Objektorientierte Systementwicklung

Assoziationen

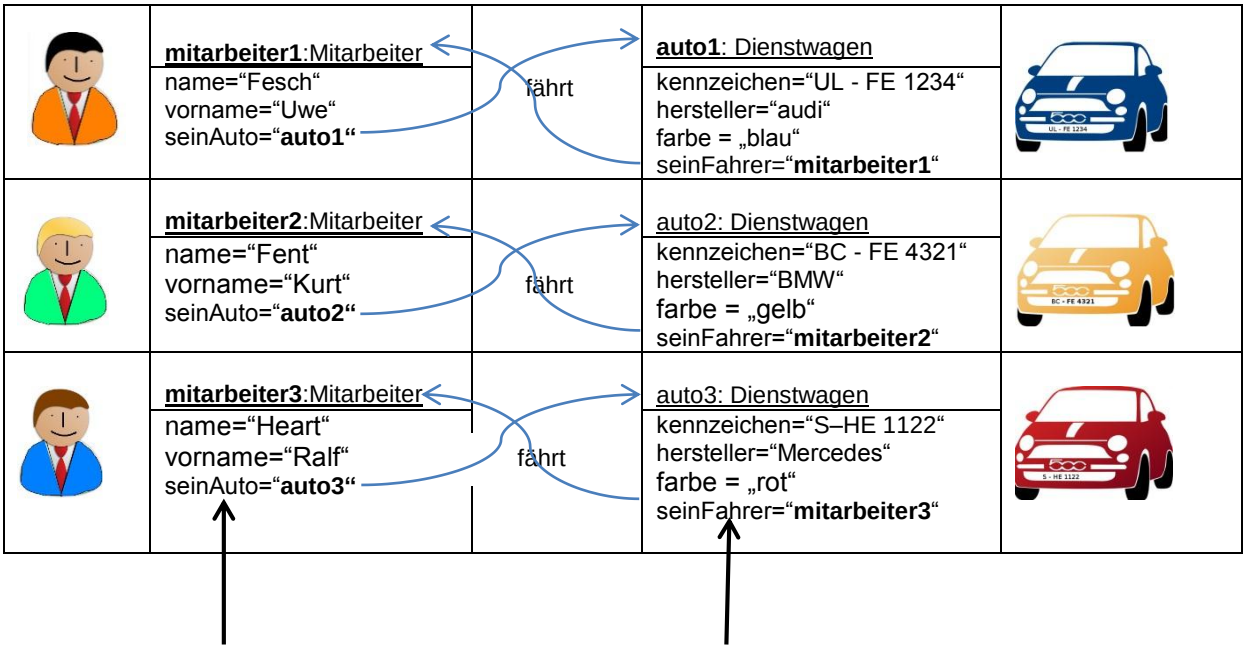
Inhaltsverzeichnis

1 Sachverhalt:

Die Firma GeLA GmbH stellt ihren drei Außendienstmitarbeitern Geschäftswagen zur Verfügung. Folgende Szenarien sind denkbar.

Jeder Mitarbeiter/in fährt immer mit demselben Auto, ein Auto wird auch immer vom selben Mitarbeiter gefahren Von den Mitarbeiterobjekten aus kann auf Autoobjekte zugegriffen werden und umgekehrt.

Die nachfolgende Darstellung zeigt, wie die Beziehung zwischen den Objekten hergestellt wird.



Soll von den Mitarbeiterobjekten aus auf die Autoobjekte zugegriffen werden, geschieht dies über das **Referenzattribut** *seinAuto*, dessen Attributwert das referenzierte Autoobjekt ist.

Soll von den Autoobjekten aus auf die Mitarbeiterobjekte zugegriffen werden, geschieht dies über das **Referenzattribut** *seinFahrer*, dessen Attributwert der referenzierte Mitarbeiter ist.

Die Referenzattribute haben als Datentyp die Klasse des verbundenen Objekts. Das Attribut *seinAuto* hat also den Datentyp *Auto*, das Attribut *seinFahrer* den Datentyp *Mitarbeiter*.

