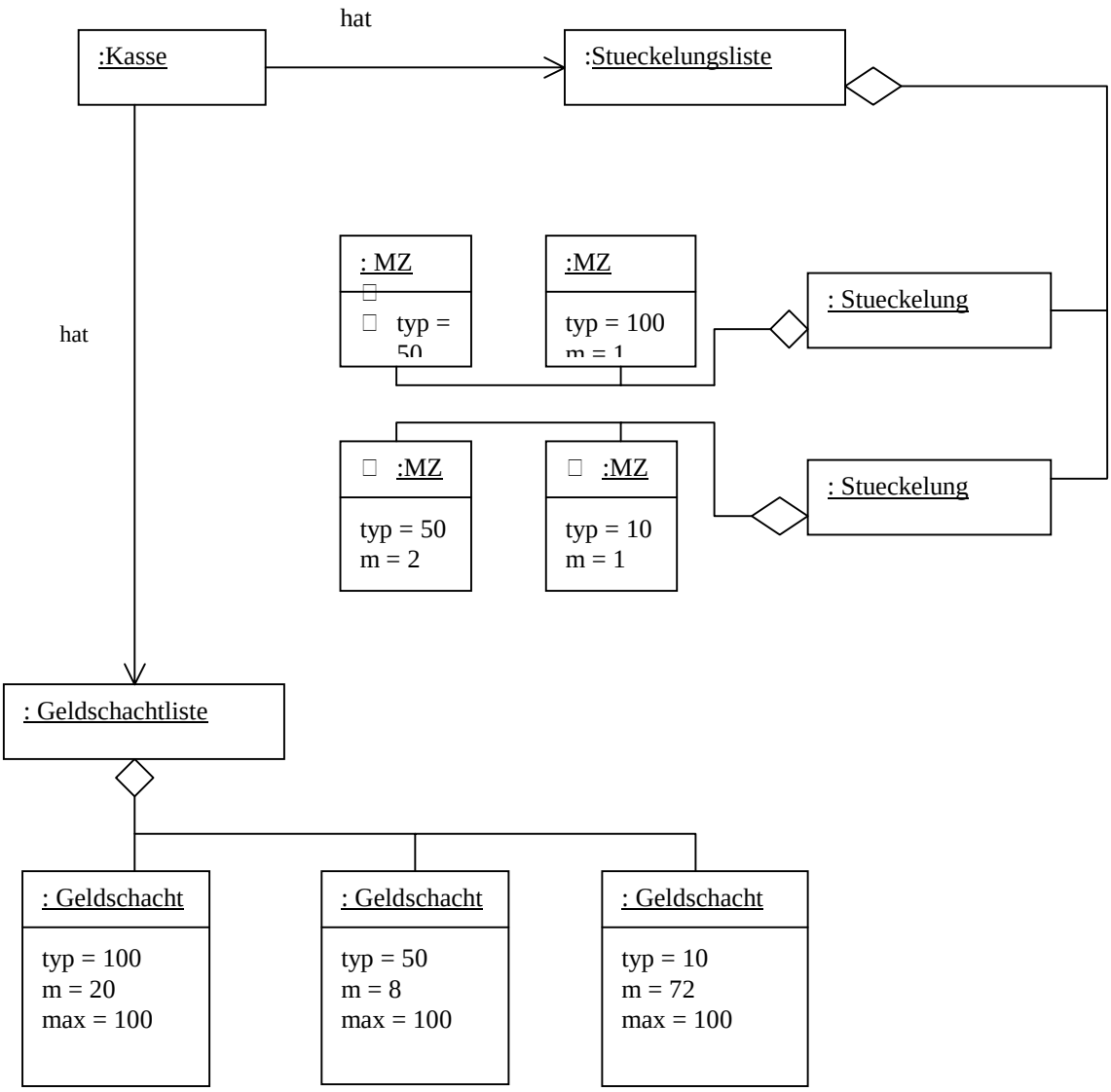


Lösung 26:

Liste kann jetzt beliebige Objekte speichern, sobald diese die Schnittstelle Element implementieren.

