

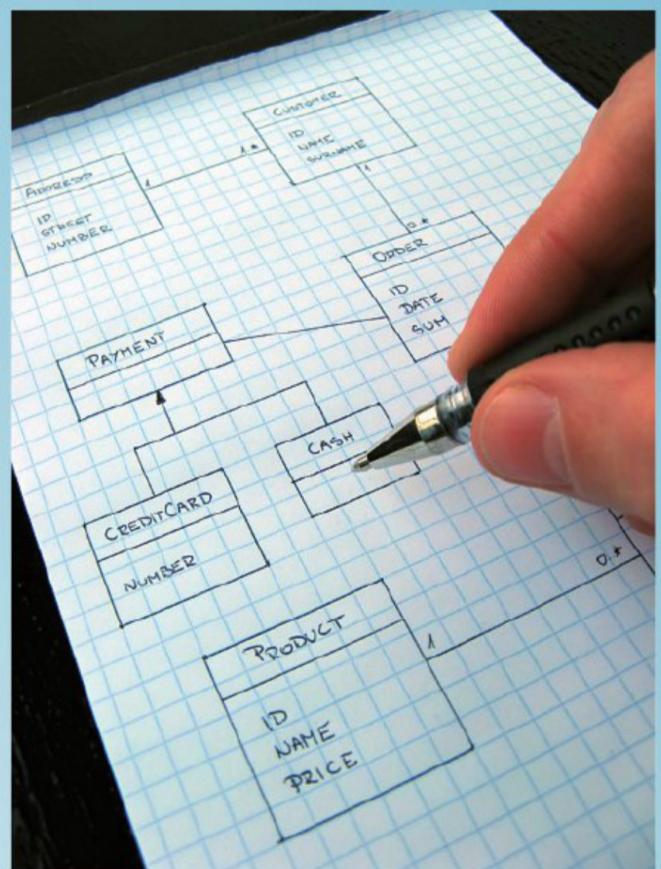
Informatik 1

5. Klassenentwurf

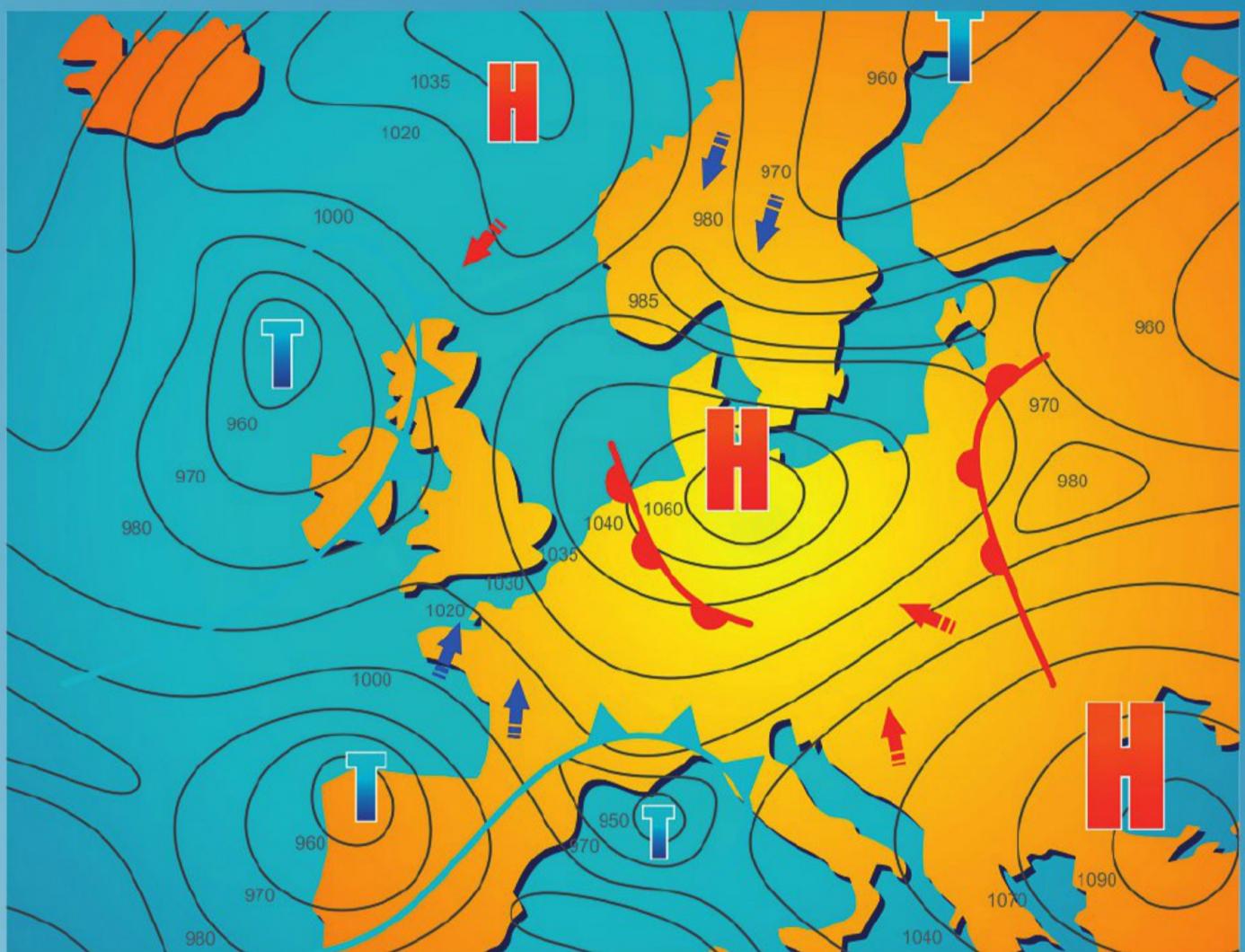


Hier wird nicht jede Blume einzeln programmiert: Es existiert nur eine Klasse *Blume* und durch Parameter für Farbe, Blütenblätterzahl, Durchmesser u.v.m. entstehen unterschiedliche Blumenobjekte.

Objektorientierte Programmierung hätte keinen Sinn, wenn die einzelnen Objekte nicht zusammenarbeiten könnten, um eine Aufgabe zu bewältigen. Damit Objekte Nachrichten untereinander austauschen können, muss eine Beziehung zwischen ihren Klassen modelliert werden. Nur wenn die Klassen und ihre Beziehungen untereinander gut entworfen wurden, kann die Implementation beginnen.



Zu Beginn der Modellierung reicht ein Entwurf, damit Entwickler und Kunden einen Überblick über die Struktur des Softwareprodukts bekommen können.



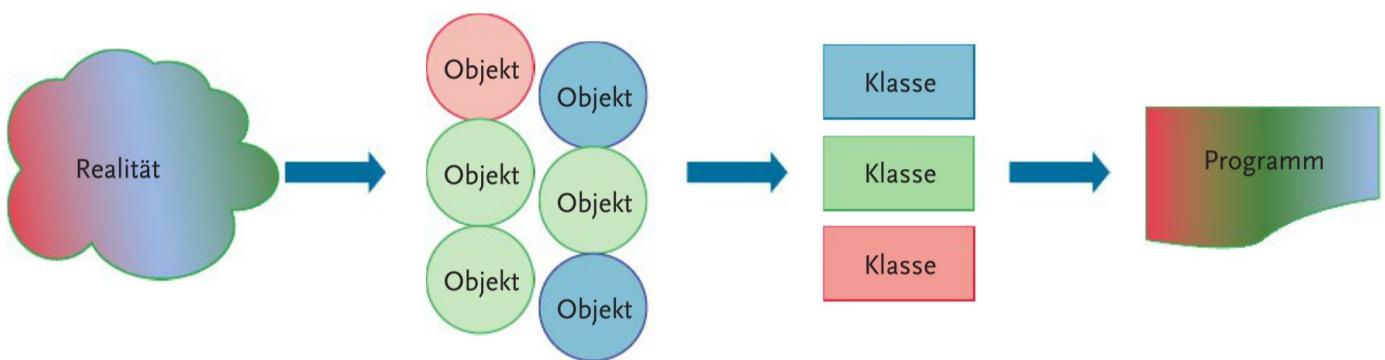
Unsere Welt ist unendlich komplex. In Modellen werden die relevanten Daten strukturiert und gegebenenfalls übersichtlich dargestellt.



In einem Softwaresystem kommunizieren Objekte miteinander: Der Benutzer führt die Geldkarte in den Automaten, der Scanner ermittelt Informationen auf der Karte, die Informationen werden an die Steuerungseinheit weiterleitet ... Welches Objekt wann und wie mit einem anderen kommuniziert, muss daher genau modelliert werden.

5.1 Von der Realität zum Programm

„Die Welt ist voller Objekte“ – Diese Aussage trifft zumindest aus der Sicht von Informatikern zu. In der objektorientierten Programmierung stellen Objekte die relevanten Akteure dar, deren Eigenschaften, Fähigkeiten und Interaktionen den Kern von vielen Computerprogrammen bilden. Die Aufgabe von Informatikern ist es, in einem gegebenen Anwendungsbereich Objekte zu identifizieren, in Form von Klassen und Beziehungen zu modellieren und schließlich mit ihrer Kommunikation zu implementieren.



Da die Realität letztendlich beliebig komplex ist, muss sie für ein Computerprogramm angemessen reduziert werden. Betrachtet man z. B. eine Freunde gruppe, in der alle einen Motorroller besitzen, so könnten die unterschiedlichsten Aspekte dieser Gruppe abgebildet werden. Die Freunde basteln gemeinsam an ihren Fahrzeugen, putzen sie oder bringen sie zur Reparatur in eine Werkstatt. Sie nutzen sie für verschiedene Zwecke, beispielsweise für den Weg zur Schule oder in ihrer Freizeit. In der objektorientierten Modellierung geht es nicht darum, die Realität in all diesen Facetten abzubilden. Vielmehr ist es das Ziel, eine bestimmte Anforderung möglichst genau zu erfüllen.

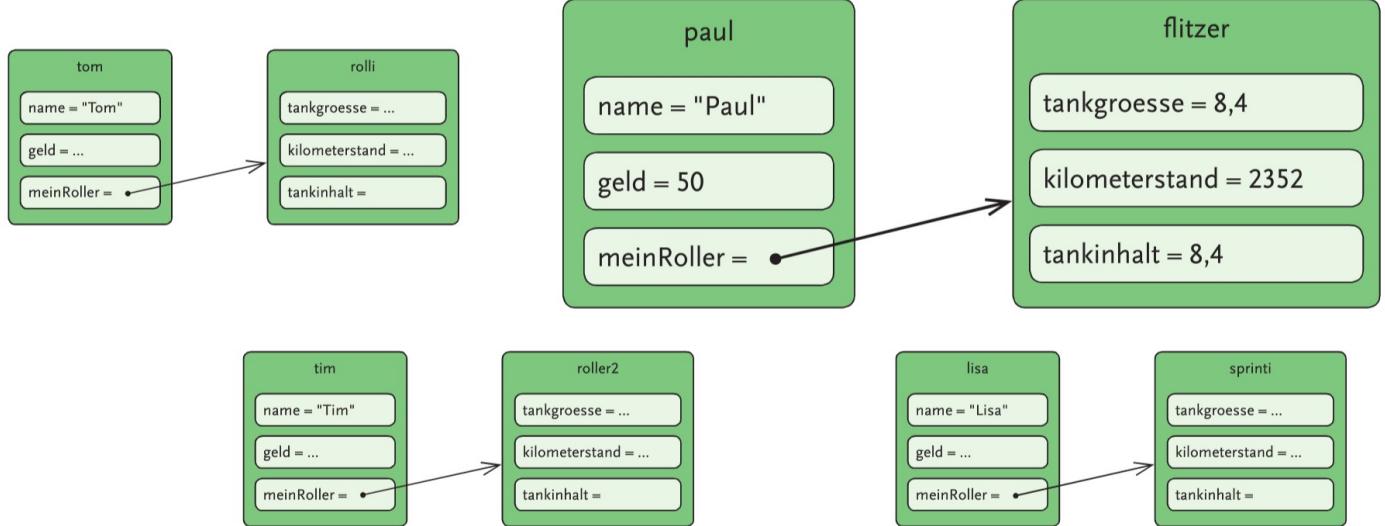


Anforderung

Paul hat zum Geburtstag einen Motorroller bekommen. Da er die Führerscheinprüfung bereits bestanden hat, kann er sofort losfahren. Zu Beginn zeigt der Kilometerzähler einen Stand von 2352 km an und der 8,4-Liter-Tank ist voll. Die Kosten für das Benzin wird Paul in Zukunft von seinem Taschengeld selbst tragen müssen. Paul hat mehrere Freunde, die auch einen eigenen Motorroller besitzen. Es soll eine Anwendung entwickelt werden, in der das Fahren und Betanken der Motorroller simuliert werden kann. Zur Vereinfachung kann davon ausgegangen werden, dass vom Taschengeld eines Rollerbesitzers nur die Benzinkosten getragen werden.

