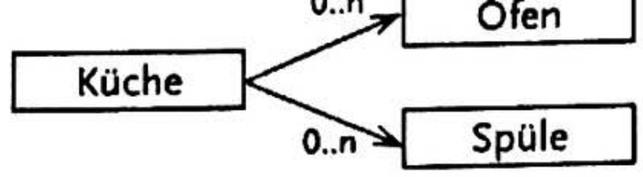
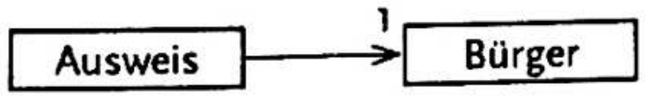
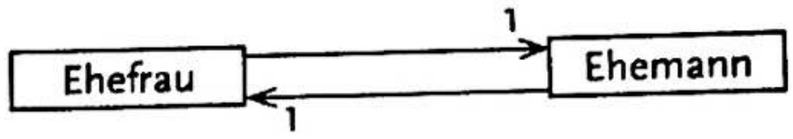
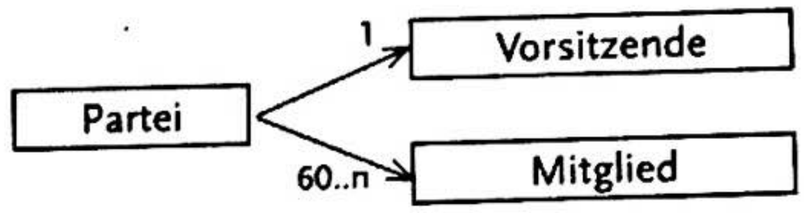
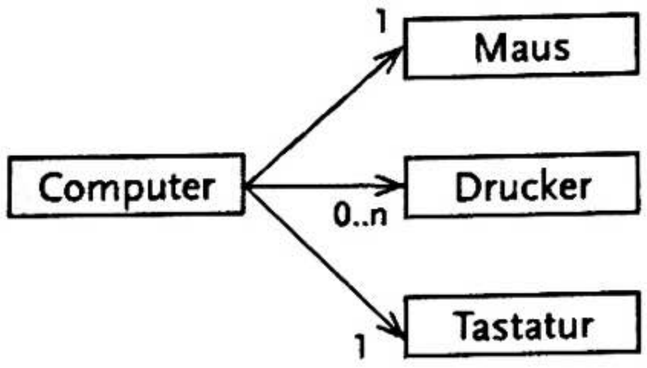










5.4 Klassen und Beziehungen implementieren

SB S. 114

1. a)

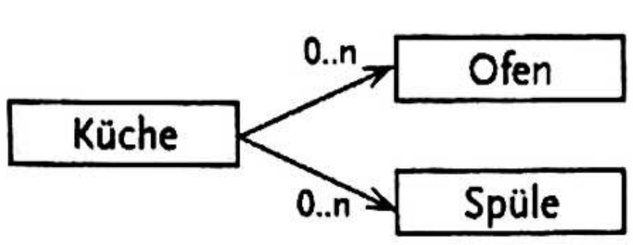
Zapfsaeule
– preisBenzin : double – preisDiesel : double – gewaehlterKraftstoff : String – abgegebene Liter : double
+ Zapfsaeule() + gibKraftstoffAb(pArt : String, pMenge : double) : void *... alle get- & set-Methoden*

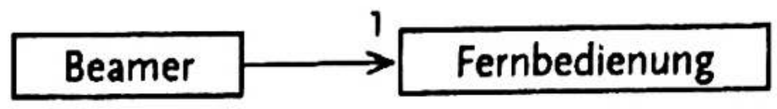
b)

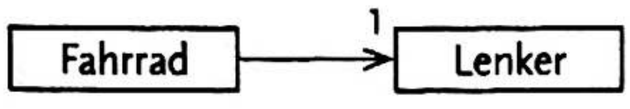
Kamera
– aufloesungHor : int – aufloesungVer : int – speicherGesamt : int – speicherBelegt : int
+ Kamera(pAH: int, pAV : int, pSG : int, pSB : int) + filmen() : void *... alle get- & set-Methoden*

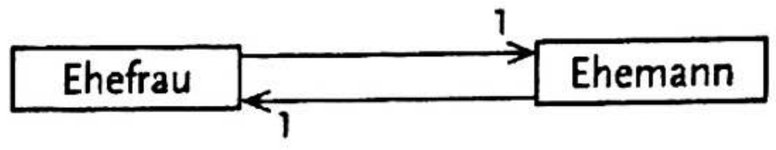
c)