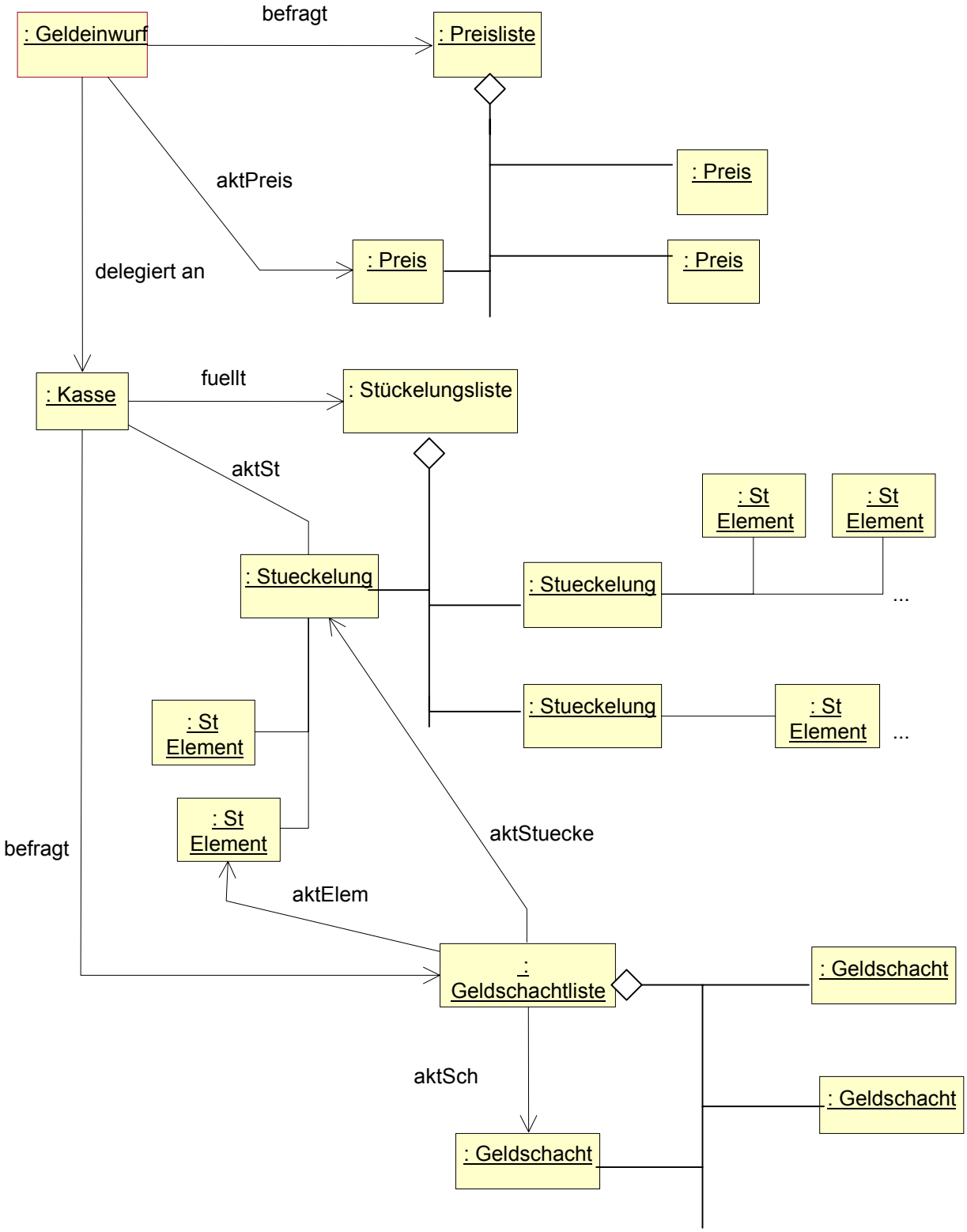


Zusätzliche Überlegung:
Genaugenommen existiert sogar eine Stückelungsliste sowie eine Geldschachtliste (und jede Stückelung ist eine Liste von Münzzahlen).

