



Technische Universität Berlin

Software and Embedded Systems Engineering Group

Prof. Dr. Sabine Glesner

[www.sese.tu-berlin.de](http://www.sese.tu-berlin.de)

Sekr. TEL 12-4

Ernst-Reuter-Platz 7

10587 Berlin



# Softwaretechnik und Programmierparadigmen WiSe 2023/2024

Prof. Dr. Sabine Glesner  
Simon Schwan  
Julian Klein

## Übungsblatt 12



Eine Autowerkstatt möchte die Abfertigung ihrer Aufträge komfortabel mit einer Software verwalten. Dazu können Mitarbeiter:innen (Employee) im System Kunden und Kundinnen (Customer) anlegen und ihnen Fahrzeuge zuordnen. Für neue Kunden und Kundinnen werden jeweils ein Name, eine Telefonnummer und eine Rechnungsadresse gespeichert und die Fahrzeuge werden mit Kennzeichen und Typ registriert.

Ein Auftrag kann entweder eine Inspektion, ein Reifenwechsel oder eine Reparatur sein. Einem neuen Auftrag wird ein Preis, ein Fahrzeug und automatisch ein Datumstempel zugewiesen. Eine Reparatur erhält außerdem eine genaue Tätigkeitsbeschreibung. Ein Auftrag kann von Mitarbeitenden als beendet markiert werden. In diesem Fall wird der Kunde/die Kundin automatisch vom System benachrichtigt. Außerdem wird für den Auftrag vermerkt, welche/r Mitarbeiter:in ihn beendet hat.




Um Missbrauch vorzubeugen, müssen sich Mitarbeitende am Browser mit ID und Passwort sicher anmelden. Ein/e Administrator:in kann Mitarbeitende anlegen und entfernen.

---

### Schlüssel:

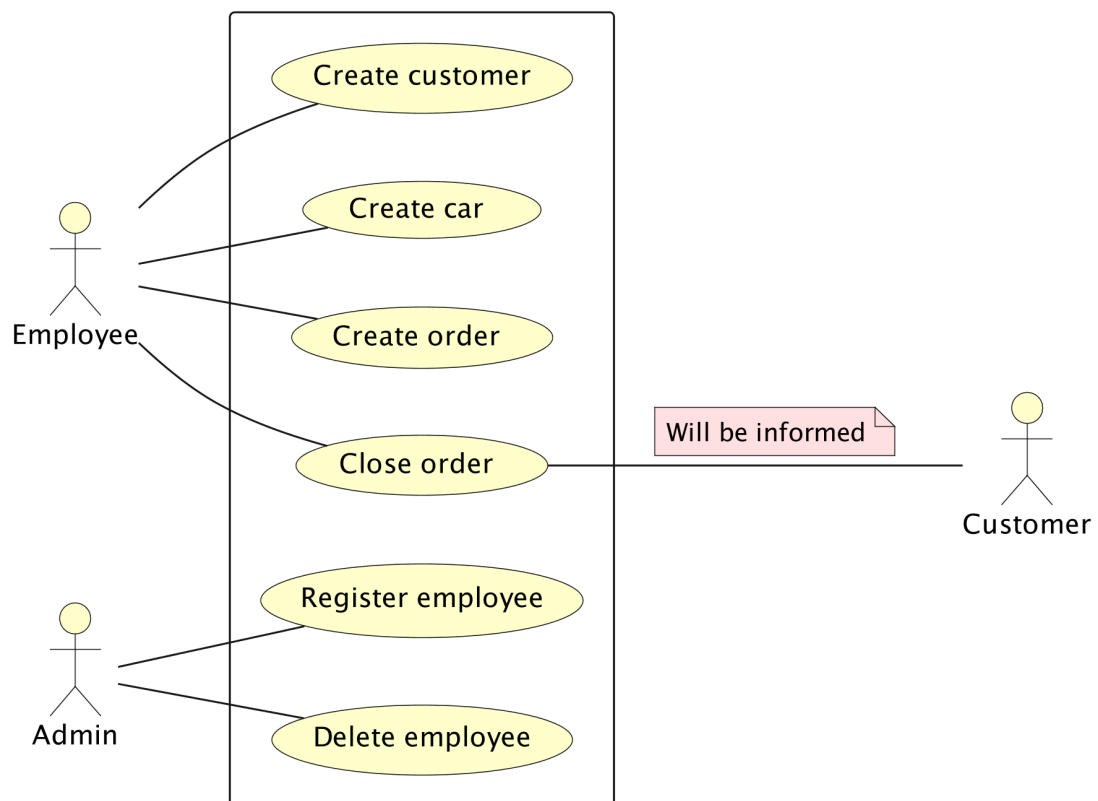
-  Ein ergänzendes Video wird zur Vor- oder Nachbereitung veröffentlicht.
-  Wird im Tutorium besprochen.