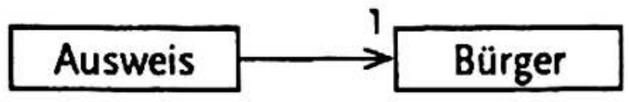
Luftballon
– groesse : int – farbe : String – aufgeblasen : boolean
+ Luftballon(pGroesse: int, pFarbe : String, pAufgeblasen : boolean) + luftAufnehmen() : void + platzen() : void *... alle get- & set-Methoden*

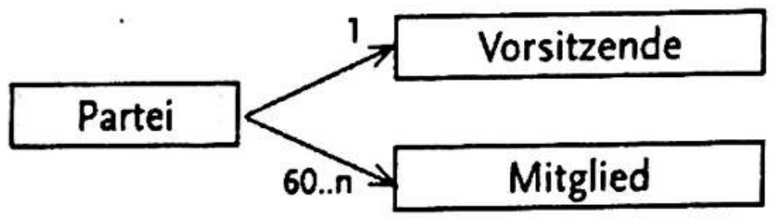


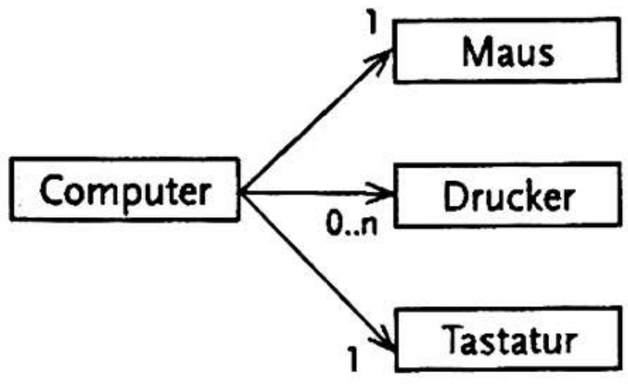












5.4 Klassen und Beziehungen implementieren

SB S. 114

1. a)

Zapfsaeule
– preisBenzin : double – preisDiesel : double – gewaehlterKraftstoff : String – abgegebene Liter : double
+ Zapfsaeule() + gibKraftstoffAb(pArt : String, pMenge : double) : void *... alle get- & set-Methoden*

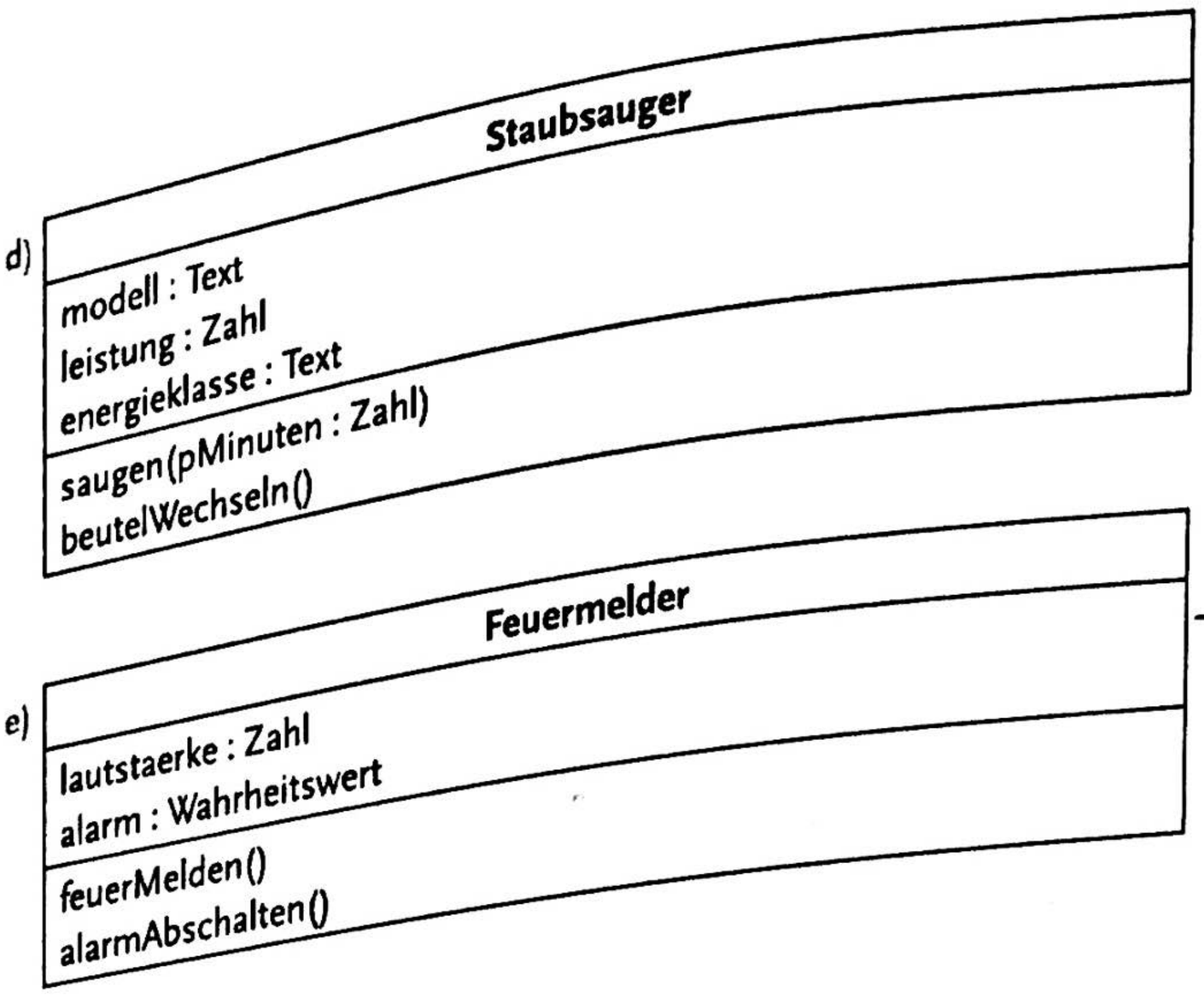
b)