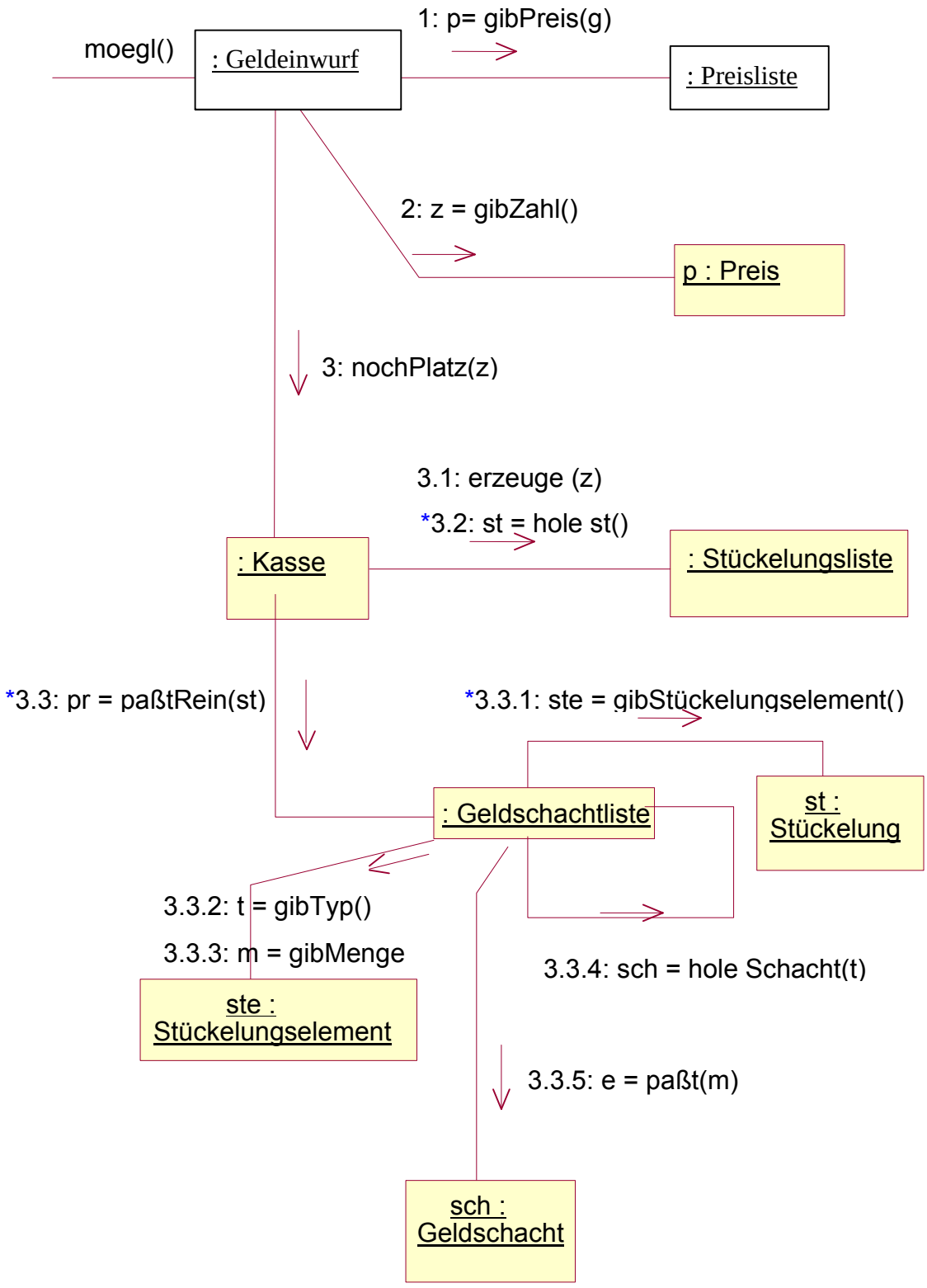


Selbstverständlich sind Stückelungsliste, Geldschachtliste und Stückelung **Spezialisierungen** von Liste;
Stückelung, Münzzahl und Geldschacht implementieren die **Schnittstelle** Element

Zusammengefasst ergibt sich folgendes Objektdiagramm:

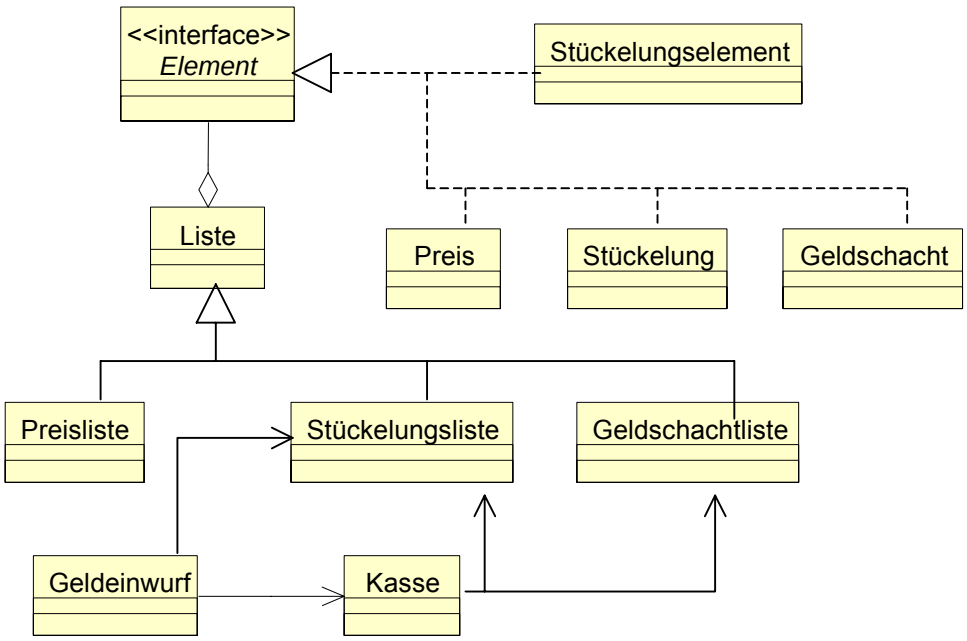


Lösung 27:


moegl()
: Geldeinwurf
1: p= gibPreis(g)
: Preisliste
2: z = gibZahl()
p : Preis
3: nochPlatz(z)
3.1: erzeuge (z)
*3.2: st = hole st()
: Kasse
: Stückelungsliste
*3.3: pr = paßtRein(st)
*3.3.1: ste = gibStückelungselement()
: Geldschachtliste
st : Stückelung
3.3.2: t = gibTyp()
3.3.3: m = gibMenge
3.3.4: sch = hole Schacht(t)
ste : Stückelungselement
3.3.5: e = paßt(m)
sch : Geldschacht