5.2 Objekte

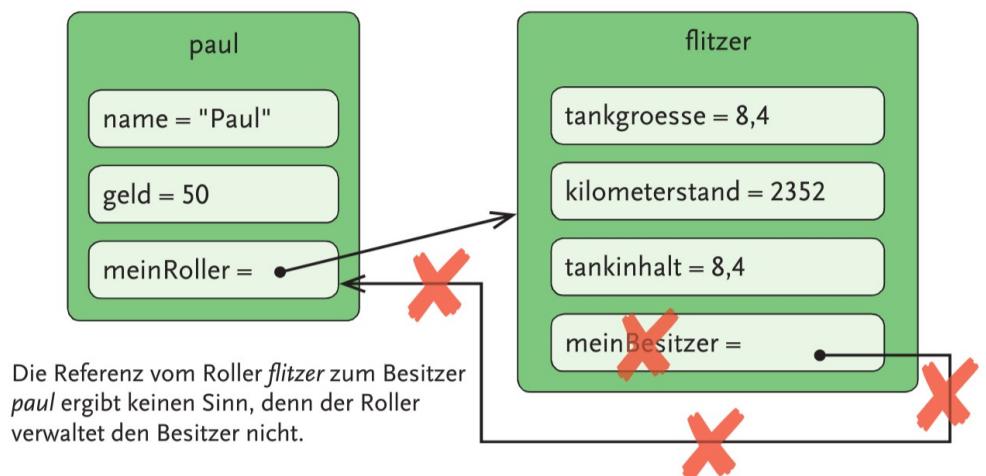
In dieser Situation lassen sich unterschiedliche Objekte und ihre, hinsichtlich der Anforderung wichtigen Eigenschaften identifizieren: zum einen der neue Roller, der einen bestimmten Kilometerstand hat. Auch der Inhalt und die Größe des Tanks sind für die Situation wesentlich, da nur so ein Betanken des Rollers modelliert werden kann. Zum anderen stellt der Besitzer des Rollers, Paul, ein Objekt dieser Situation dar. Er muss über einen bestimmten Geldbetrag verfügen, um den Roller tanken zu können. Darüber hinaus besitzt er ja auch den Roller, was ebenfalls als Eigenschaft gespeichert werden muss. Das Geld und der Tank könnten auch als separate Objekte modelliert werden, da sie aber lediglich eine Eigenschaft eines anderen, wichtigeren Objekts darstellen und in einem Attribut gespeichert werden können, kann hier darauf verzichtet werden. Für Pauls Freunde und ihre Fahrzeuge ergeben sich entsprechende Objekte:



Die aus der Anforderung entwickelten Objekte mit ihren Eigenschaften und konkreten Wertbelegungen. Die Beziehung zwischen dem jeweiligen Besitzer und dem Roller nennt man Objektreferenz.

Jedes Objekt verwaltet bestimmte Informationen, über deren jeweilige Werte sein aktueller Zustand definiert wird. Beispielsweise speichert eine Person ihren Namen und ihren aktuell verfügbaren Geldbetrag. Darüber hinaus soll jedoch auch abgebildet werden, welcher Motorroller dieser Person gehört. Diese Information kann nicht mehr als einfacher Text-, Wahrheits- oder Zahlenwert im Objekt gespeichert werden, sondern wird durch einen Verweis auf das jeweilige Rollerobjekt realisiert. Ein Objekt kann also generell neben Attributen mit primitiven Typen (Text-, Zahlen- oder Wahrheitswerte) auch Attribute dauerhaft verwalten, die einen Objekttyp besitzen. Während die primitiven Typen ihre Werte direkt in den jeweiligen Variablen spei-

chern, verwalten Objekttypen nur eine Referenz auf ein existierendes Objekt. Mithilfe von Objektreferenzen kann also die Zugehörigkeit der einzelnen Fahrzeuge zu den jeweiligen Personen abgebildet werden.



5.3 Klassen und Beziehungen entwerfen

Besitzer
name : Text
geld : Zahl
meinRoller : Roller



Roller
tankgroesse : Zahl
kilometerstand : Zahl
Tankinhalt : Zahl

In einem zweiten Schritt lassen sich aus den identifizierten Objekten Klassen ableiten. Klassen kategorisieren die vorhandenen Objekte, indem sie gleichartige Objekte zu einem Bauplan – der Klasse – zusammenfassen, aus dem sich dann später die tatsächlichen Objekte instanziieren lassen. An dieser Stelle reicht es zuerst einmal aus, ein programmiersprachenunabhängiges, auch für Informatiklaien noch gut nachvollziehbares Diagramm zu entwickeln – das Entwurfsdiagramm. Die Objekte *paul*, *tom*, *lisa* und *tim* verfügen strukturell über dieselben Eigenschaften und können zu einer Klasse *Besitzer* zusammengefasst werden. Für ihre jeweiligen Fahrzeuge stellt die Klasse *Roller* einen geeigneten Bauplan dar. Somit können in dieser Situation die zwei Klassen *Besitzer* und *Roller* modelliert werden.

Die Attribute der Klassen lassen sich direkt aus den Eigenschaften der Objekte entwickeln, es müssen lediglich die konkreten Werte in die Typen *Zahl*, *Text*, *Wahrheitswert* oder den Objekttyp der assoziierten Klasse überführt werden.

Klassen legen neben den Attributen ihrer Objekte auch die Fähigkeiten fest, über die diese verfügen sollen. Es gilt also, zu überlegen, welche Methoden in der gegebenen Situation relevant sind.

Aktionen des Objekts ...	Resultierende Methoden der Klasse ...
flitzer – Eine Strecke fahren – Den Tank füllen	Roller – fahre – tanke
paul – Taschengeld erhalten – Mit dem Motorroller fahren – Den Motorroller auftanken	Besitzer – erhalteTaschengeld – fahreDeinenRoller – tankeDeinenRoller

Des Weiteren werden die Referenzen, die zwischen den einzelnen Objekten bestehen, durch Klassenbeziehungen (Assoziationen) abstrahiert. Eine Assoziation liegt immer dann vor, wenn die Objekte der einen Klasse dauerhaft Objekte der anderen Klasse durch eine Objektreferenz verwalten. Die Objektreferenz ist dabei also nicht abhängig vom Zustand der Objekte, sondern besteht über die gesamte Laufzeit des Programms.

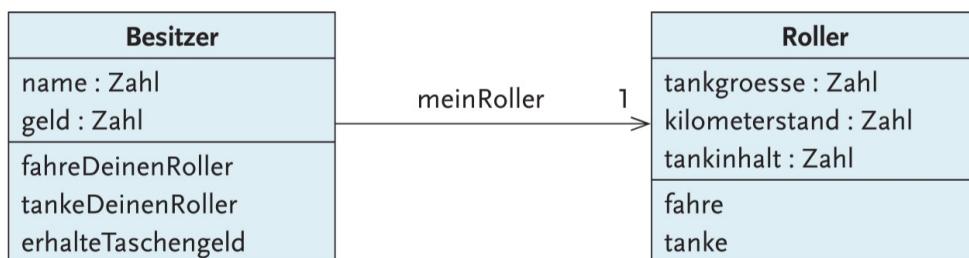
Im Beispiel liegt eine gerichtete Assoziation zwischen den Klassen *Besitzer* und *Roller* vor. Denn jedes Objekt vom Typ *Besitzer* verwaltet dauerhaft ein Objekt vom Typ *Roller*. Bei Assoziationen kann zudem über die Multiplizität spezifiziert werden, zu wie vielen Objekten einer Klasse eine Beziehung besteht. Die Multiplizität der Beziehung zwischen den Klassen *Besitzer* und *Roller* beträgt 1, da jedes Objekt vom Typ *Besitzer* immer auf genau ein Objekt vom Typ *Roller* verweist.

L Eine **Assoziation** zwischen zwei Klassen besteht, wenn Objekte der assoziierenden Klasse Objekte der assoziierten Klasse dauerhaft verwalten. Die Multiplizität einer Assoziation gibt an, wie viele Objekte jeweils assoziiert werden. Dabei wird unterschieden zwischen:

- 1 genau ein assoziiertes Objekt
- 0..1 kein oder ein assoziiertes Objekt
- 0..* beliebig viele assoziierte Objekte
- 1..* mindestens ein, beliebig viele assoziierte Objekte

Die Assoziation wird durch einen geöffneten Pfeil dargestellt. Dieser zeigt von der verwaltenden Klasse zur verwalteten Klasse.

Um die Ergebnisse der Abstraktion von Objekten hin zu Klassen darzustellen, wird in dieser Phase des Klassenentwurfs ein Entwurfsdiagramm angelegt, das unabhängig von einer Programmiersprache die wesentlichen Attribute und Methoden sowie die Beziehungen unter den Klassen beinhaltet. Als Datentypen werden die Typen *Zahl*, *Text* und *Wahrheitswert* unterschieden. Anfragen werden durch die Angabe ihres Rückgabetyps spezifiziert.



Ein Entwurfsdiagramm für die Roller-Simulation

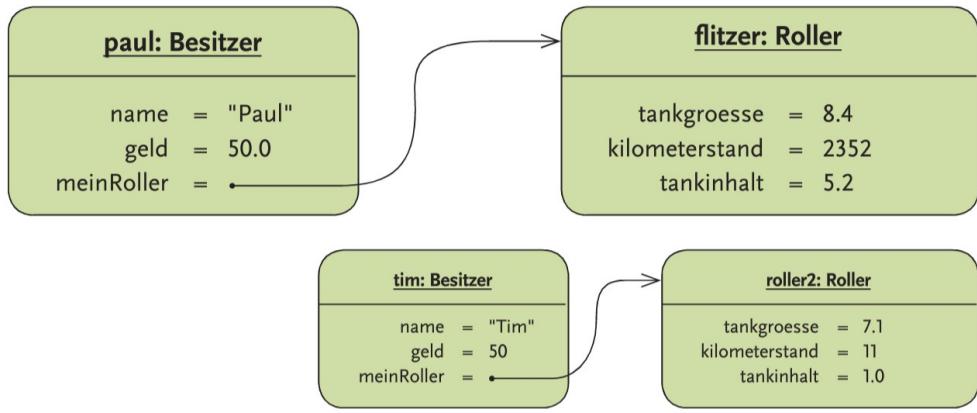
Eine Assoziation wird im Entwurfsdiagramm durch einen Pfeil dargestellt. Der Pfeil sollte mit dem Bezeichner des Bezugsobjektes beschriftet werden. Am Ende des Pfeils wird die Multiplizität der Assoziation angegeben.

Durch das Entwurfsdiagramm wurde eine Modellierung der generellen informatischen Struktur der Anforderung erreicht. Diese Sichtweise ist erforderlich, um zu überprüfen, ob alle relevanten Bereiche der Anforderung in ein Modell einbezogen wurden. Erst auf dieser Grundlage kann die eigentlich Programmierung aufsetzen.

Objektdiagramme stellen immer nur eine Momentaufnahme der Situation dar. Zu einem späteren Zeitpunkt kann der Zustand der Objekte ein anderer sein, z. B. könnte der Kilometerstand eines Motorrollers ein höherer sein. Klassendiagramme hingegen bieten eine statische Sicht auf das System und seine generelle Struktur.

Objekt- und Klassendiagramme werden zusätzlich im Methodenanhang erläutert.

Möchte man hingegen die Abläufe im System planen und verstehen, ist es sinnvoll, auf die Ebene der Objekte zu wechseln. Ein Objektdiagramm zeigt, welche Objekte zu einem bestimmten Zeitpunkt während der Ausführung eines Programms existieren, wie ihre Werte belegt sind und welche Referenzen unter ihnen bestehen.



Ein Objektdiagramm für die Roller-Simulation

→ Aufgabe 1, 2

Aufgaben

1. Leeren Sie mit einem Partner Ihre Etuis und/ oder Portemonnaies und identifizieren Sie mindestens fünf unterschiedliche Objekte.
 - a) Zeichnen Sie jeweils ein Objektdiagramm mit passenden Attributen und Wertbelegungen.
 - b) Notieren Sie an einem der Diagramme die Namen der Bestandteile des Objektdiagramms.
 - c) Sortieren Sie die Objekte so, dass – auch unter Einbeziehung bisher unberücksichtigter Objekte aus Etui und/oder Portemonnaie – gleichartige Objekte nebeneinander liegen. Begründen Sie dabei Ihre Sortierentscheidungen.
 - d) Entwickeln Sie jeweils ein passendes Entwurfsdiagramm zu den ähnlichen Objekten. Die Methoden können vernachlässigt werden.
 - e) Stellen Sie Ihre Ergebnisse den anderen Gruppen in einem Gallery Walk vor. Diskutieren Sie anschließend Ihre Ergebnisse.