Kamera
– aufloesungHor : int – aufloesungVer : int – speicherGesamt : int – speicherBelegt : int
+ Kamera(pAH: int, pAV : int, pSG : int, pSB : int) + filmen() : void *... alle get- & set-Methoden*

c)

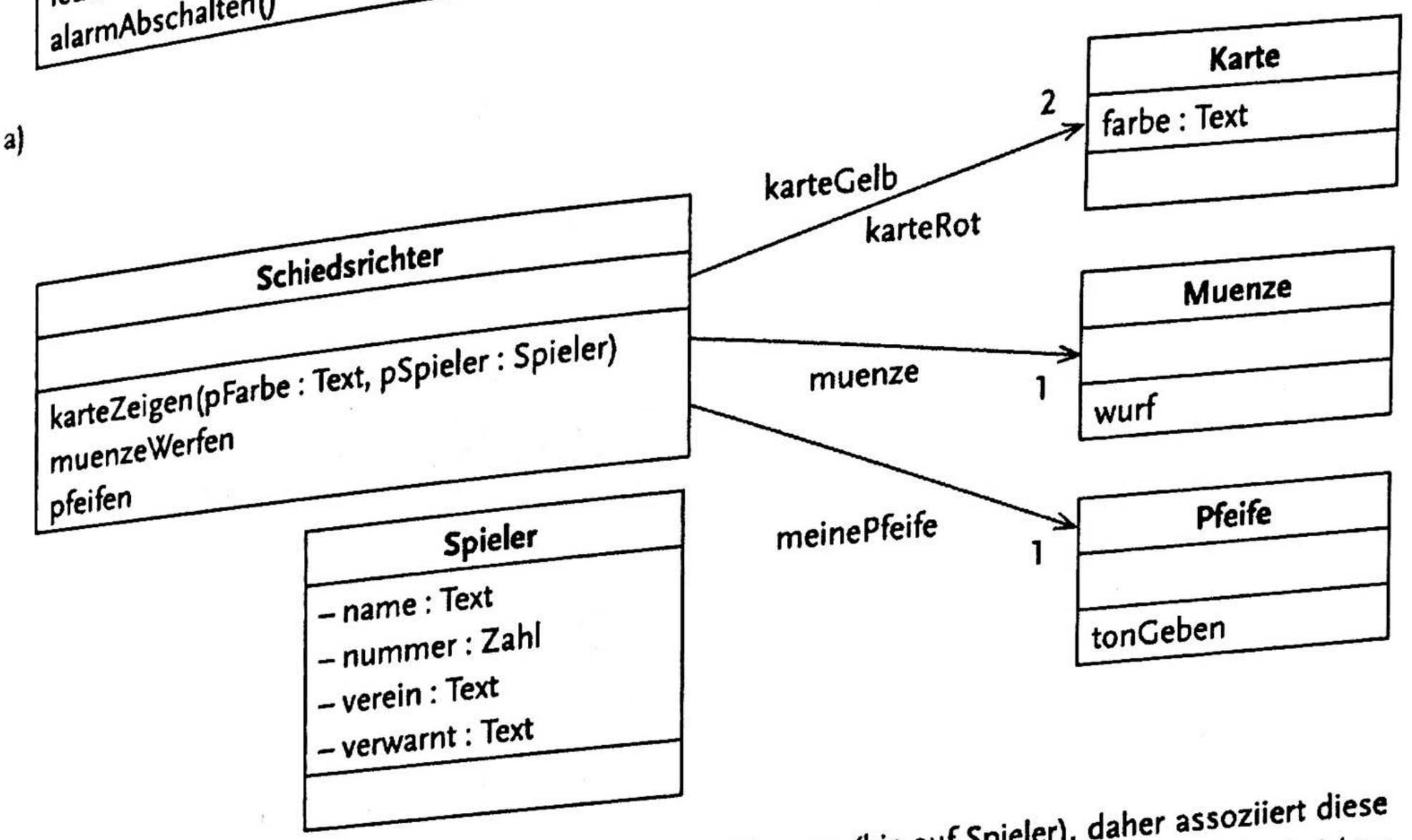
Luftballon
– groesse : int – farbe : String – aufgeblasen : boolean
+ Luftballon(pGroesse: int, pFarbe : String, pAufgeblasen : boolean) + luftAufnehmen() : void + platzen() : void *... alle get- & set-Methoden*