## 1. Use-Case-Diagramm

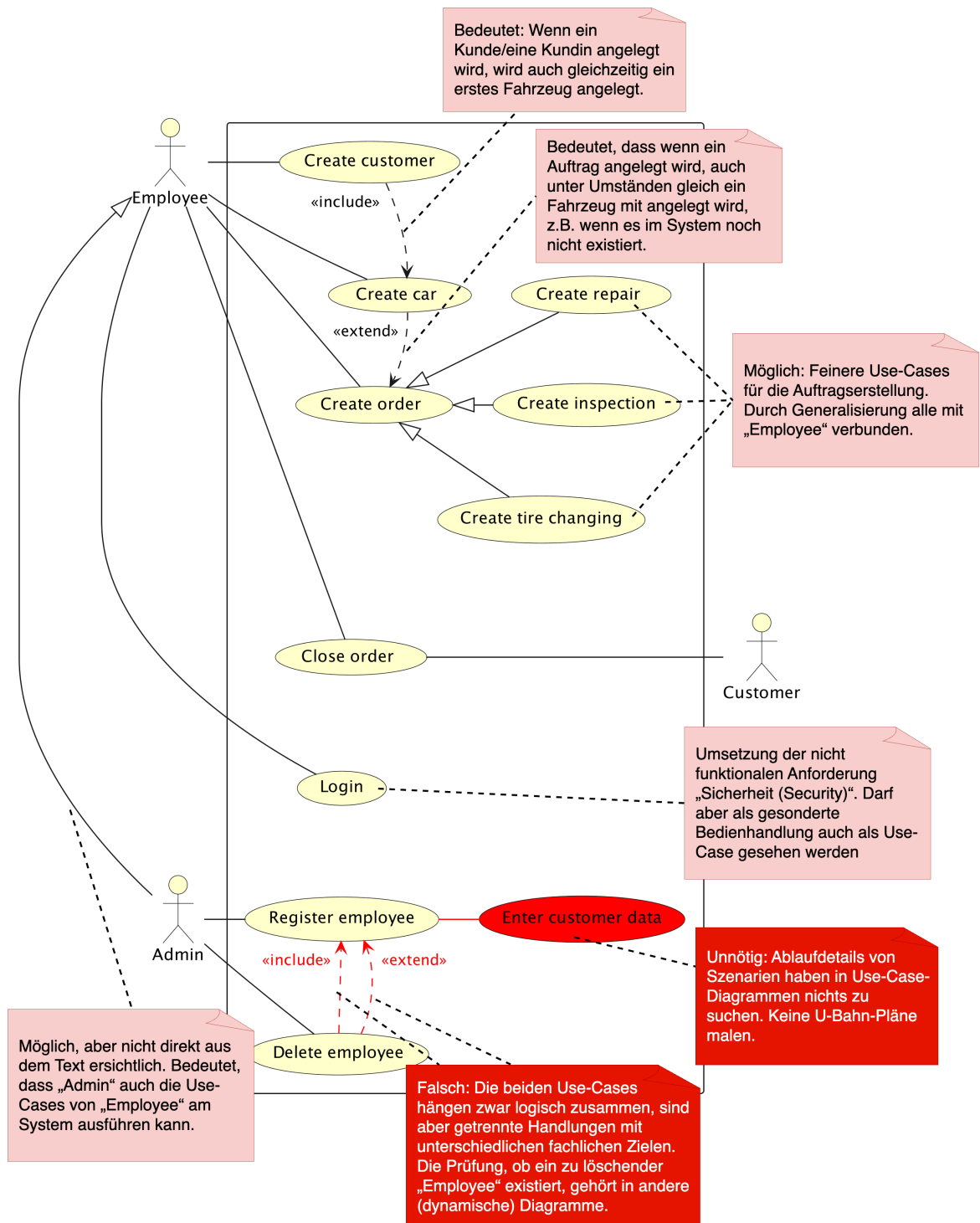
- a) Erstellt aus den Anforderungen ein Use-Case-Diagramm, in dem beteiligte Akteure/Aktuerinnen und Anwendungsfälle enthalten sind. 
- b) Überlegt, ob Beziehungen zwischen Use-Cases die Beschreibung der Anforderungen sinnvoll erweitern würden.  