2. Folgendes Entwurfsdiagramm ist gegeben:

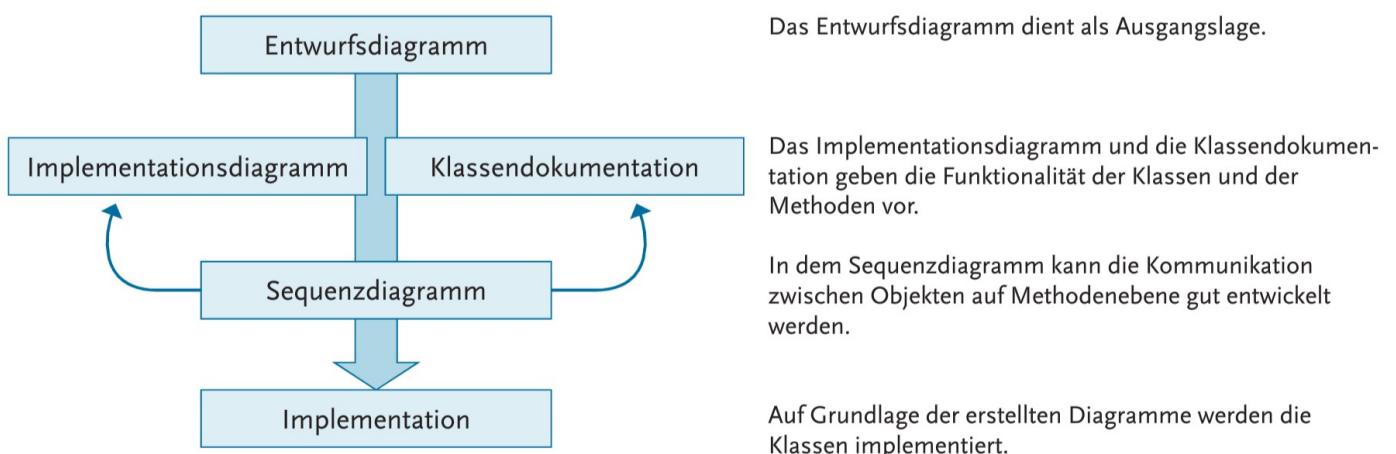
- a) Benennen Sie die Bestandteile des Entwurfsdiagramms und erläutern Sie den Unterschied zu einem Objektdiagramm.
- b) Zeichnen Sie drei unterschiedliche Objektdiagramme zu dem Entwurfsdiagramm.
- c) Die Anforderung der einfachen Zeitanzeige soll um eine Weckerfunktion und eine Stoppuhr erweitert werden. Erweitern Sie das Entwurfsdiagramm entsprechend und begründen Sie Ihre Entscheidungen.

UhrDigital
stunde : Zahl
minute : Zahl
batteriestand : Zahl
minuteVor
stundeVor

- d) Entwickeln Sie eine Klasse *UhrAnalog*, deren Zeitanzeige über den Drehwinkel des Stunden-, Minuten- und Sekundenzeigers geregelt wird. Erstellen Sie dazu passend zwei Objektdiagramme.
3. Zeichnen Sie zu folgenden Begriffsgruppen ein Entwurfsdiagramm, in denen nur die Assoziationen inklusive Multiplizitäten vorkommen sollen (ohne weitere Attribute oder Methoden).
- Ofen – Spüle – Küche
 - Beamer – Fernbedienung
 - Lenker – Fahrrad
 - Ehemann – Ehefrau
 - Personalausweis – Bürger
 - Partei – Parteivorsitzender – Parteimitglied
 - Computer – Drucker – Maus – Tastatur

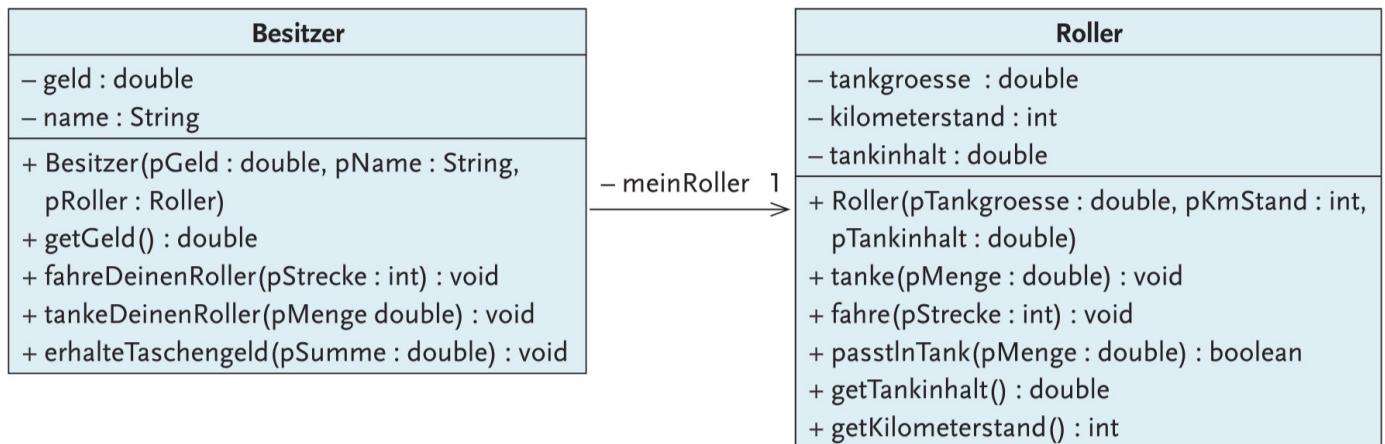
5.4 Klassen und Beziehungen implementieren

Ausgehend vom noch relativ groben und auch von Laien nachvollziehbaren Entwurfsdiagramm folgt der Schritt der Implementierung. Vor der eigentlichen Programmierung wird das Entwurfsdiagramm noch in ein programmiersprachenspezifisches Implementationsdiagramm überführt.



Das aus einem Entwurfsdiagramm entwickelte Implementationsdiagramm stellt dann die Grundlage für eine Umsetzung der Klassen in einer Programmiersprache dar. Dies läuft in folgenden Schritten ab:

- Notieren der programmierspezifischen Methoden
- Verfassen der Dokumentation der Methoden inklusive der Signatur
- Abilden der Objektkommunikationen mithilfe von Sequenzdiagrammen



Ein Implementationsdiagramm für die Roller-Simulation auf der Grundlage des Entwurfsdiagramms

Programmiersprachenspezifische Methoden

Ein Entwurfsdiagramm ist universell, das Implementationsdiagramm muss jedoch in diesem Fall auf die Programmiersprache Java zugeschnitten werden. Dazu muss geregelt werden, wie Objekte erzeugt werden und wie der lesende und ändernde Zugriff auf die Objektattribute geregelt wird. In der Programmiersprache Java müssen der Konstruktor sowie die get- und set-Methoden definiert werden.

H

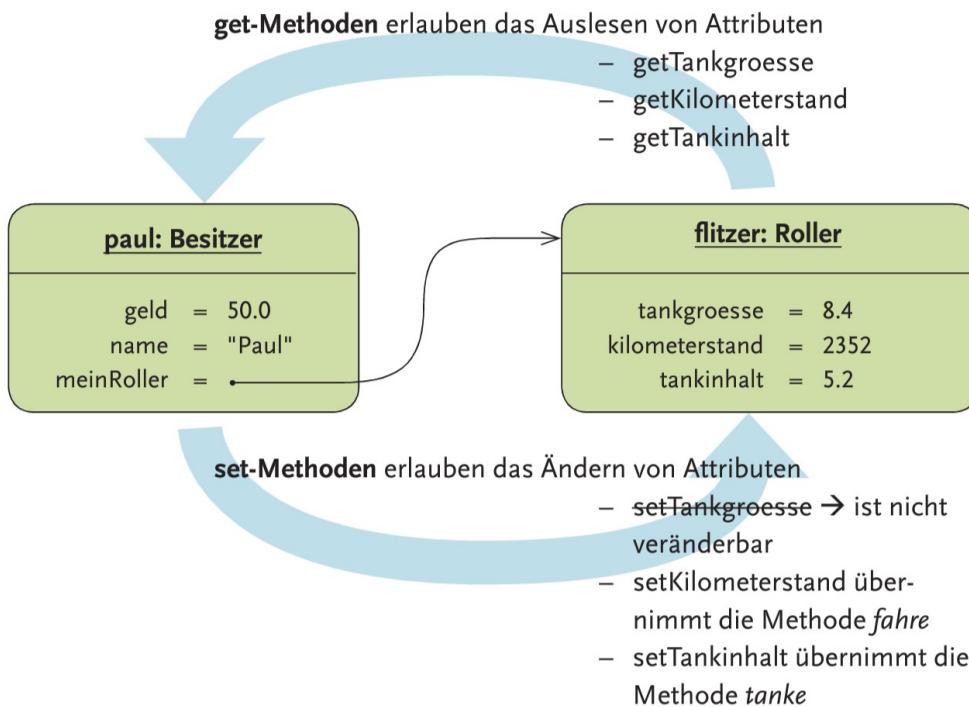
Die auf den folgenden Seiten vorgestellten Bestandteile der Implementationsmodellierung (Dokumentation, Sequenzdiagramm und Implementationsdiagramm) können im Ablauf der Softwareentwicklung nicht immer klar voneinander getrennt werden. Häufig läuft ihre Entwicklung zeitgleich und eine Entscheidung in einem der Bestandteile hat sofort Rückwirkungen auf eventuell vorher getroffene Entscheidungen.

Da der Konstruktor Objekte der Klasse erzeugt, muss in der Regel nur geklärt werden, welche Attributwerte bei der Objekterzeugung bei allen Objekten gleich sein sollen und welche nicht. Danach entscheidet sich, welche Attribute per Konstruktorparameter festgelegt werden und welche nicht.

Aufbau eines Konstruktors

```
public Klassenname (attributstyp01 param01)
{
    attributname01 = param01;
    attributname02 = festerWert;
}
```

Die Verantwortung des Auslesens und des Änderns der Attributwerte soll zu jeder Zeit beim Objekt selbst liegen. Daraus folgt, dass jede Klasse explizit erlauben muss, auf ihre Attribute lesend oder ändernd zuzugreifen. Dies erfolgt über sondierende (get-)Methoden und verändernde (set-)Methoden.



H Ändernde und lesende Methoden können auch schon im Entwurfsdiagramm vorkommen, wenn ihre Aufgabe Kernbestandteil der Klasse ist.

Aufbau einer get-Methode

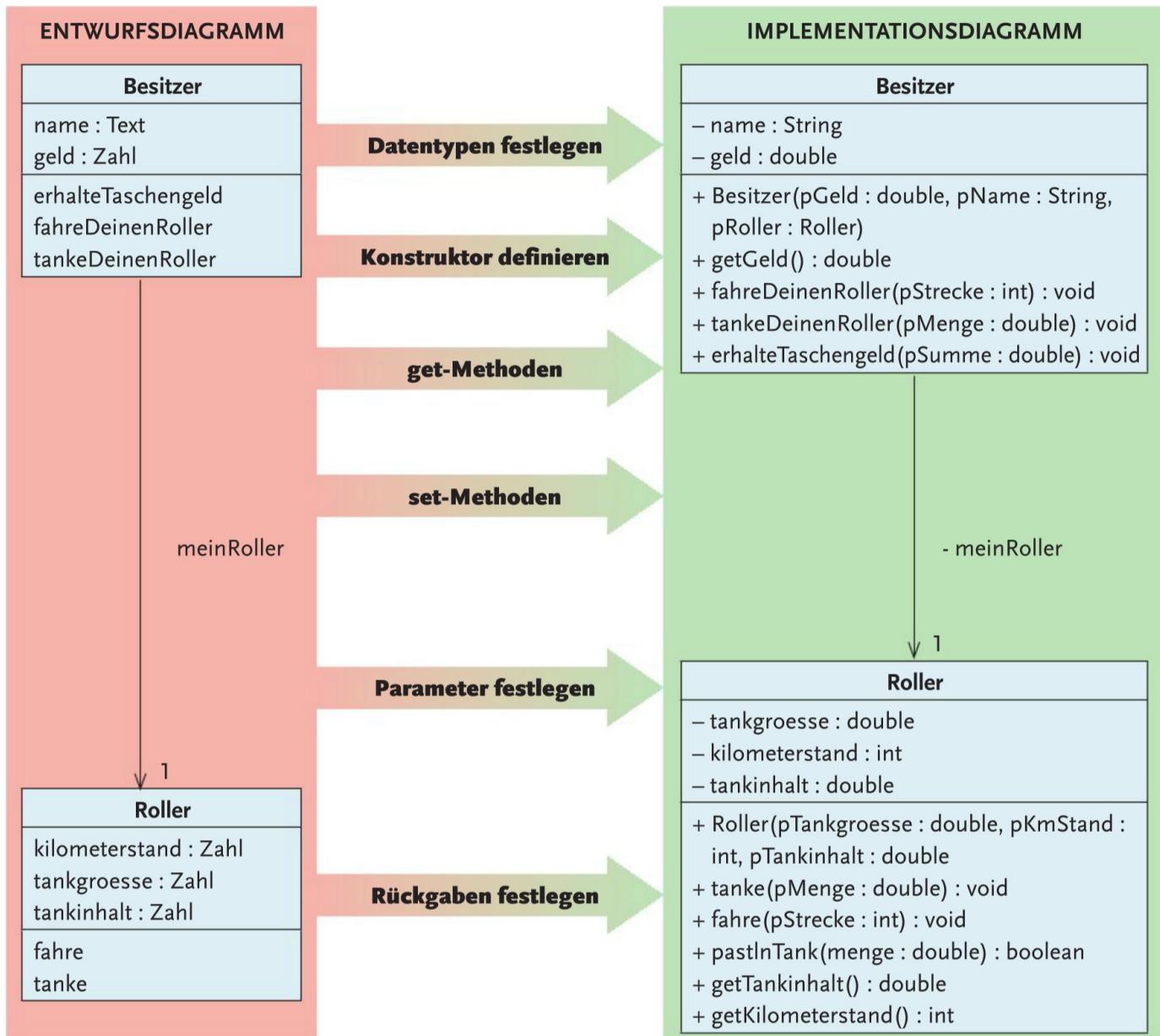
```

public attributstyp getAttributname()
{
    return attributname;
}
  