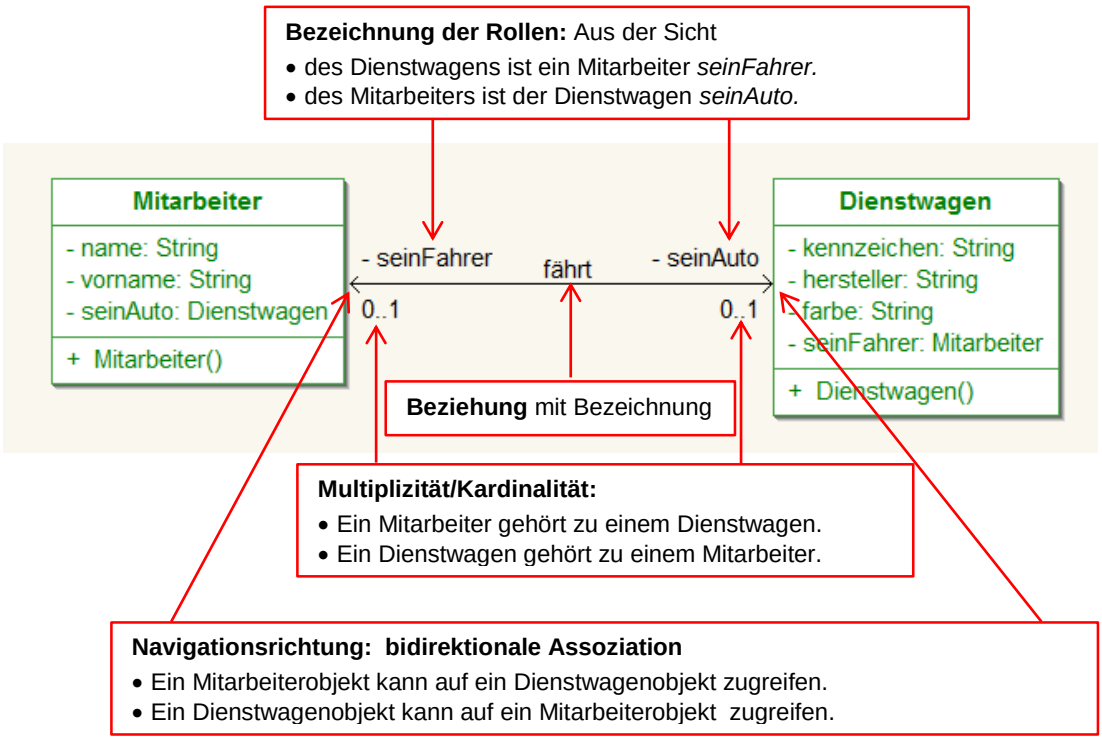
2 Assoziationen mit der Multiplizität 1 (1 – 1 - Assoziationen)

2.1 Beschreibung der 1 – 1 - Assoziation

Eine Beziehung zwischen zwei oder mehreren Klassen wird als **Assoziation** bezeichnet. Die Assoziation beschreibt die Beziehung aller Objekte der beteiligten Klassen.

Assoziationen werden auf Objektebene hergestellt und auf Klassenebene modelliert.

In der nachfolgenden Abbildung werden die wichtigsten Notationen beim Modellieren von Assoziationen dargestellt und erläutert.

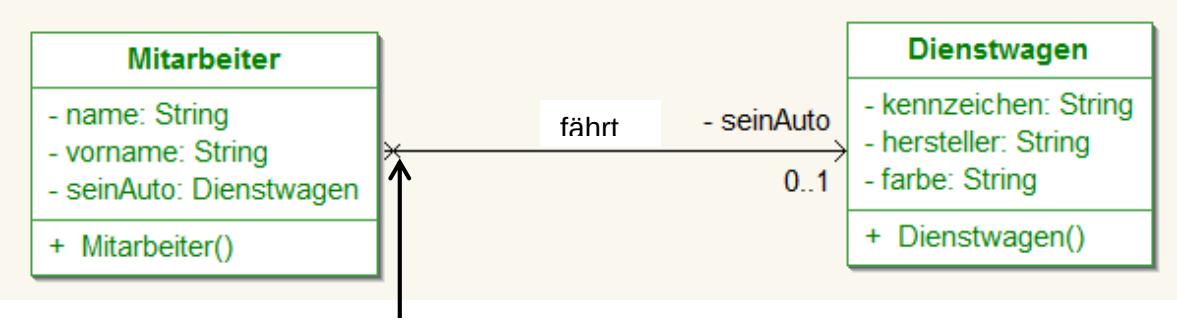


Beim Modellieren von Assoziationen im UML- Klassendiagramms müssen folgende Festlegungen getroffen werden:

- **Navigierbarkeit:** Dabei wird unterschieden in **unidirektionale** und **bidirektionale Assoziationen**.

  Zwischen den Klassen *Mitarbeiter* und *Dienstwagen* besteht eine *bidirektionale Assoziation*. Es kann von *Mitarbeiter* zu *Dienstwagen* navigieren werden, da jedes Objekt aus der Klasse *Mitarbeiter* eine Referenz zu genau einem Objekt aus der Klasse *Dienstwagen* hat. Umgekehrt weiß aber ein Objekt aus der Klasse *Dienstwagen* auch, welche Objekte ihm aus der Klasse *Mitarbeiter* zugeordnet sind.