d)



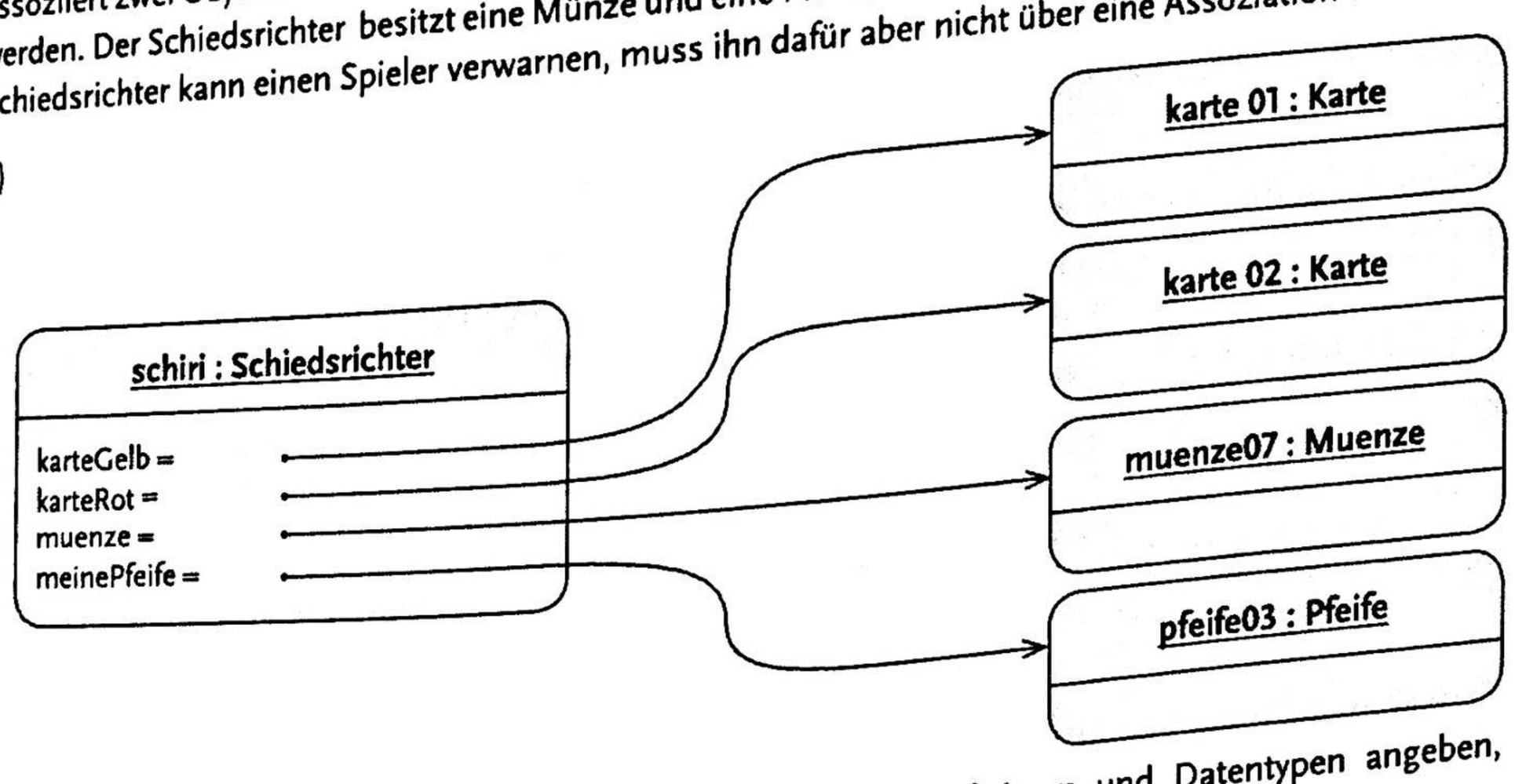
e)