```

Aufbau einer set-Methode

```

public void setAttributname (attributstyp param)
{
    attributname = param;
}
  
```



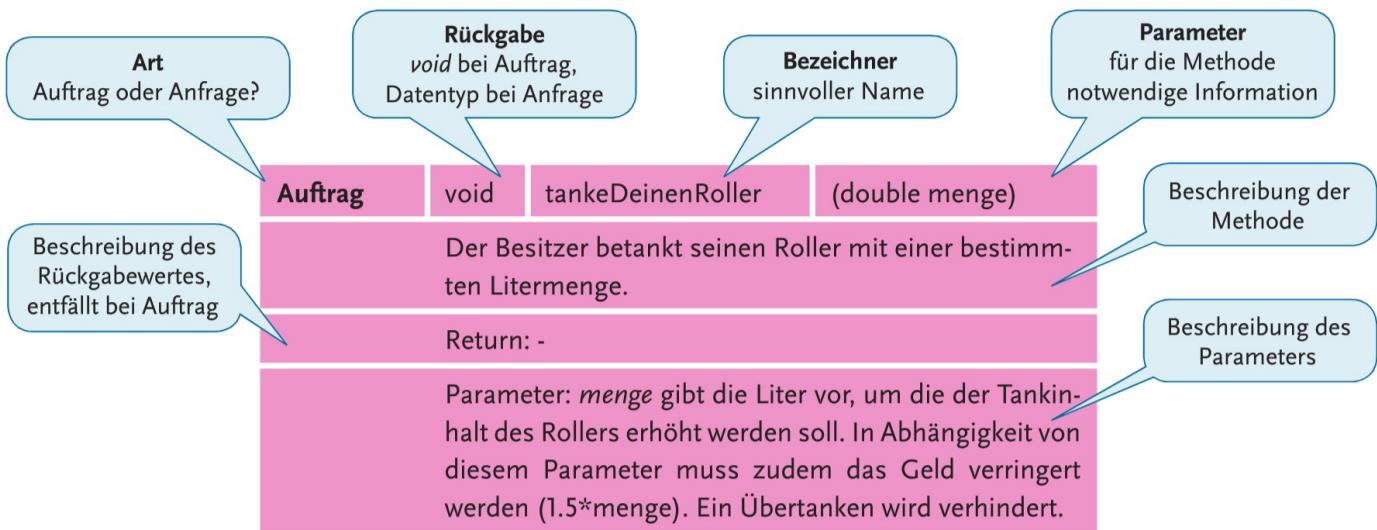
Die Überführung des Entwurfsdiagramms in ein Implementationsdiagramm für Java

→ Aufgabe 1

Klassendokumentation

Die Dokumentation der Klasse besteht aus einer kurzen Funktionsbeschreibung der Klasse und den Methodendokumentationen. Die Klassendokumentation gehört im Prinzip nicht zum Implementationsdiagramm, ohne jedoch eine Vorstellung von der Aufgabe einer Methode zu haben, kann deren Signatur mit Rückgabetypr und Parameter nicht konkretisiert werden.

Sollten Objekte anderer Klassen beteiligt sein, muss auch schon der nächste Punkt, Objektkommunikation, beachtet werden. Die Methodendokumentation erfolgt wie in Kapitel 3 beschrieben.



Die Dokumentation von Klassen dient neben der Erleichterung der Arbeit des Programmierers dazu, Klassen auch für andere Benutzer nutzbar zu machen. Da sich allein aus dem Methodennamen die Funktionalität einer Methode in der Regel nicht erschließen lässt, benötigt man eine Beschreibung der Methode über die Rückgabe, die Parameter und die Ergebnisse der Methode. Eine Dokumentation ist allerdings keine Beschreibung oder Erläuterung des Quellcodes. Vielmehr geht es gerade darum, die Klasse und ihre Methoden anzuwenden, ohne den Quellcode zu kennen.

Die Dokumentation sollte zur Information des Programmierers direkt im Quellcode notiert werden; sie lässt sich bei korrekter Formatierung dann aber auch im sogenannten Javadoc-Format als HTML-Datei anzeigen.

```
/**  
 * ein- oder mehrzeiliger Kommentar  
 */
```

Aufbau einer Javadoc-fähigen Kommentierung

Der erste Kommentar beschreibt die grundlegende Funktionalität der Klasse. Die reservierten Schlüsselwörter `@author` und `@version` leiten die Angabe des Autors und der Version ein.

```
/**  
 * Beschreibung der Klasse  
 * @author Thomas Kempe  
 * @version 23.02.2014  
 */
```

Diese Informationen müssen vor der eigentlichen Klasse stehen.

```
/**  
 * Beschreibung der Methode  
 * @param Wirkung des Parameters  
 * @return Beschreibung der Rückgabe  
 */
```

Fehlen Parameter oder Rückgabe, reicht ein „–“ statt der Beschreibung.

H Die gesamte Klassendokumentation findet sich in den Quelltexten ab S. 110.

→ Aufgabe 2

Objektkommunikation definieren

Damit Objektkommunikation stattfinden kann, benötigt ein Objekt die **Referenz** (Verweis) auf ein anderes Objekt. Entweder muss eine dauerhafte Referenz über eine Assoziation oder eine temporäre als Parameterübergabe gegeben sein.

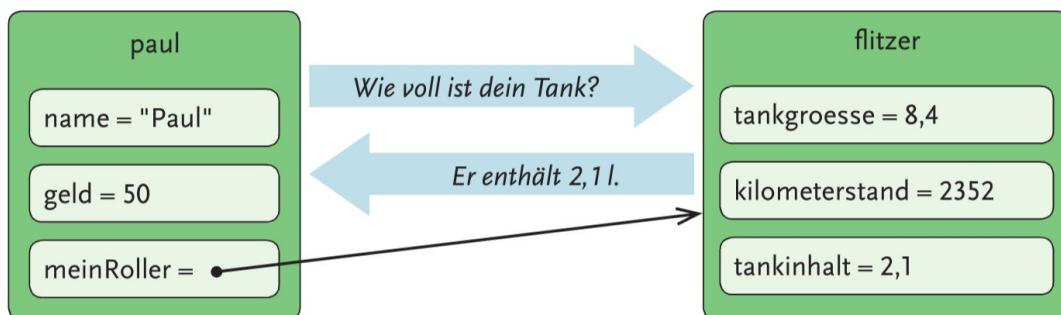
L

Ohne eine Interaktion zwischen den einzelnen Objekten wäre ein objektorientiertes Programm letztendlich nicht funktionsfähig. Objekte müssen untereinander kommunizieren, indem sie entweder Aufträge oder Anfragen anderer Objekte aufrufen.

Damit eine Kommunikation zwischen zwei Objekten funktioniert, müssen zwei Voraussetzungen gegeben sein:

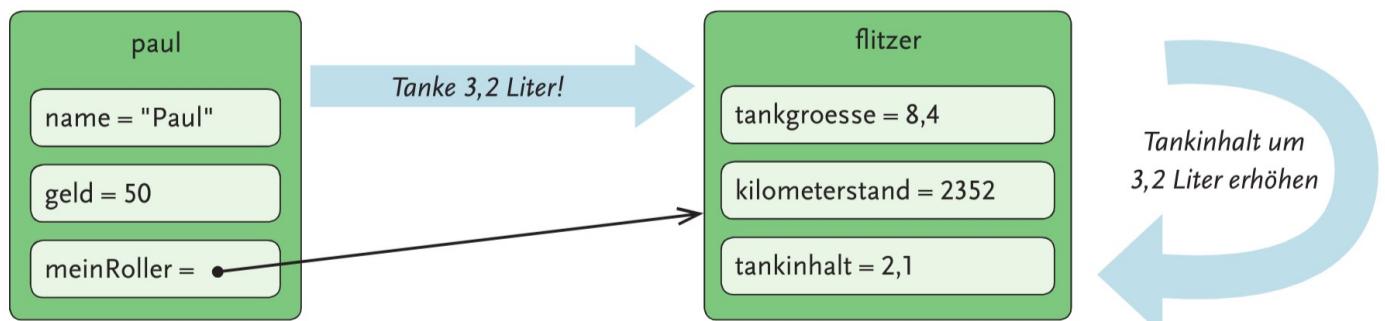
1. Das sendende Objekt benötigt eine Objektreferenz zum empfangenden Objekt.
2. Das empfangende Objekt muss geeignete Methoden (Anfragen oder Aufträge) zur Verfügung stellen, die das sendende Objekt sinnvoll benutzen kann.

Ein Beispiel für eine Anfrage, die das Objekt *paul* an seinen Motorroller stellen kann, ist die Information über den Tankinhalt. Das Objekt *flitzer* muss dazu eine Methode bereitstellen, die den Tankinhalt als Rückgabewert liefert. Dieses wird klassischerweise über eine get-Methode realisiert.



Das Objekt *paul* stellt eine Anfrage an das Objekt *flitzer* und erhält eine entsprechende Antwort.

Das sendende Objekt kann auch Aufträge an das empfangende Objekt erteilen, so z. B. den Auftrag, zu tanken. Voraussetzung dafür ist natürlich wiederum, dass das Objekt *flitzer* eine geeignete Methode für das Fahren zur Verfügung stellt.

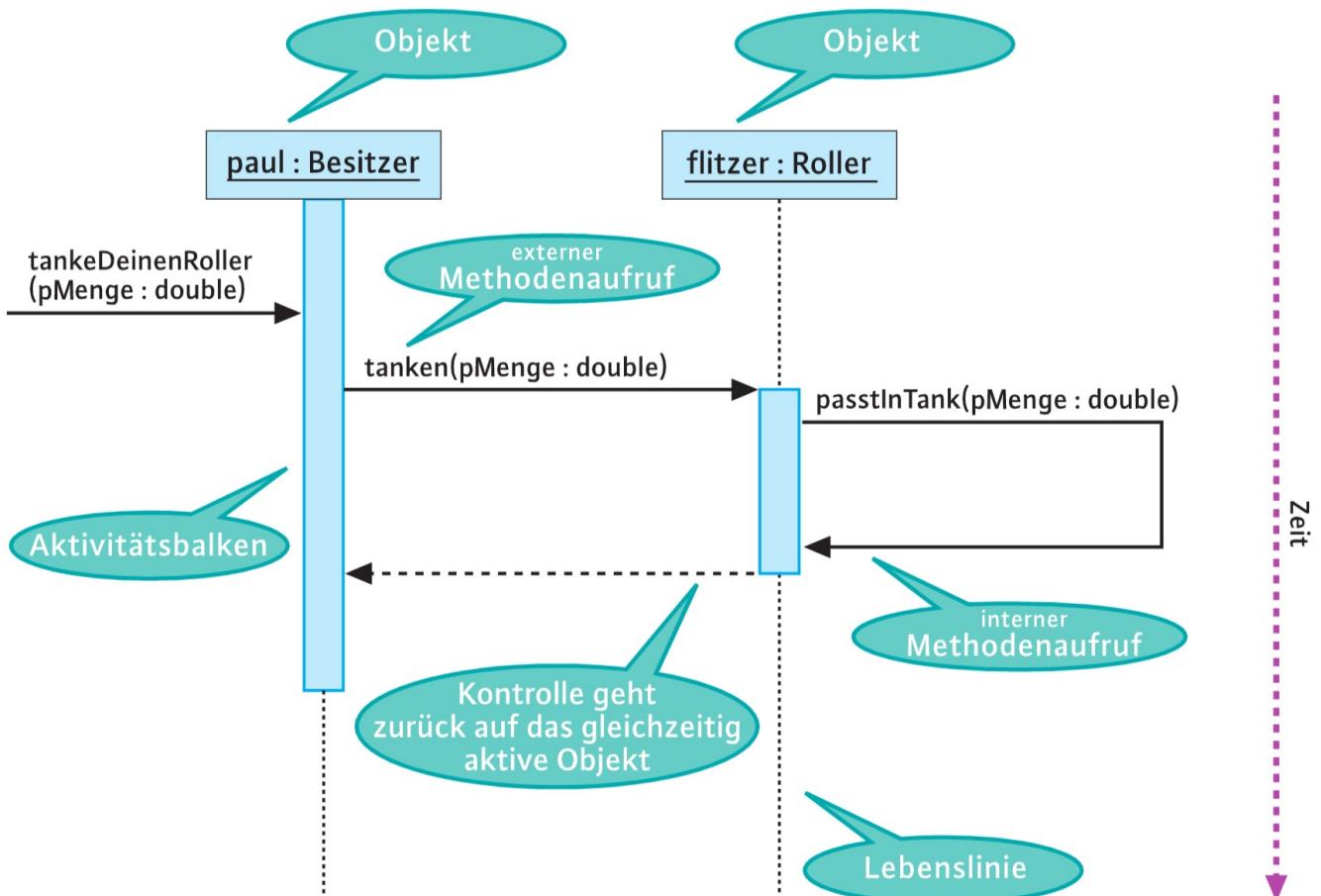


Das Objekt *paul* gibt einen Auftrag an das Objekt *flitzer* und löst damit bestimmte Aktionen des Objektes aus.

Mit einem Sequenzdiagramm lässt sich die Objektkommunikation übersichtlich darstellen. Dies ermöglicht die Vorwegnahme entscheidender Bestandteile der eigentlichen Implementation.