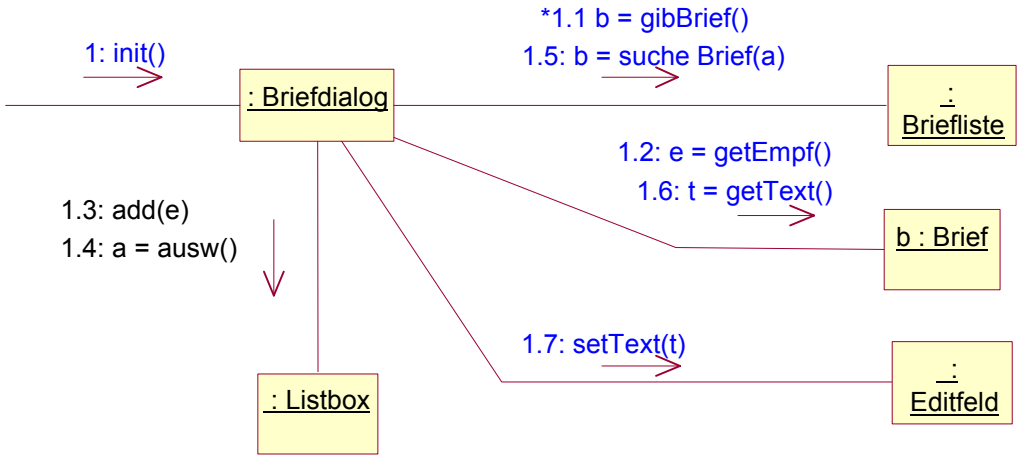
Lösung 28:

Zusammengefaßtes Klassendiagramm

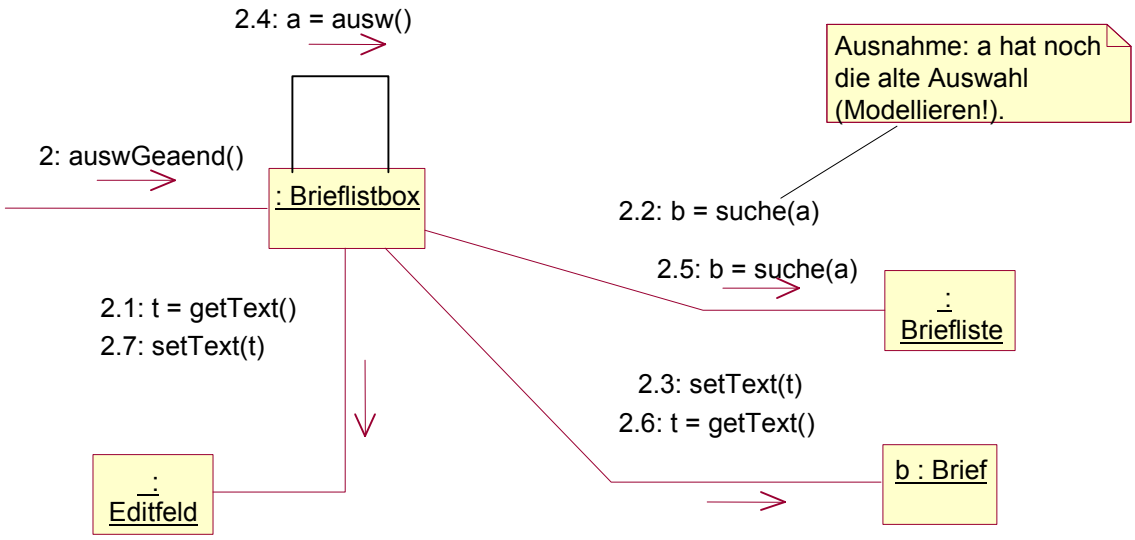


Lösung 29:

1.