2. a)



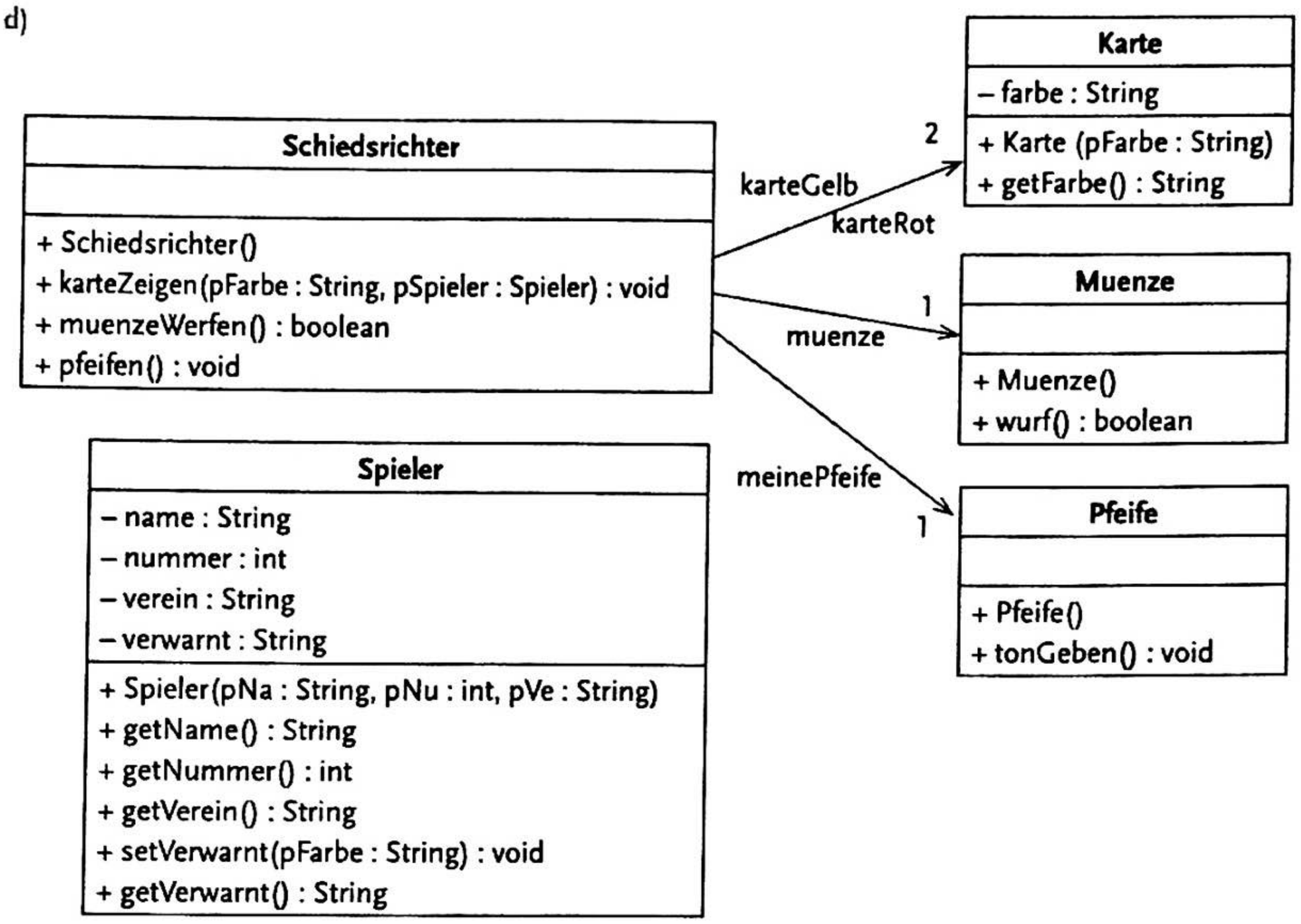
Ein Schiedsrichterobjekt verwaltet Objekte der anderen Klassen (bis auf Spieler), daher assoziiert diese Klasse die jeweils anderen. Zwischen den anderen besteht keine Assoziation. Die Klasse *Schiedsrichter* assoziiert zwei Objekte der Klasse *Karte*, eine rote und eine gelbe. Es könnten hier auch zwei Pfeile benutzt werden. Der Schiedsrichter besitzt eine Münze und eine Pfeife, die keine Attribute besitzen (müssen). Der Schiedsrichter kann einen Spieler verwarnen, muss ihn dafür aber nicht über eine Assoziation verwalten.

b)



c) Checkliste: Datentypen ändern, Parameter und Rückgaben definieren und Datentypen angeben, Konstruktor(en) definieren, get- und set-Methoden einfügen
Die Methoden *pfeifen/tonGeben* können evtl. Rückgaben besitzen.

d)



Klasse Schiedsrichter

Die Klasse *Schiedsrichter* kann über das Zeigen von Karten, Münzwürfe und Pfeifen Einfluss auf das Spiel nehmen.