Sequenzdiagramme sind wie Klassendiagramme ein Bestandteil des Unified-Modeling-Language-Pakets. Mit diesen ist es möglich, dynamische Beschreibungen von Aktivitäten zwischen Objekten darzustellen.

Das Sequenzdiagramm stellt zum einen die beteiligten Objekte dar und ordnet ihnen auf einer Zeitlinie bzw. Lebenslinie bestimmte Kommunikationsaktivitäten in Form von Methodenaufrufen zur Darstellung eines bestimmten Ablaufs zu. Die Kommunikation erfolgt dabei durch die Übergabe von Parametern und Rückgabewerten. Durch die parallele Anordnung der beteiligten Objekte und zwischen ihnen angeordneter Kommunikationspfeile kann ein exemplarischer Ablauf dargestellt werden, wobei die innere Differenzierung einer Methode durch z. B. bedingte Anweisungen nicht abgebildet werden muss.



Der Besitzer `paul` ist während des gesamten Ablaufs `tankeDeinenRoller` aktiv. Da sich beim Tanken das Attribut `tankinhalt` des Rollers `flitzer` um die getankte Menge `menge` verändern muss, beauftragt der Besitzer das Roller-Objekt, die Methode `tanke` aufzurufen. In diesem Moment wird das Objekt `flitzer` aktiv. Nach der Beendigung der Methode `tanke` des Objekts `flitzer` findet ein Rücksprung statt.

→ Aufgabe 3

```

public void tankeDeinenRoller(double pMenge)
{
    ...
    meinRoller.tanke (menge) ;
    ...
}

public void tankeRollerVonFreund(Roller andererRoller, double pMenge)
{
    ...
    andererRoller.tanke (menge) ;
    ...
}

```

Möglichkeit I

Die Objektreferenz wird direkt über das Attribut *meinRoller* hergestellt.

Möglichkeit II

Die Objektreferenz wird über einen Parameter hergestellt.

Mit Entwicklungsumgebungen wie dem Java-Editor kann die gesamte Modellierung des Implementationsdiagrammes per Maus zusammengeklickt werden. Die fehlenden Algorithmen können dann im zugehörigen Editor programmiert werden.

H

Nach der gesamten Vorarbeit über die Objektidentifikation, der Entwurfsdiagrammerstellung, der Ableitung des Implementationsdiagramms inklusive Klassendokumentation (diese findet sich hier in der Implementation) und der Erstellung von Sequenzdiagrammen bei Objektkommunikationen folgt erst im letzten Schritt die eigentliche Programmierung. Bei dieser müssen lediglich die Algorithmen in die „Methodenhüllen“ geschrieben werden.

→ Aufgabe 4

Implementation der Klasse *Besitzer*

```

/**
 * Der Besitzer eines Rollers kann diesen fahren und ihn betanken, sofern er
 * genug Geld besitzt.
 * @version 1.0 vom 25.04.2014
 * @author Thomas Kempe
 */

public class Besitzer {
    // Anfang Attribute
    private String name;
    private double geld;
    private Roller meinRoller;
    // Ende Attribute

    // Anfang Methoden
    /**
     * Der Konstruktor erzeugt ein Objekt der Klasse, wobei alle Attribute per
     * Parameterübergabe initialisiert werden.
     * @param pName Gibt den Startwert des Attributs name an.
     * @param pGeld Gibt den Startwert des Attributs geld an.
     * @param pRoller Ein Objekt der Klasse Roller wird als meinRoller referen-
     * ziert.
     */

```

```
public Besitzer (String pName, double pGeld, Roller pRoller) {  
    name = pName;  
    geld = pGeld;  
    meinRoller = pRoller ;  
}  
  
/**  
 * Die sondierende Methode für das Attribut name  
 * @return Das Attribut name wird zurückgegeben.  
 */  
public String getName() {  
    return name;  
}  
  
/**  
 * Die sondierende Methode für das Attribut geld.  
 * @return Das Attribut geld wird zurückgegeben.  
 */  
public double getGeld() {  
    return geld;  
}  
  
/**  
 * Der Tankinhalt des Objekts meinRoller wird erhöht und das Geld wird (der  
 * Einfachheit halber) um pMenge reduziert.  
 * @param pMenge Die zu tankende Menge.  
 */  
public void tankeDeinenRoller(double pMenge) {  
    if ((pMenge * 1.5 <= geld)  
    {  
        meinRoller.tanke(pMenge);  
        geld = geld - (pMenge * 1.5);  
    }  
}  
/**  
 * Der Kilometerstand des Objekts meinRoller wird erhöht.  
 * @param pStrecke Gibt die zu fahrende Strecke an.  
 */  
public void fahreDeinenRoller(int pStrecke) {  
    meinRoller.fahre(pStrecke);  
}  
  
/**  
 * Das Geld des Besitzers wird erhöht.  
 * @param pSumme Der Wert, um den geld erhöht wird.  
 */  
public void erhalteTaschengeld(double pSumme) {  
    geld = geld + pSumme;  
}  
}
```

Implementation der Klasse *Roller*

```

/**
 *
 * Die Klasse Roller ist das Fahrzeug, mit dem der Besitzer fahren kann.
 *
 * @version 1.0 vom 25.06.2014
 * @author Thomas Kempe
 */

public class Roller

    // Anfang Attribute
    private double tankgroesse;
    private double tankinhalt;
    private int kilometerstand;
    // Ende Attribute

    /**
     * Der Konstruktor erzeugt ein Objekt der Klasse, wobei alle Attribute per
     * Parameterübergabe initialisiert werden.
     * @param pTankgroesse Gibt den Startwert des Attributs tankgroesse an.
     * @param pKmStand Gibt den Startwert des Attributs kilometerstand an.
     * @param pTankinhalt Gibt den Startwert des Attributs tankinhalt an.
     */
    public Roller(double pTankgroesse, int pKmStand, double pTankinhalt) {
        tankgroesse = pTankgroesse;
        kilometerstand = pKmStand;
        tankinhalt = pTankinhalt;
    }

    // Anfang Methoden
    /**
     * Die sondierende Methode für das Attribut kilometerstand.
     * @return Das Attribut kilometerstand wird zurückgegeben.
     */
    public int getKilometerstand() {
        return kilometerstand;
    }

    /**
     * Das Attribut tankinhalt wird erhöht, sofern die Menge in den Tank passt.
     * @param pMenge Um diesen Wert wird der Tankinhalt erhöht.
     */
    public void tanke(double pMenge) {
        if(passtInTank(pMenge))
        {
            tankinhalt = tankinhalt + pMenge;
        }
    }
}

```

```
/**  
 * Die sondierende Methode für das Attribut tankinhalt.  
 * @return Das Attribut tankinhalt wird zurückgegeben.  
 */  
public double getTankinhalt() {  
    return tankinhalt;  
}  
  
/**  
 * Der Roller fährt eine bestimmte Strecke, sofern genug Tankinhalt vorhanden  
 * ist. Der Roller verbraucht 0.03 Liter pro Kilometer.  
 * @param pStrecke Die zu fahrende Strecke.  
 */  
public void fahre(int pStrecke) {  
    double spritverbrauch = 0.03 * pStrecke;  
    if (spritverbrauch <= tankinhalt)  
    {  
        kilometerstand = kilometerstand + pStrecke;  
        tankinhalt = tankinhalt - spritverbrauch;  
    }  
}  
  
/**  
 * Die sondierende Methode für das Attribut tankgroesse.  
 * @return Das Attribut tankgroesse wird zurückgegeben.  
 */  
public double getTankgroesse()  
{  
    return tankgroesse;  
}  
  
/**  
 * Es wird geprüft, ob die gewünschte Menge Sprit noch in den Tank passt.  
 * @param pMenge Gibt die gewünschte zu tankende Menge an.  
 * @return Gibt an, ob die gewünschte Menge getankt werden kann.  
 */  
public boolean passtInTank(double pMenge) {  
    if(tankgroesse >= tankinhalt + pMenge)  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}  
  
// Ende Methoden  
} // end of Roller
```

Aufgaben

1. Wandeln Sie die folgenden Diagramme entweder „vorwärts“ in Implementations- oder „rückwärts“ in Entwurfsdiagramme um.

a)

Zapfsaeule
preisBenzin : Zahl
preisDiesel : Zahl
gewaehlterKraftstoff : Text
abgegebeneLiter : Zahl
gibKraftstoffAb

b)

Kamera
aufloesungHor : Zahl
aufloesungVer : Zahl
speicherGesamt : Zahl
speicherBelegt : Zahl
filmen

c)

Luftballon
groesse : Zahl
farbe : Text
aufgeblasen : Wahrheitswert
luftAufnehmen
platzen

d)

Staubsauger
– modell : String
– leistung : int
– energieklaesse : char
+ Staubsauger (pM : String, pL : int, pE : char)
+ getModell() : String
+ getLeistung() : int
+ getEKlaesse() : char
+ saugen(pMinuten : int) : void
+ beutelWechseln() : void

e)

Feuermelder
– lautstaerke : double
– alarm : boolean
+ Feuermelder (pL : double, pB : Batterie, pA : boolean)
+ getLautstaerke() : double
+ getBatterie() : Batterie
+ getAlarm() : boolean
+ feuerMelden() : void
+ alarmAbschalten() : void

– batt

1

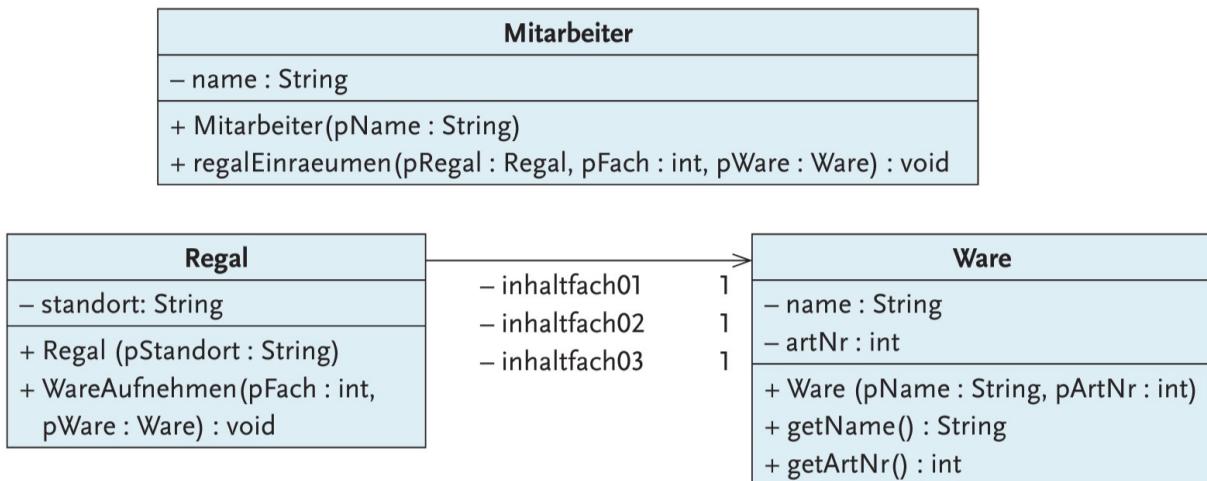
Batterie
– ladestand : int
+ Batterie ()
+ getLadestand() : int

2. Zur Ausrüstung eines Fußballschiedsrichters gehören eine Münze, eine gelbe Karte, eine rote Karte und eine Pfeife. Der Schiedsrichter kann Spielern eine Karte zeigen, mit der Münze eine Entscheidung treffen und mit der Pfeife pfeifen.

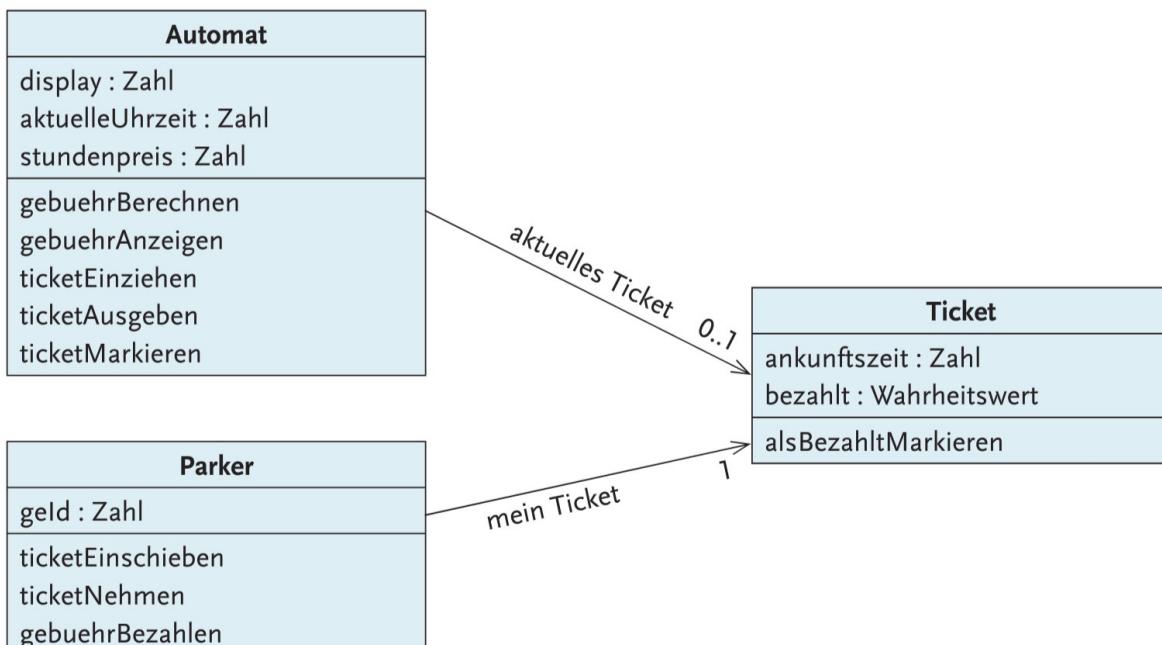
- a) Entwickeln Sie ein Entwurfsdiagramm, welches die obige Anforderung erfüllt (inklusive Methoden). Begründen Sie dabei die Entscheidungen der Assoziationsbeziehungen zwischen den Klassen.
- b) Zeichnen Sie ein passendes Objektdiagramm, in welchem vor allem auch die Beziehungen korrekt dargestellt werden.
- c) Entwickeln Sie aus dem Entwurfs- ein Implementationsdiagramm. Erstellen Sie zuvor eine Checkliste, die die Änderungen vom Entwurfs- zum Implementationsdiagramm enthält.
- d) Erstellen Sie eine Klassendokumentation für die Klasse *Schiedsrichter*.