– **Lösung Anfang** – – 30 min –

- a) Lösung siehe Diagramm:







- b) Möglichkeiten der Erweiterung kurz bzw. nach Anfrage diskutieren. Für die vollständige Diskussion auf den Video verweisen. 



– Lösung Ende –

## 2. Klassendiagramme

- a) Erstellt aus dem Text ein Klassendiagramm, das alle Informationen aus dem Anforderungstext enthält, sofern sie abbildbar sind. 
- b) An welchen Stellen hattet ihr verschiedene Möglichkeiten zur Modellierung des Klassendiagramms? Diskutiert alternative Entwurfsentscheidungen.  
- c) Überlegt, wie Stereotype nach dem *entity-boundary-control* Pattern auf unser Beispiel angewendet werden könnten. 

– Lösung Anfang –

### 45 min Klassendiagramm


- a) Es sollen schrittweise Elemente aus dem Text extrahiert werden und dabei möglichst schon sichtbar werden was wichtig ist und was nicht. Einige Aspekte werden im Video zur Nachbereitung besprochen, insbesondere zur Bewertung von Testaufgaben (was ist Pflicht, was ist falsch). Diesen können gerne auch im Tutorium besprochen werden, aber für Details kann man auf die Videos verweisen.

**Klassen:** Alle Substantive (mit Prüfung ob sie für das System eine Rolle spielen.)


**Generalisierung:** Alternativen auszeigen - was kann durch Gen/Spec eigentlich verallgemeinert werden

**Attribute:** Alternativen bei simplen Klassen/Attributen z.b. Rechnungsadresse=String oder Klasse diskutieren. Ein bisschen auf die Typen eingehen


**Operationen:** Use-Cases in den entsprechenden (Actor-)Klassen als Operationen unterbringen. Dazu sagen: Hier muss nicht jede Hilfsfunktion stehen. Wichtig sind Use-Cases, Szenarien, wesentliche Operationen die sich aus späteren Diagrammen ergeben.


Wir sagen dazu wenn wir Operationen sehen wollen oder wenn sie für die Übersichtlichkeit weggelassen werden können. 

**Assoziationen:** Verbinden was zusammen gehört. Beziehungen zwischen Datenobjekten stehen meist direkt im Text.

Leserichtung und Verb sind schön aber nicht Pflicht. Rollen sind nur Pflicht, wenn mehrdeutige Assoziationen existieren. 


**Multiplizitäten:** Für jedes Assoziationsende diskutieren. Weglassen bedeutet “unterspezifiziert”, kann also in der Implementierung entsprechend wie beliebige Multiplizitäten umgesetzt oder auch ganz weggelassen werden.

Generell nicht erforderlich für ein Modell, bei uns können sie aber in der Aufgabenstellung gefordert werden. 