Konstruktor **Schiedsrichter**
Ein Objekt mit Festlegung der Attribute wird instanziiert.

Auftrag **void karteZeigen (String pFarbe, Spieler pSpieler)**
Ein Objekt der Klasse *Spieler* wird mit einer Karte verwarnt, sodass dessen Attribut *verwarnt* auf *pFarbe* gesetzt wird.

Return: –

Parameter: Als Parameter werden der Spieler und die Farbenkarte übergeben.

Auftrag **boolean muenzeWerfen ()**
Für zu treffende Entscheidungen gibt der Wurf mit 50%iger Wahrscheinlichkeit *true* oder *false* zurück.

Return: Der Wurf liefert *true* oder *false* zurück.

Parameter: –

Auftrag **void pfeifen ()**
Der Pfiff hat in dieser Simulation noch keine Auswirkungen.

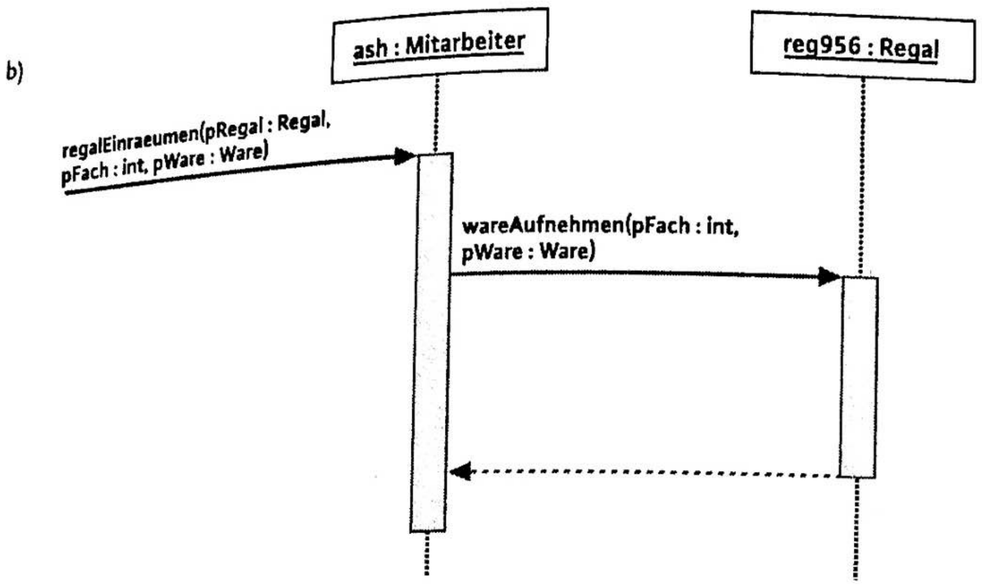
Return: –

Parameter: –

B S. 115

3. a) Entscheidend bei der Simulation ist, dass das Einräumen des Regals vom Mitarbeiter „angestoßen" wird. Im Prinzip wird die Ware selbst nicht durch Methodenaufrufe aktiv, es wird lediglich eine Referenz zuerst vom Mitarbeiter und dann vom Regal auf die Ware gesetzt.

b)