- 3.** Folgende Klassen sind gegeben. Der Mitarbeiter im Supermarkt kann in eines der Fächer genau eine Ware in der Anzahl 1 einräumen.



- a)** Simulieren Sie das Einräumen mithilfe eines Zettels (das Regal) und Inhalten Ihres Etuis (Waren). Zur Simulation benutzen Sie die im Klassendiagramm vorgegebenen Methoden.
- b)** Erstellen Sie ein Sequenzdiagramm, das das Einräumen einer Ware in ein Regalfach durch einen Mitarbeiter darstellt.
- 4.** In einem Parkhaus holt sich der Parkende bei der Einfahrt ein Parkticket, auf dem die Ankunftszeit vermerkt ist. Bevor der Parkende das Parkhaus mit seinem Auto verlassen kann, muss er das Ticket bezahlen. Dieses muss dann an einem Schrankenautomaten eingegeben werden.
- Folgendes Entwurfsdiagramm eines Ticketautomaten ist gegeben.



- a)** Beschreiben Sie anhand des Entwurfsdiagramms den Bezahlvorgang. Bedenken Sie, dass immer nur Objekte der vorgegebenen Klassen während dieses Prozesses aktiv handeln, also ihre Methoden benutzen können.

- b) Erstellen Sie auf Grundlage von a) und unter Benutzung der Vorlage der Dokumentation der Klasse *Automat* eine Dokumentation aller beteiligten Klassen. Es sollen als Rückgaben und Parameter schon die Javatypen benutzt werden. Sollten Sie eine Entwicklungsumgebung wie den Java-Editor benutzen, dann können Sie die Dokumentation auch dort schon vor der eigentlichen Implementation im Quelltext notieren und über Javadoc ausgeben lassen.

Klasse Automat

Die Klasse *Automat* erhält von dem Parkenden ein Ticket, berechnet die Parkgebühr und gibt das als bezahlt markierte Ticket nach der Bezahlung der Gebühr wieder aus.

Konstruktor Automat (

Auftrag void gebuehrBerechnen ()
Die Parkdauer wird berechnet und mit dem Stundenpreis multipliziert.

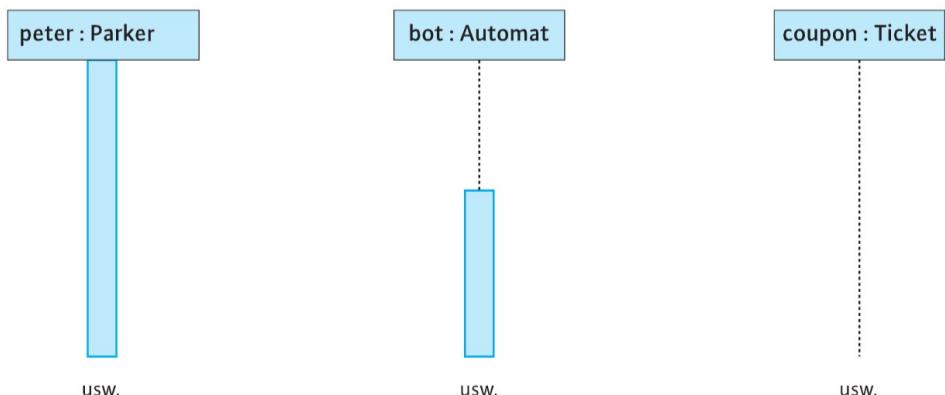
Auftrag void gebuehrAnzeigen ()
Die Gebühr wird in display gespeichert.

Auftrag void ticketEinziehen ()

... ...

- c) Erstellen Sie aus dem gegebenen Entwurfsdiagramm und auf Grundlage der Dokumentationen ein passendes Implementationsdiagramm.

- d) Erstellen Sie für den Bezahlvorgang ein Sequenzdiagramm. Angestoßen wird dieser Vorgang durch den Aufruf der Methode *ticketEinschieben* der Klasse *Parker*. Am Ende soll der Parker die Gebühr bezahlt haben und das Ticket als bezahlt markiert worden sein.



- e) Analysieren Sie gruppenteilig die Klassen *Besitzer* und *Roller* auf Seite 110ff. Erläutern Sie dabei, auf welchen Grundlagen aus diesem Kapitel die Programmierung beruht und welche Vorarbeit jeweils geleistet werden muss.
- f) Implementieren Sie anhand der bisherigen Ergebnisse das vorgestellte und entwickelte System „Parkticketautomat“. Nutzen Sie die in e) gewonnenen Ergebnisse.

5.5 Vererbung

Ein Grundkonzept der objektorientierten Programmierung ist, dass Doppelungen sowohl in der Modellierung als dann auch später in der Implementation weitestgehend vermieden werden sollen. Eine Idee, die diesem Konzept Rechnung trägt, ist die Vererbungsbeziehung. Diese Beziehung ermöglicht es, dass Attribute einer Klasse – der Oberklasse – an andere Klassen – die Unterklassen – vererbt werden. Dies soll an einem Beispiel verdeutlicht werden.

Anforderung

Der Rollerbesitzer Paul wird ein richtiger Motorfan und kauft sich in den kommenden Jahren noch ein Auto und ein Crossmotorrad. Grundsätzlich kann er beide Fahrzeuge auch fahren und sie betanken. Mit dem Auto kann er jedoch noch auf der Autobahn fahren und mit dem Crossmotorrad fährt er Rennen. Da der Roller ein historisches Modell ist, ist die Information über das Baujahr besonders wichtig. Der Einfachheit halber verbrauchen alle Fahrzeuge beim Fahren die gleiche Menge Benzin.

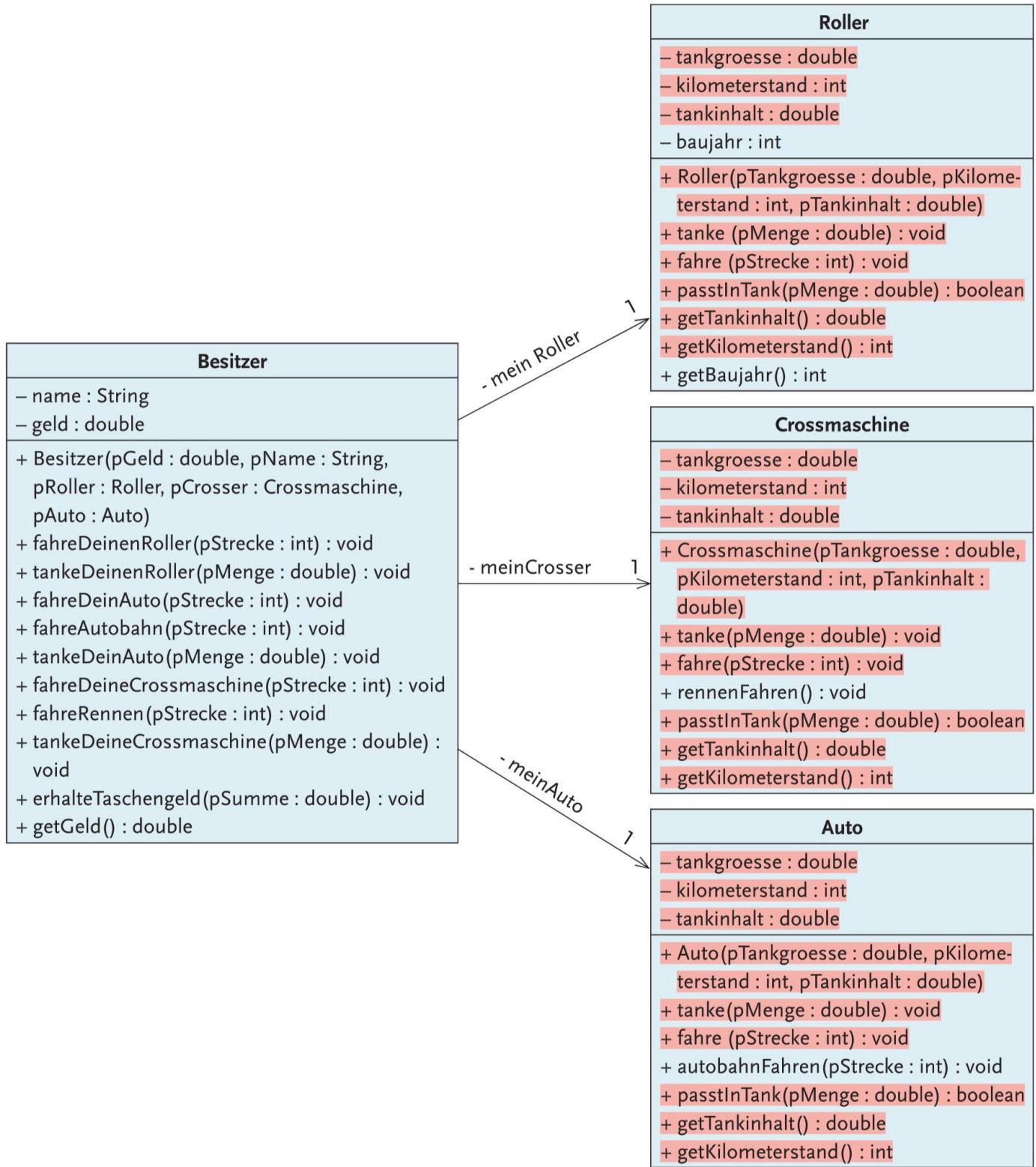


Ein Klassendiagramm müsste also die Klassen *Besitzer*, *Roller*, *Auto* und *Crossmaschine* beinhalten. Das Problem ist jedoch, dass schon wesentliche Teile im Diagramm mehrfach vorkommen und später dann auch mehrfach implementiert werden müssen.

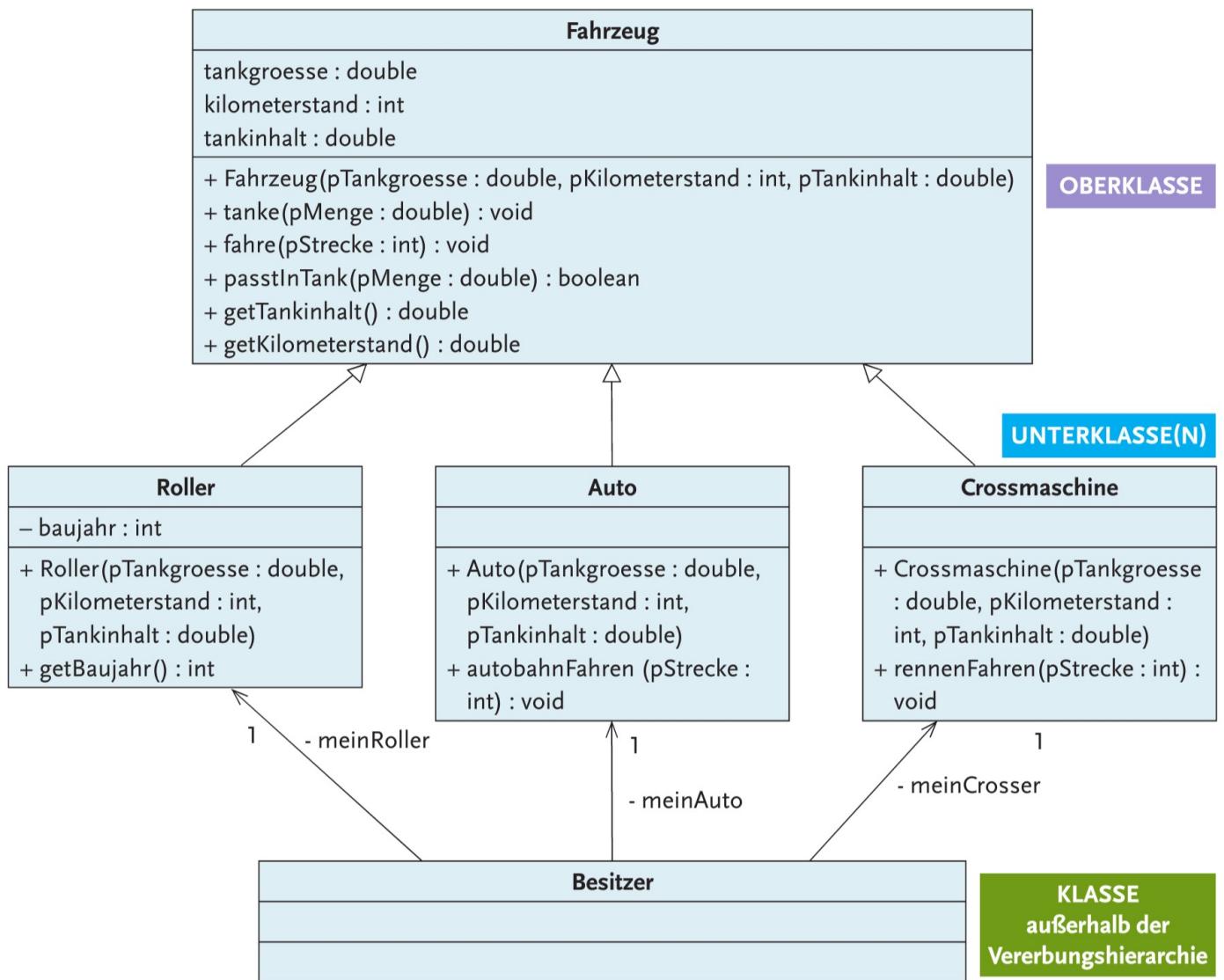


Mithilfe der **Vererbung** (oder Ist-Beziehung) erben die Unterklassen (oder Subklassen) Attribute und Methoden der Oberklasse (Superklasse). Die Unterklassen spezialisieren dabei die Oberklasse (Spezialisierung), die Oberklasse generalisiert ihre Unterklassen (Generalisierung).