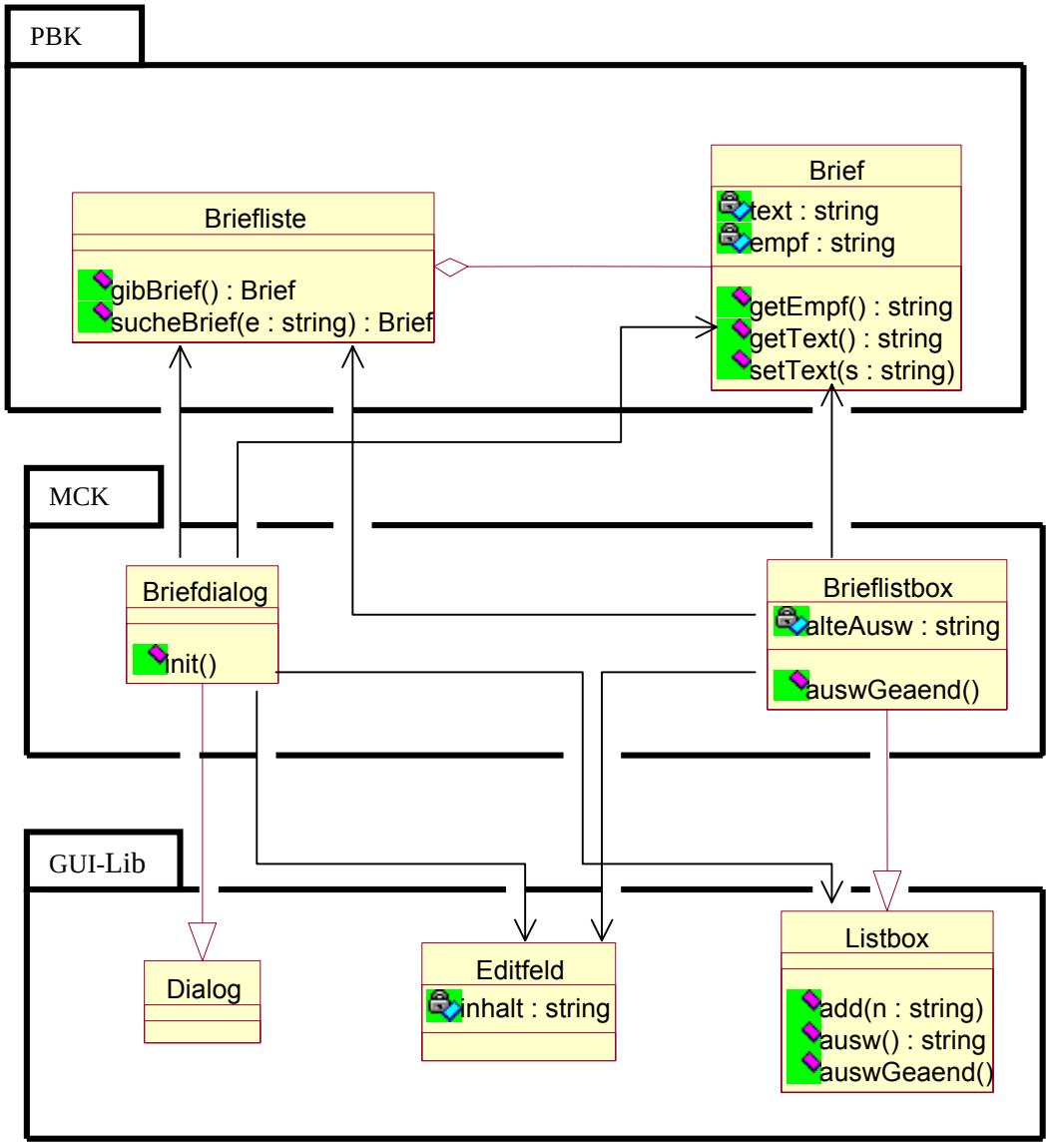
1.: init() ist anwendungsbezogen; also neue Klasse
1.1: Briefliste kann nacheinander alle Briefe liefern, neue Methode
1.2: Brief kann Empfänger nennen, neue Methode
1.5, 1.6, 1.7: neue Methode



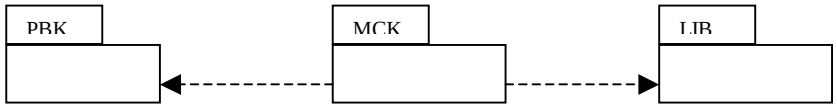
Bemerkung:

* Neue Klasse Brieflistbox, weil neues Verhalten
* 2.1, 2.2, 2.3 haben typischerweise ein anderes (das alte) Brief-Objekt als das in 2.4 gefundene und in 2.5, 2.6,2.7 verwendete (das Neue)!
* 2.4 holt Auswahl des jetzt aktuellen Briefs
* 2.5 bis 2.7 holen die Daten des jetzt aktuellen Briefs

Letzter Schritt: Klassendiagramm hierzu. (Komponenten?)!