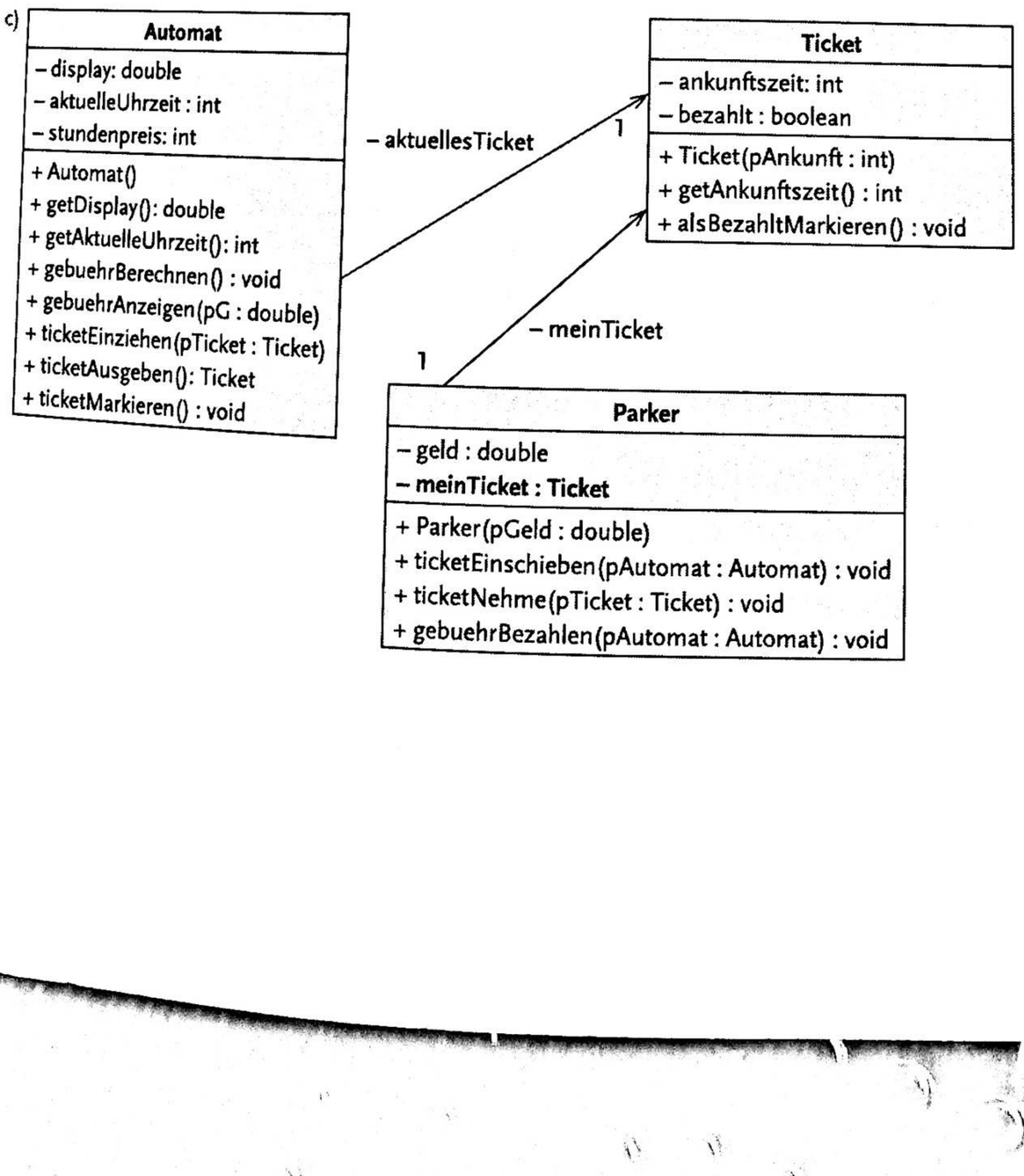
4. Diese Aufgabe kann auch als kleinere Projektaufgabe gestellt werden.

a) Da die Methodennamen Interpretationen zulassen, können auch andere Lösungen sinnvoll sein.
- Ein Parker schiebt das Ticket (meinTicket) in den Ticketautomaten ein. → *ticketEinschieben* der Klasse *Parker*
- Der Automat zieht das Ticket als *aktuellesTicket* ein → *ticketEinziehen* der Klasse *Automat*, beim Parker muss *meinTicket* auf null gesetzt werden
- Der Automat holt sich von *aktuellesTicket* die Ankunftszeit und berechnet mit Hilfe von *aktuelleUhrzeit* die Parkdauer und daraus die Gebühr (diese kann in *display* als Attribut gespeichert werden)
- Der Parker bezahlt die Gebühr, wodurch beim Automaten *aktuellesTicket* als bezahlt markiert wird und *aktuellesTicket* ausgegeben wird.
- Der Parker übernimmt das Ticket wieder als Attribut durch *ticketNehmen*, *aktuellesTicket* wird beim Automaten auf null gesetzt.

SB S. 116

b) Die Klassendokumentation findet sich in der Implementation und kann über Javadoc angezeigt werden (siehe f).

c)



d)

e)