Vererbungshierarchien können sich über mehrere Ebenen erstrecken. Die Vererbung wird durch einen geschlossenen, nicht gefüllten Pfeil gekennzeichnet. Dieser zeigt von Richtung Unterklasse zur Oberklasse.



Bei genauer Betrachtung lässt sich erkennen, dass die Klassen *Roller*, *Auto* und *Crossmaschine* im Prinzip nahezu identisch sind. Diese Eigenschaft nutzt man aus, um sie in einer Oberklasse zu generalisieren. Somit erhalten die drei Klassen eine Oberklasse *Fahrzeug*, die die gemeinsamen Attribute und Methoden enthält und diese an die Unterklassen weitervererbt. Lediglich die spezialisierten Attribute und Methoden der Unterklassen – dazu gehört auch der Konstruktor – werden zusätzlich im Diagramm notiert und später auch implementiert.



Neben dem offensichtlichen Vorteil der Übersichtlichkeit und Vermeidung von Duplikierung, erleichtert eine Vererbungshierarchie auch die Erweiterbarkeit eines Systems. Neue Fahrzeuge lassen sich, sollten sie ähnlich zu den bisherigen sein, leicht als Spezialisierung der Oberklasse *Fahrzeug* ableiten.

Die Implementation der Vererbung ist einfach. In den Unterklassen signalisiert das Wort *extends* zu Beginn der Klasse, von welcher Oberklasse Attribute und Methoden geerbt werden. Diese werden in der Unterklasse dann nicht mehr programmiert; die speziellen Methoden, die in den Klassendiagrammen der Unterklasse auftauchen, müssen natürlich programmiert werden. Zudem muss ein Konstruktor der jeweiligen Unterklasse inklusive Parameterübergaben für die Initialisierung der Attribute programmiert werden. Mithilfe des Befehls *super(Parameter)* wird der Konstruktor der Oberklasse aufgerufen.

H # bedeutet „protected“. Ein Zugriff auf die Attribute ist auch von Unterklassen aus möglich, was bei „private“ nicht möglich wäre.

→ Aufgabe 1, 2

```
/**
 *
 * Die Klasse Fahrzeug ist die Vorlage für Fahrzeuge, mit denen der Besitzer
 * fahren kann.
 *
 * @version 1.0 vom 25.06.2014
 * @author Thomas Kempe
 */

public class Fahrzeug {

    // Anfang Attribute
    protected double tankgroesse;
    protected double tankinhalt;
    protected int kilometerstand;
    // Ende Attribute

    /**
     * Der Konstruktor erzeugt ein Objekt der Klasse, wobei alle Attribute per
     * Parameterübergabe initialisiert werden.
     * @param pTankgroesse Gibt den Startwert des Attributs tankgroesse an.
     * @param pKmStand Gibt den Startwert des Attributs kilometerstand an.
     * @param pTankinhalt Gibt den Startwert des Attributs tankinhalt an.
     */
    public Fahrzeug(double pTankgroesse, int pKmStand, int pTankinhalt) {
        tankgroesse = pTankgroesse;
        kilometerstand = pKmStand;
        tankinhalt = pTankinhalt;
    }

    // Anfang Methoden gleich wie bei der Klasse Roller auf Seite 112
    ...
    // Ende Methoden
} // end of Fahrzeug
```

```
/**
 *
 * Die Klasse Roller ist ein Fahrzeug, mit dem der Besitzer fahren kann.
 *
 * @version 1.0 vom 25.06.2014
 * @author Thomas Kempe
 */

public class Roller extends Fahrzeug {

    // Anfang Attribute werden alle bis auf baujahr geerbt.
    private int baujahr;
    // Ende Attribute
```

```

/**
 * Der Konstruktor erzeugt ein Objekt der Klasse, wobei alle Attribute per
 * Parameterübergabe initialisiert werden.
 * @param pTankgroesse Gibt den Startwert des Attributs tankgroesse an.
 * @param pKmStand Gibt den Startwert des Attributs kilometerstand an.
 * @param pTankinhalt Gibt den Startwert des Attributs tankinhalt an.
 */
public Roller(double pTankgroesse, int pKmStand, double pTankinhalt, int
pBaujahr) {
    super(pTankgroesse, pKmStand, pTankinhalt);
    this.baujahr = pBaujahr;
}
// Anfang Methoden werden alle bis auf getBaujahr() : int geerbt.
...
// Ende Methoden
} // end of Roller

```

```

/**
 *
 * Die Klasse Auto ist ein Fahrzeug, mit dem der Besitzer fahren kann.
 *
 * @version 1.0 von 25.06.2014
 * @author Thomas Kempe
 */
public class Auto extends Fahrzeug {

    // Anfang Attribute werden alle geerbt.
    // Ende Attribute

    /**
     * Der Konstruktor erzeugt ein Objekt der Klasse, wobei alle Attribute per
     * Parameterübergabe initialisiert werden.
     * @param pTankgroesse Gibt den Startwert des Attributs tankgroesse an.
     * @param pKmStand Gibt den Startwert des Attributs kilometerstand an.
     * @param pTankinhalt Gibt den Startwert des Attributs tankinhalt an.
     */
    public Auto (double pTankgroesse, int pKmStand, double pTankinhalt) {
        super (pTankgroesse, pKmStand, pTankinhalt);
    }

    // Anfang Methoden werden alle bis auf autobahnFahren geerbt.
    /**
     * Das Auto fährt eine bestimmte Strecke auf der Autobahn, sofern genug
     * Tankinhalt vorhanden ist.
     * Das Auto verbraucht 0.11 Liter pro Kilometer.
     * @param pStrecke Die zu fahrende Strecke.
     */

```

```

public void autobahnFahren(int pStrecke) {
    double spritverbrauch = 0.11 * pStrecke;
    if(spritverbrauch < tankinhalt)
    {
        kilometerstand = kilometerstand + pStrecke;
        tankinhalt = tankinhalt - spritVerbrauch;
    }
}

// Ende Methoden
} // end of Auto

```

Aufgaben

- 1.**
 - a)** Erläutern Sie das Konzept der Vererbung anhand der Begriffe „Spezialisierung“, „Generalisierung“ und „Erben von Attributen und Methoden“.
 - b)** Gegeben sind die Klassen *Personenkraftfahrzeug* (PKW), *Lastkraftfahrzeug* (LKW), *Landfahrzeug*, *Kraftfahrzeug* und *Motorrad*.
 - i) Entwickeln Sie ein geeignetes Klassendiagramm, das diese Struktur unter Zuhilfenahme der Vererbung abbildet.
 - ii) Erweitern Sie die Vererbungshierarchie um Wasser- und Luftfahrzeuge mit diversen Unterklassen.
 - iii) Erläutern Sie anhand des Beispiels aus Teilaufgabe i) und ii) die Spezialisierung und deren Vorteile.
 - iv) Implementieren Sie eine Oberklasse und zwei ihrer Unterklassen. Es reichen jeweils die Methodenhüllen. Ersichtlich sollte aber auf jeden Fall werden, welche Bestandteile zur Programmierung einer Vererbung notwendig sind.
 - c)** Entwickeln Sie ein Klassendiagramm mit einer Vererbungshierarchie mit vier Ebenen, also: Es existiert eine Oberklasse, diese hat mehrere Unterklassen, diese hat wiederum mehrere Unterklassen, die nächste ebenso. Es bieten sich Vererbungshierarchien aus der Natur oder dem Sport an.
- 2.** Vervollständigen Sie die Fahrzeug-Klassen und erweitern bzw. verändern Sie die Klasse *Besitzer* anhand des Klassendiagramms auf S. 118.

5.6 Prüfungsvorbereitung

Aufgabe 1 Objekte und Klassen

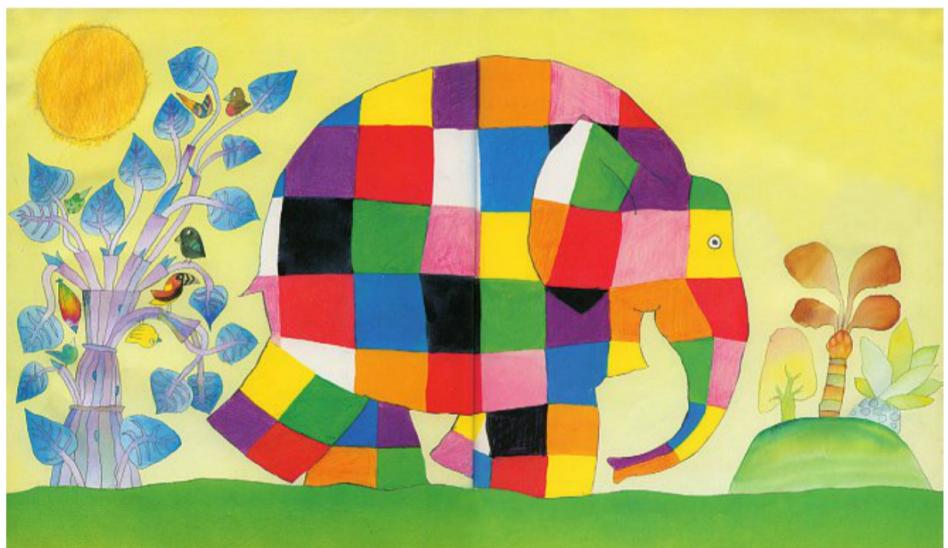
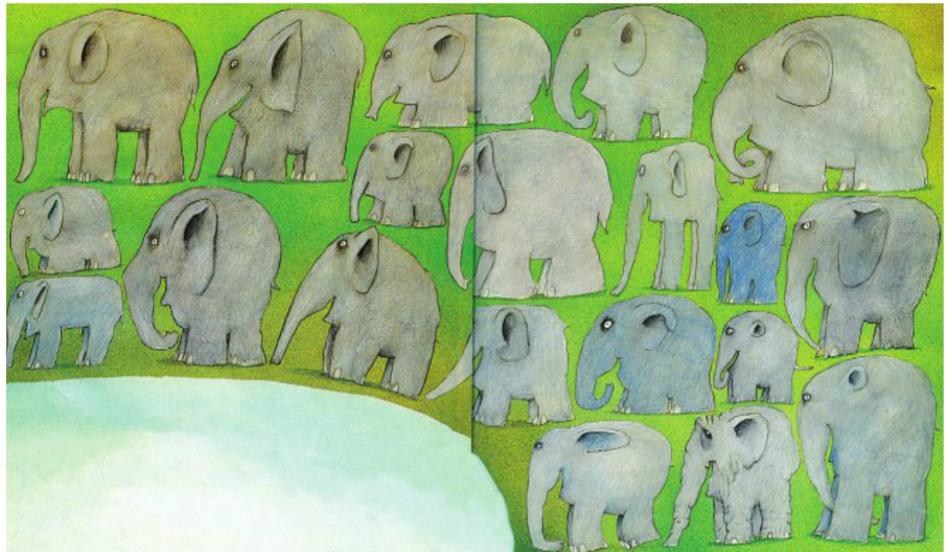
In dem Kinderbuch „Elmar“ von David McKee wird eine Herde von Elefanten vorgestellt.

- a) Erstellen Sie abgeleitet aus den einzelnen Elefant-Objekten eine Klasse *Elefant* mit möglichst vielen, in dem Bild auftretenden Attributwerten als Entwurfsdiagramm.

Erläutern Sie Ihre Entscheidungen für eine Attributwahl jeweils anhand einzelner Elefant-Objekte.

- b) Bestimmen Sie, welche Attribute Ihrer Klasse *Elefant* bei der Erzeugung aller Elefanten gleich sind und welche unterschiedlich. Notieren Sie dafür einige Objektdiagramme zu Elefanten aus dem Bild.