**Aggregation/Komposition:** Stellen finden die Sinn machen und gut erklären. Klar machen, dass es keine Pflicht ist diese Features unterzubringen und dass das auch durch Multiplizitäten äquivalent beschrieben werden kann. Deutlich machen, 

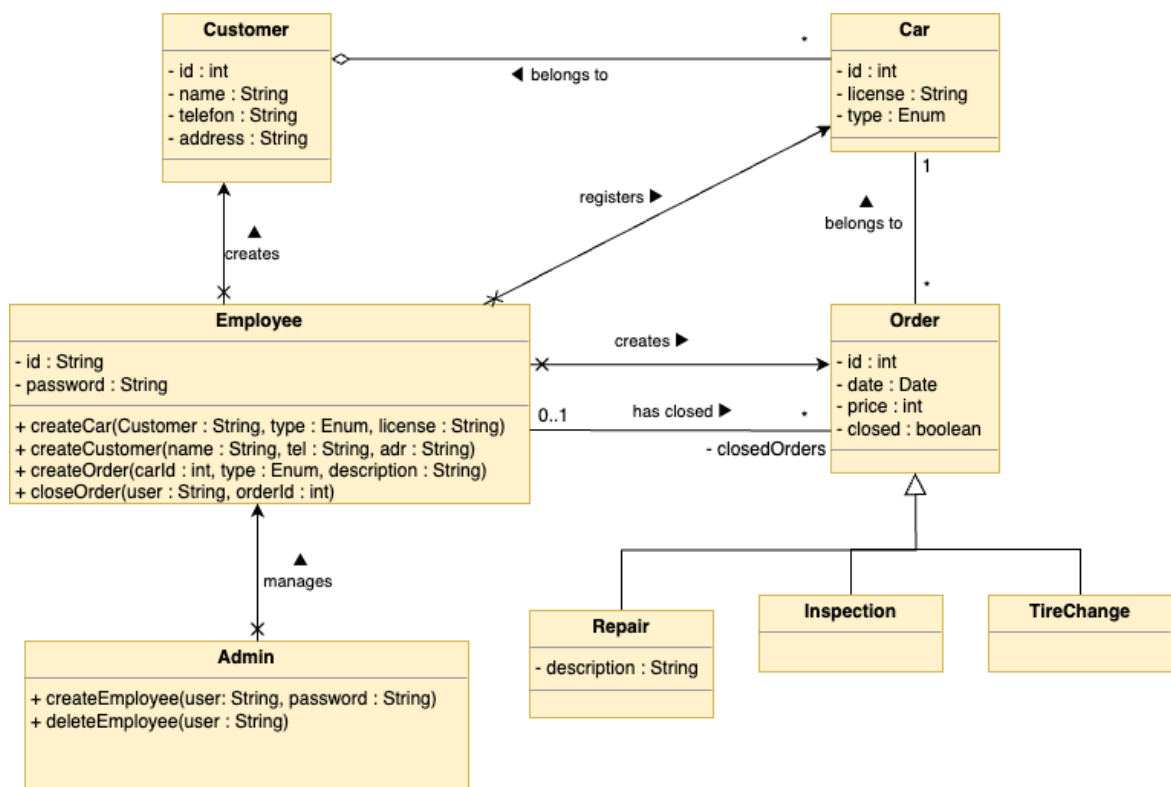
dass es (zumindest in Java) keine Entsprechung gibt, die sich von Multiplizitäten unterscheidet. Also Zusatz-Info.

**Navigationsrichtung:** Weglassen ist eine Unterspezifikation, d.h. beides ist möglich. Von Controller-artigen Klassen aus einseitig einschränken ist häufig gut. (Eingeschränkte Enden und gleichzeitig Multiplizitäten widersprechen sich zumindest aus Implementierungs-Sicht; kann aber sinnvoll sein um zu beschreiben dass ein Produkt z.B. in mehreren Warenkörben sein kann, die es nicht kennt oder dass es nur einen OnlineShop gibt, auf den vom Kunden aus kein Zugriff besteht).