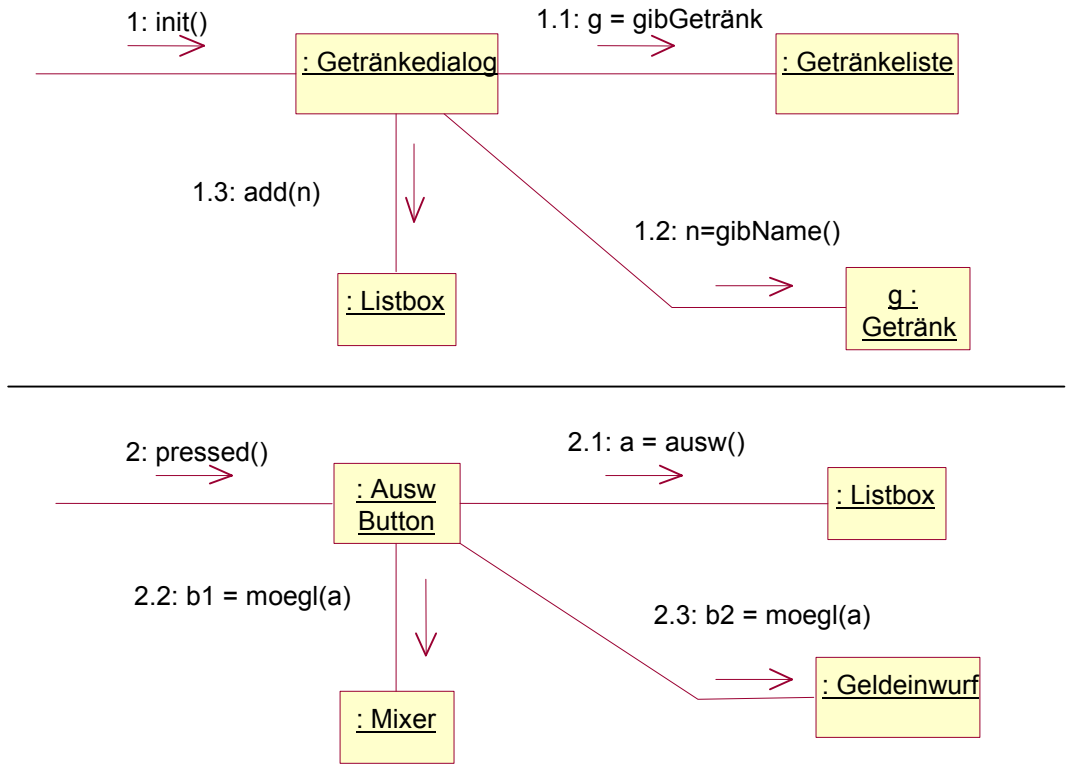
Paketdiagramm hierzu:

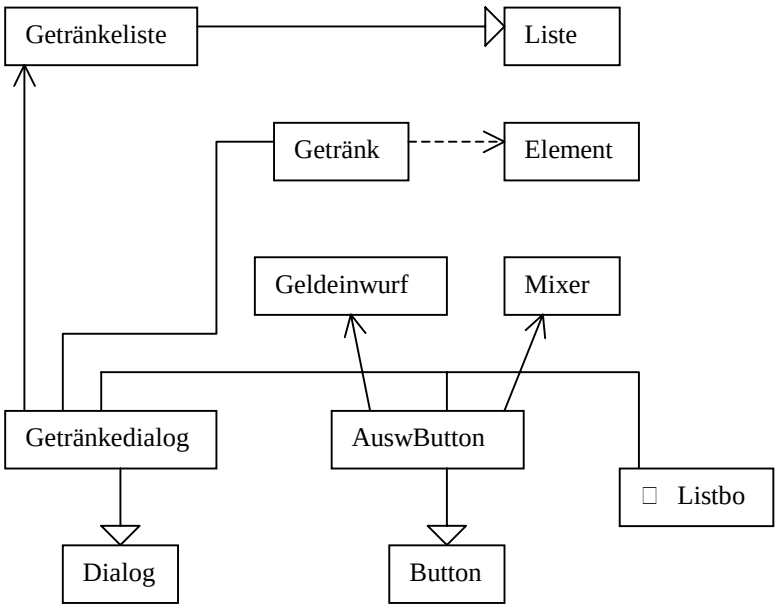


PBK ist vollständig getrennt von der (plattformabhängigen) Lib!

- Übrigens ist hier jetzt nicht dargestellt, dass Briefliste evtl. Spezialisierung von Liste ist und dass Brief die Schnittstelle Listenelement implementiert.

Lösung 30:


1: init()
: Getränkedialog
1.1: g = gibGetränk
: Getränkeliste
1.3: add(n)
1.2: n=gibName()
: Listbox
g : Getränk
2: pressed()
: Ausw Button
2.1: a = ausw()
: Listbox
2.2: b1 = moegl(a)
2.3: b2 = moegl(a)
: Mixer
: Geldeinwurf


Getränkeliste
Liste
Getränk
Element
Geldeinwurf
Mixer
Getränkedialog
AuswButton
Listbo
Dialog
Button

Literaturverzeichnis

[1] Ken Barclay et al., Objektorientiertes Design mit C++, Prentice Hall, 1997

[2] Daniel J. Berg et al., Advanced Techniques for Java Developers, Wiley & Sons, 1999

[3] Grady Booch, Object-oriented Design, Benjamin/Cummings, 1991

[4] Rainer Burkhardt, UML – Unified Modeling Language, Addison-Wesley, 1998

[5] Peter Coad, Edward Yourdon, Object-orientied Design, Yourdon-Press, 1991

[6] Mark Coats et al., Using the CMOSpecification, Dr. Dobb's Journal, June 99

[7] Klaus-Peter Eckert, Objekt-Orientiertheit in offenen Systemen, Thomson, 1995

[8] Igor T. Hawryszkiewycz, Systemanalyse und –Design, Prentice Hall, 1995

[9] Gerti Kappel, Michael Schrefl, Objektorientierte Informationssysteme, Springer, 1996

[10] Robert C. Martin. Objektorientierte C++-Anwendungen, Prentice Hall, 1996

[11] Bertrand Meyer, Objektorientierte Softwareentwicklung, Hanser, 1988

[12] Udo Müller, C++-Implementierungstechniken, Thomson, 1997

[13] Bernd Oesterreich, Objektorientierte Softwareentwicklung, Oldenbourg, 1998

[14] Bernd Oesterreich, Erfolgreich mit Objektorientierung, Oldenbourg, 1999

[15] Objektorientierte Analyse und Design, Siemens Nixdorf Trainings Center, 1996

[16] Objektorientierte Programmierung, Siemens Nixdorf Trainings Center, 1996

[17] Dirk Riehle, Entwurfsmuster für Softwarewerkzeuge, Addison-Wesley, 1997

[18] Jeremy Rosenberger, CORBA, SAMS/Markt&Technik, 1998

[19] Günther Wahl, UML kompakt, OBJEKTspektrum 2/1998

[20] Rebecca Wirfs-Brock et al., Objektorientiertes Software-Design, Hanser, 1993

Stichwortverzeichnis