- c) Der Elefant Elmar stellt in dem Buch einen ganz besonderen Elefanten dar. Erläutern Sie, warum er dies in dem Buch ist und warum er es in Ihrem Modell aus a) höchstwahrscheinlich nicht ist.



Aufgabe 2 Objekte

Die höchste Auszeichnung des deutschen Staates für einen Bürger ist das Bundesverdienstkreuz. Der Ministerpräsident eines Bundeslandes kann einen Bürger für die Ehrung vorschlagen, woraufhin der Bundespräsident dann das Bundesverdienstkreuz (BVK) verleihen kann. Das Bundesverdienstkreuz gibt es in einer Herren- und in einer Damenversion.



a) Stellen Sie die Wertbelegung der Attribute der Objekte dar, wenn

- i) Herr Konrad nichts mit dem BVK-Verfahren zu tun hat,
- ii) Herr Konrad für das BVK vorgeschlagen wurde,
- iii) Herr Konrad als BVK-Träger ernannt wurde.

i)

<u>konrad : Buerger</u>
name :
vorgeschlagen :
verliehen :
meinBVK :

<u>bvk15681 : BVK</u>
version :

b) Erläutern Sie die Wahl der sich aus a) ergebenden Datentypen für die Attribute der beiden Objekte.

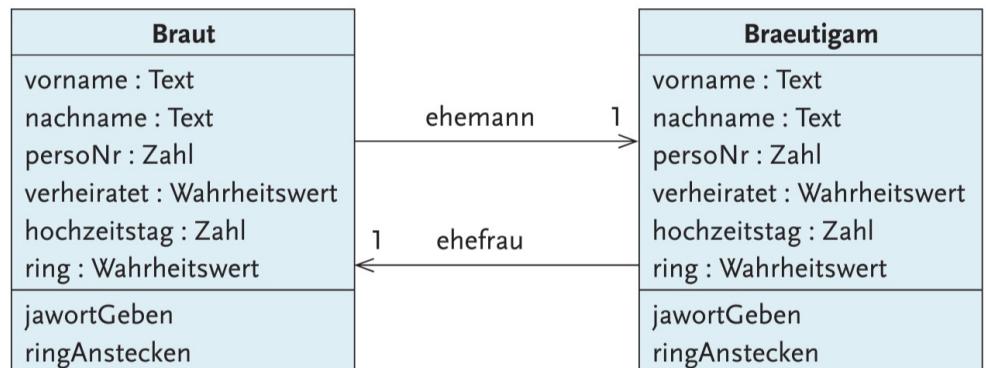
c) Beschreiben Sie den Zusammenhang und den Unterschied zwischen Objekt und Klasse.

Aufgabe 3 Klassendiagramme

Folgendes Entwurfsdiagramm ist gegeben:

Dokumentation zur Klasse
Braut/Bräutigam s. S. 224

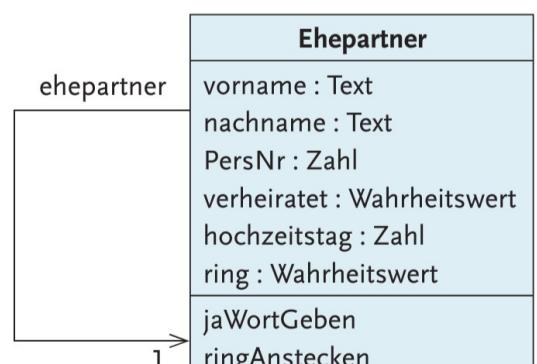
H



a) Beschreiben Sie den Aufbau des Entwurfsdiagramms mithilfe der Fachbegriffe.

b) Analysieren Sie, ob und inwiefern das Klassendiagramm *Ehepartner* dasselbe (ein Ehepaar) modelliert wie das oben gegebene Klassendiagramm mit den Klassen *Braut* und *Bräutigam*. Gehen Sie dabei auch auf Vor- und Nachteile der beiden Modellierungen ein.

c) Benennen Sie die Schritte, die bei der Überführung eines Entwurfsdiagramms in ein Implementationsdiagramm beachtet werden müssen.



d) Überführen Sie das Klassendiagramm der Klasse *Braut* in ein Implementationsdiagramm. Treffen Sie dabei aufgrund der gegebenen Kurzdokumentation im Anhang sinnvolle Entscheidungen hinsichtlich Rückgabetypen, Parametern und get- und set-Methoden. Begründen Sie Ihre Entscheidungen jeweils kurz.

e) Implementieren Sie

- den Konstruktor der Klasse *Braut*,
- jeweils eine get- und eine set-Methode der Klasse *Braut*.

f) Entwickeln Sie anhand der gegebenen Anforderung ein Implementationsdiagramm.



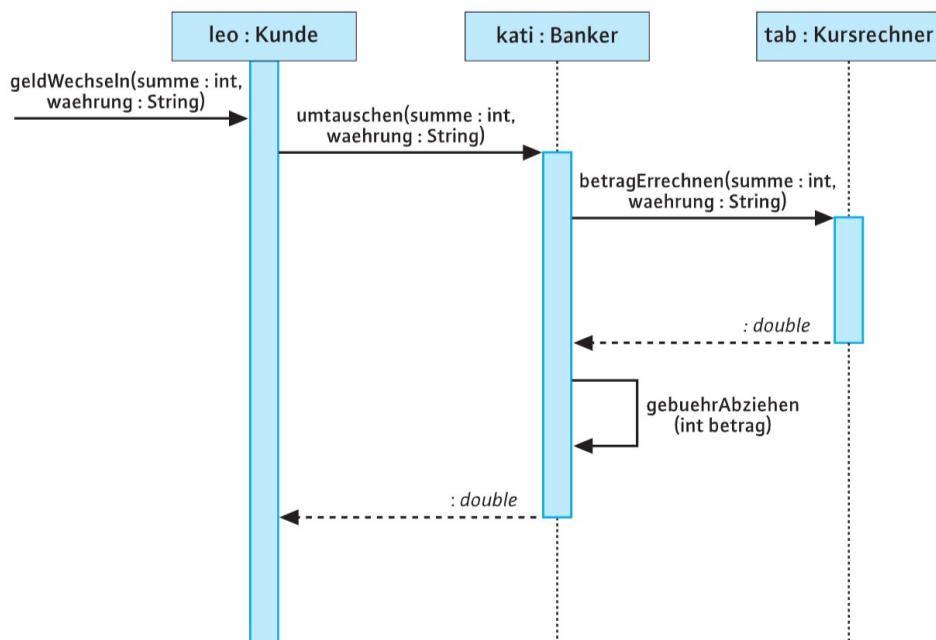
Anforderung

Eine Ehe kann in Deutschland von einem Standesbeamten, der einen Vor- und Nachnamen besitzt, geschlossen werden. Jeder Standesbeamte hat ein eigenes Dienstsiegel, mit dem er Dokumente besiegen kann. Für die Eheschließung muss er zuerst die korrekte Identität der Ehewilligen feststellen (*persoNr* im Klassendiagramm). Er schließt die Ehe dann, indem er sich das Ja-Wort der beiden Ehewilligen geben lässt. Je nach Wunsch kann er dann noch den Nachnamen des einen Ehepartners auf den anderen übertragen.

g) Beschreiben Sie mithilfe der von Ihnen in d) und f) entwickelten Methoden, wie eine Namensänderung (siehe Anforderung) programmiert werden kann.

Aufgabe 4 Objektkommunikation

Folgendes Sequenzdiagramm ist gegeben:



Dokumentation zu den
Klassen *Kunde*, *Banker* und
Kursrechner s. S. 223

H

a) Analysieren Sie das Sequenzdiagramm unter Beachtung der Dokumentation im Anhang.

b) Implementieren Sie die Methode *double betragErrechnen(int summe, String waehrung)* der Klasse *Kursrechner* (siehe Dokumentation).

Es können in dieser Methode nur Euros in zwei mögliche Währungen getauscht werden:

- in „Dollar“ mit dem Wechselkurs 1 Euro = 1,38 Dollar und
- in „Pfund“ mit dem Wechselkurs 1 Euro = 0,83 Pfund.

Klassendokumentationen

Dokumentation: Klasse Braut/Braeutigam

Anfrage boolean jawortGeben()

Die Methode gibt über einen Wahrheitswert die Einwilligung oder nicht-Einwilligung in die Heirat an.

Auftrag void ringAnstecken()

Dem Ehepartner (Ehefrau/Ehemann) wird der Ring angesteckt.

Dokumentation: Klasse Kunde

Auftrag void geldwechsel(int summe, String waehrung)

Der Kunde möchte die angegebene Summe (in Euro) in die angegebene Währung eintauschen.

Dokumentation: Klasse Banker

Anfrage double umtauschen(int summe, String waehrung)

Der Banker zahlt dem Kunden den berechneten Betrag abzüglich einer Gebühr aus.

Auftrag void gebuehr(int summe, String waehrung)

Auftrag void gebuehrAbziehen(int Betrag)

Fünf Prozent des Betrags werden einbehalten.

Dokumentation: Klasse Kursrechner

Anfrage double betragErrechnen (int summe, String waehrung)

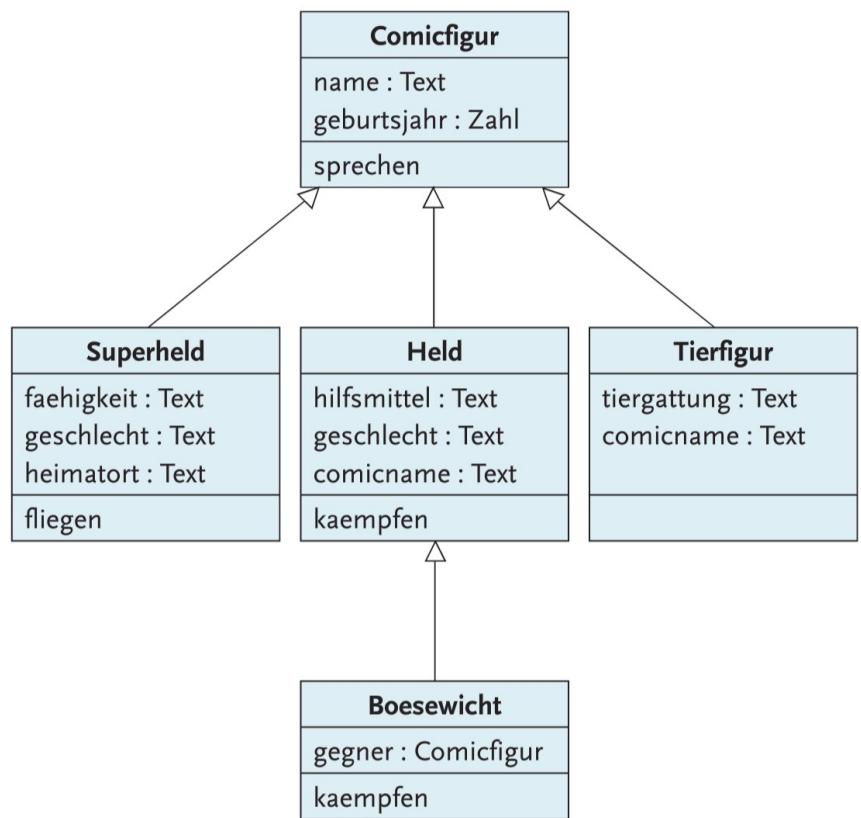
Die Summe wird anhand eines Tauschkurses in die gegebene Währung umgerechnet

Aufgabe 5 Vererbung

In dem abgebildeten Klassendiagramm mit Vererbungsstruktur ist folgende Anforderung umgesetzt worden:

Anforderung

Eine Comicfigur besitzt einen Namen und ihr wird als Geburtsjahr das Jahr ihres ersten Erscheinens zugerechnet. Jede Comicfigur ist entweder männlich oder weiblich und hat zudem einen Heimatort, in dem sie lebt. Jede Comicfigur kommt hauptsächlich in einem Comic mit einem Namen als Titel vor. Die Comicfiguren *Superhelden* haben eine übernatürliche Fähigkeit und einige von ihnen können fliegen, sie alle können aber kämpfen. Die Comicfiguren *Helden* kämpfen wie *Superhelden* gegen das Böse und benutzen ein besonderes Hilfsmittel. Die Comicfiguren *Tierfiguren* sind klar einer Tiergattung zuzuordnen. Die Comicfigur *Bösewicht* ist der Gegner eines *Superhelden* oder eines *Helden* und kann gegen diese kämpfen. *Bösewichte* können *Superbösewichte* (ähnlich zu *Superhelden*) oder normale *Bösewichte* (ähnlich zu *Helden*) sein.



- Beschreiben Sie die Grundstruktur einer Vererbungshierarchie und erläutern Sie an einem Beispiel Spezialisierung und Generalisierung.
- Analysieren Sie das gegebene Klassendiagramm hinsichtlich der formulierten Anforderung, indem Sie korrekte und nicht korrekte Umsetzungen identifizieren.
- Entwickeln Sie aufgrund der Anforderung ein eigenes Klassendiagramm.
- Entwickeln Sie für eine Oberklasse und eine zugehörige Unterklasse ein Implementationsdiagramm.
- Implementieren Sie die beiden Klassen aus d), wobei besondere Methoden nur als „Hülle“ mit Signatur vorkommen sollen.



9 783140 371261