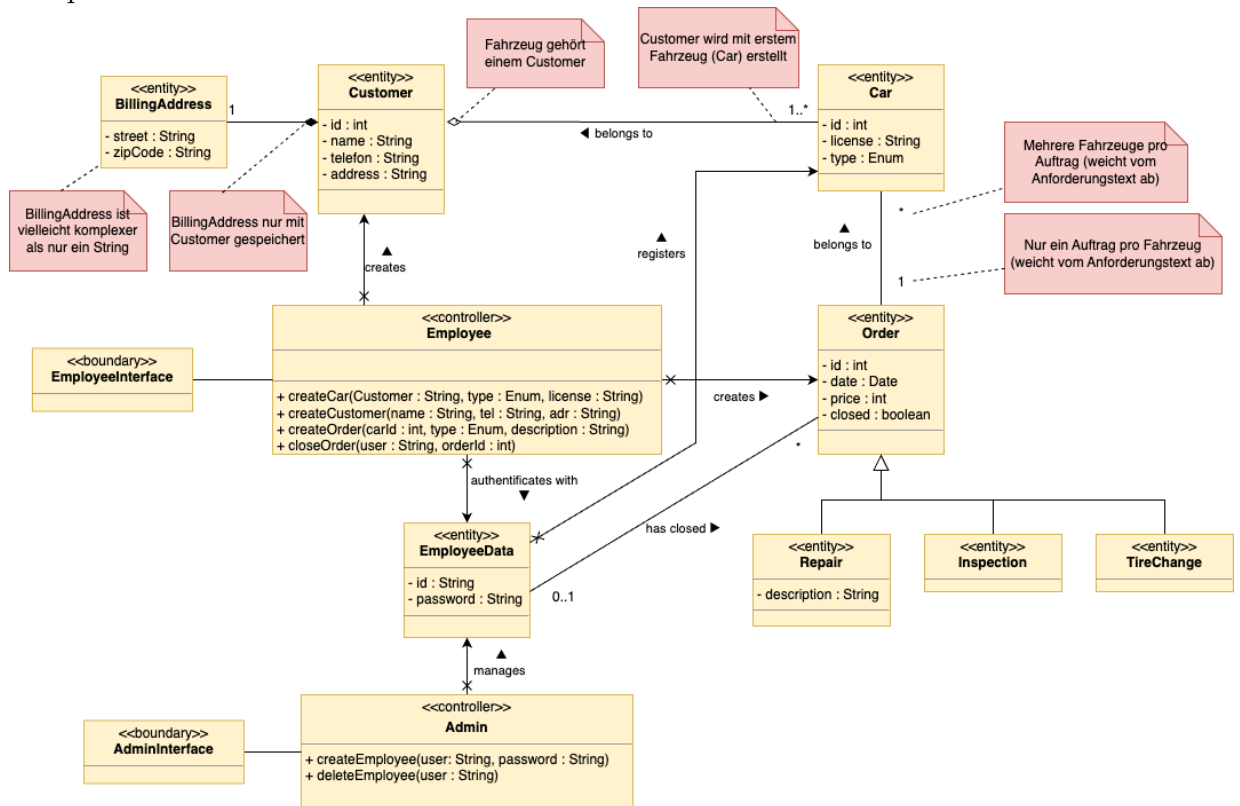
Navigation in Use-Cases (was brauche ich dafür) erarbeiten. Generell nicht erforderlich für ein Modell, bei uns kann es aber an der Aufgabenstellung erfordert werden. 

Operationen können weggelassen werden um nicht zu viel tippen zu müssen. Customer ist hierbei mit Absicht control und entity. Damit wir das später sinnvoll trennen können. Wenn sich das aber in der Mitarbeit anders ergibt ist das ok.