e) Grundlagen si
tationsdiagrar
und Paramete
rallel muss ei
kommunikati

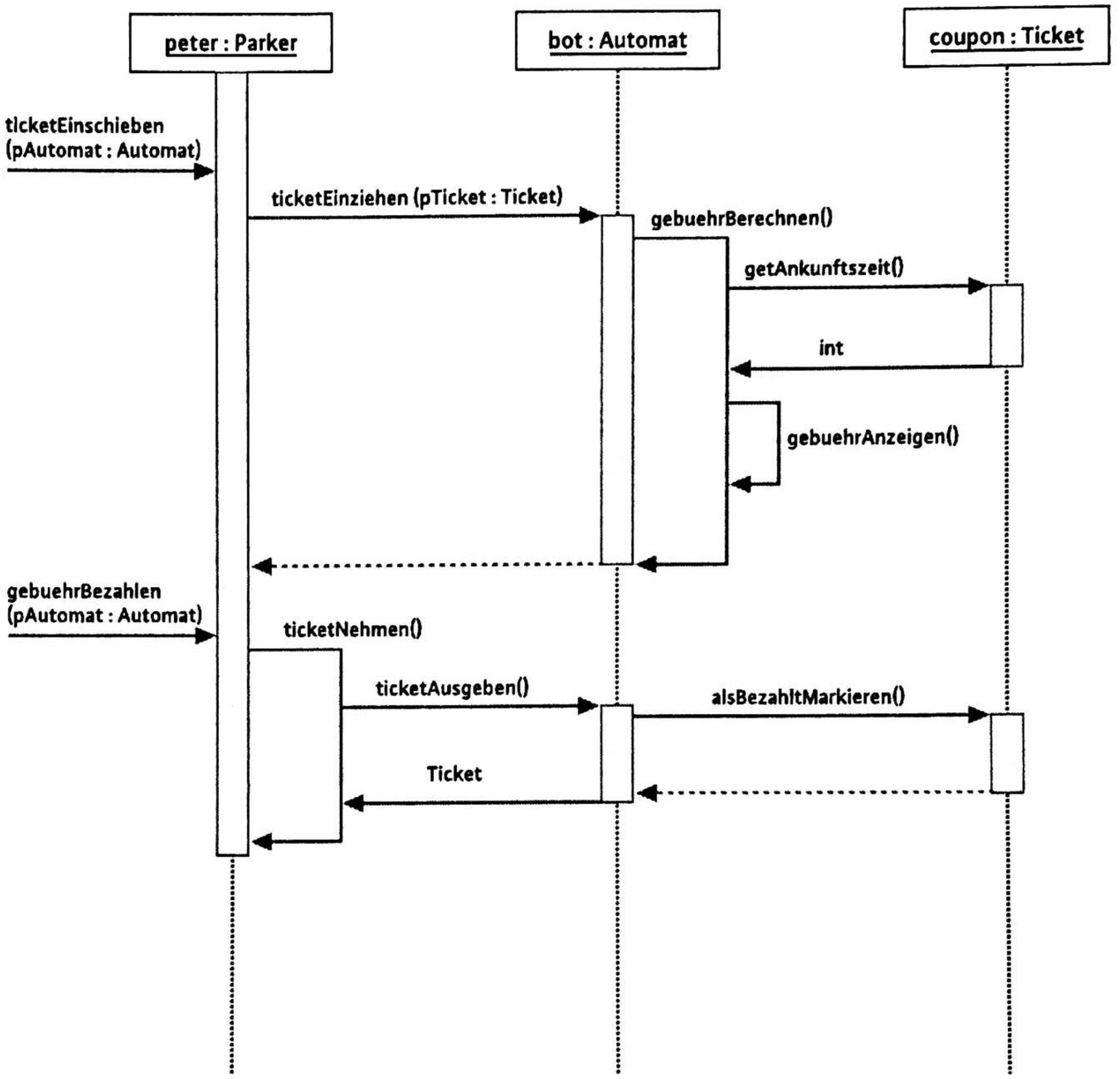
f) Siehe die Pro

5.5 Vererb

SB S. 122

1. a) Bei der Verer
eine Oberkla
Betrachtet m
dass die Unt
Unterklassen
generalisiert.
hinweg) die
implementie

d)



e) Grundlagen sind der Klassenentwurf im Entwurfsdiagramm und dessen Überführung in ein Implementationsdiagramm. Dabei müssen Datentypen programmiersprachenspezifisch geändert, Rückgaben und Parameter eingeführt, get- und set-Methoden gesetzt und der Konstruktor entwickelt werden. Parallel muss eine Klassendokumentation erstellt werden. Hilfreich für die Implementation von Objektkommunikation sind Sequenzdiagramme.

f) Siehe die Programmierung im Anhang.

5.5 Vererbung

SB S. 122

1. a) Bei der Vererbung werden mindestens zwei Klassen in eine hierarchische Beziehung gesetzt. Es gibt eine Oberklasse und eine oder mehrere Unterklassen, die wiederum Unterklassen besitzen können. Betrachtet man die Vererbungsstruktur von der Oberklasse zu den Unterklassen, spricht man davon, dass die Unterklassen die Oberklasse spezialisieren. Betrachtet man die Vererbungsstruktur von den Unterklassen zur Oberklasse, so bedeutet dies, dass die Oberklasse ihre gemeinsamen Unterklassen generalisiert. Das Besondere an der Vererbung ist, dass alle Unterklassen (auch über mehrere Ebenen hinweg) die Attribute und Methoden der Oberklasse besitzen, ohne dass diese erneut modelliert oder implementiert werden müssen.