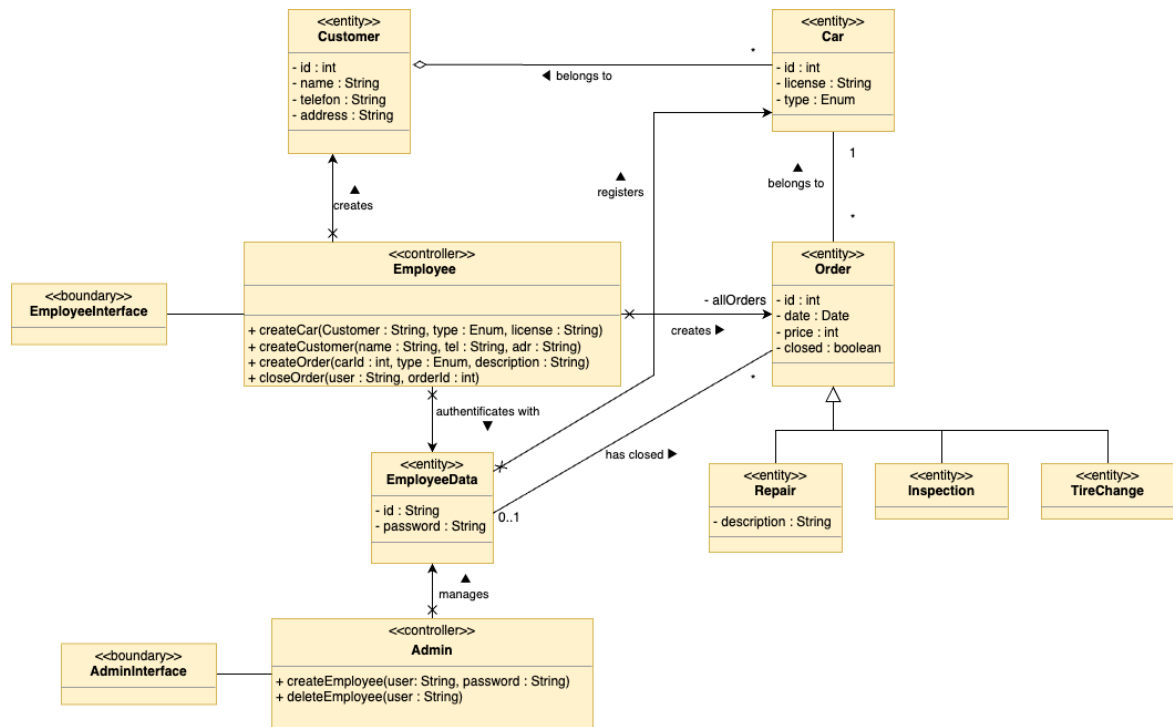
Beispiel-Lösung:



b) Ein paar Alternativen für die Diskussion:



- c) Beispiel-Lösung mit Stereotypen und Aufteilung der Customer-Klasse in Daten und Controller (Um dem ECB-Pattern zu genügen und zu persistierende Daten sauber von der Controller-Schicht zu trennen):



Noch einmal drüber reden wie das ECB-Pattern gemeint ist: Nicht dogmatisch nur Methoden in Controller und Daten in Entities auftrennen. Sondern: Die Entities beinhalten (persistente) Daten und die Controller übergeordnete Funktionalität, also insbesondere die Funktionen, die als Einsprung für die Use-Cases gelten.

Entities können aber auch Operationen haben (sonst wäre das keine Objektorientierung), diese arbeiten aber dann auf der lokalen Datensicht des jeweiligen Objekts. Controller dürfen auch (transiente) Attribute haben wenn das hilft. Assoziationen (mit 1 → \*) haben sie ja schließlich auch.



- Lösung Ende –
- Lösung Anfang –

– Lösung Ende –

### 3. Implementierung

Diskutiert wie eine mögliche Implementierung eures Klassenmodells in Java aussehen könnte. 

– Lösung Anfang –

– 15 min –

Teilweise zusammen in Java entwickeln, siehe Beispiel-Implementierung (Java). Was dabei klar werden soll:

- es gibt nur entweder Attribut oder Assoziation
- Ein Attribut kann einfach mit Typ als Member-Variable in die Klasse. Kurz erwähnen dass auch Sichtbarkeit ungefähr von UML übernommen werden kann.
- Eine Assoziation ist je nach Spezifikation der Navigation durch Attribute/Member-Variablen in einer oder beiden beteiligten Klassen darzustellen. Namen sind entweder frei wählbar oder benannte Assoziationsenden. Multiplizitäten bestimmen Typ:
- Bei 1 und 0..1 entspricht der Typ der Klasse auf der anderen Seite der Assoziation. bei 1 muss der Entwickler sicherstellen, dass das Attribut nie null wird.
- Bei Multiplizitäten  $> 1$  muss ein Collection-Typ oder ein Array erstellt werden. In Java muss der Entwickler darauf achten, dass m..n Multiplizitäten eingehalten werden.
- Generalisierung z.B. durch Vererbung. Abstrakte Klassen oder Interfaces verwenden wenn die Oberklasse nicht selbst instanziiert werden kann. Enums funktionieren nur in Ausnahmefällen.

– Lösung Ende –