  In der nächsten Abbildung ist eine **unidirektionale Assoziation** dargestellt. In der Klasse *Dienstwagen* gibt es keine Referenz auf die Klasse *Mitarbeiter*, es kann also keine Objektverbindung eines Dienstwagenobjekts zu einem Mitarbeiterobjekt hergestellt werden.



Unidirektionale Assoziation: Die Navigation ist nur in eine Richtung möglich. Einem Dienstwagen kann kein Fahrer zugeordnet werden, wohl aber einem Fahrer ein Dienstwagen.

- **Beziehung:** Die Beziehung, die durch die Assoziation festgelegt wird, wird mit einer Linie zwischen den beteiligten Klasse dargestellt. Eine Assoziation kann zusätzlich auch bezeichnet werden, beispielsweise mit „fährt“.

- **Rolle:** Rollennamen beschreiben die Bedeutung der Objekte der assoziierten Klasse näher. So ist beispielsweise ein Dienstwagen für einen Mitarbeiter „*seinAuto*“ oder ein Mitarbeiter für einen Dienstwagen „*seinFahrer*“.

- **Multiplizität:** Sie gibt an, mit wie vielen Objekten der assoziierten Klasse ein Objekt verbunden werden kann. Die Multiplizität in der Abbildung auf Seite 4 ist auf jeder Seite mit "0.. 1" bezeichnet, was bedeutet, dass ein Objekt der Klasse *Mitarbeiter* mit einem Objekt der *Klasse Dienstwagen* assoziiert sein kann.
  Softwaretechnisch wird diese Assoziation mit dem Referenzattribut vom Typ der jeweils assoziierten Klasse verwirklicht. So erhält die Klasse *Mitarbeiter* das Attribut *seinAuto* vom Typ *Dienstwagen* und die Klasse *Dienstwagen* das Attribut *seinFahrer* vom Typ *Mitarbeiter*.

  Mögliche Multiplizitäten sind:

  - 0..1 kein oder ein assoziiertes Objekt
  - 1 genau ein assoziiertes Objekt
  - * kein, ein oder beliebig viele assoziierte Objekte
  - 1..* ein oder beliebig viele assoziierte Objekte
  - n..m von n bis m assoziierte Objekte, wobei n, m Є IN

  Beispiele und Lösungen zu diesen Multiplizitäten werden in späteren „Szenarien“ vorgestellt.

2.2 Implementieren einer 1 – 1 – Assoziation

Die im Szenario auf Seite 3 dargestellten Objektverbindungen zeigen, wie die Dienstwagen den Mitarbeitern zugeordnet sind. Es soll die Zuordnung in beiden Richtungen möglich sein, das bedeutet, dass sowohl die Dienstwagen den Mitarbeitern, als auch die Mitarbeiter den Dienstwagen zugeordnet werden (= bidirektionale Assoziation).

Mitarbeiter	**Beziehung** ⟷	**Auto**
Uwe Fesch	*fährt*	Audi, blau, UL - FE 1234
Kurt Fent	*fährt*	BMW, gelb, BC - FE 4321
Ralf Heart	*fährt*	Mercedes, rot, S – HE 1122

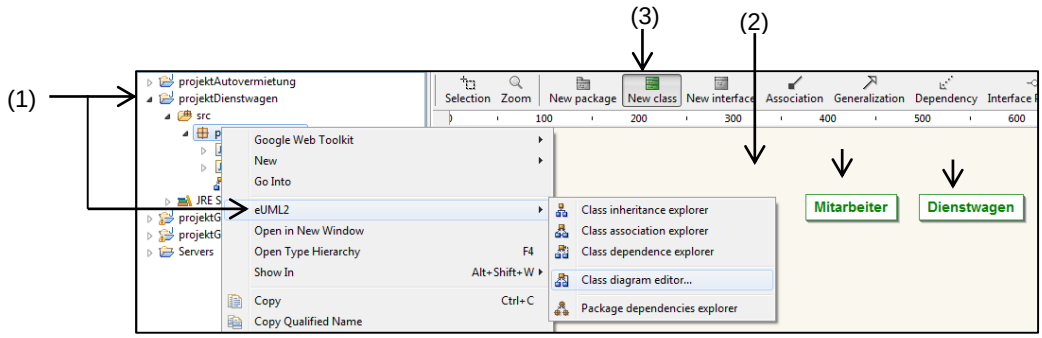
Folgende Aufgaben sind zu bearbeiten:

- Das Projekt *projektDienstwagen1* mit dem Paket *paketDienstwagen1* anlegen.
- Mithilfe des UML-Editors *eUML2* die Klassen *Mitarbeiter* und *Dienstwagen* wie auf Seite 4 dargestellt modellieren.
- Assoziationen modellieren. Dabei werden die Referenzattribute angelegt und die Zugriffmethoden generiert.
- In der Klasse *Startklasse* die Objekte mit den Objektverbindungen anlegen und im Konsolenfenster anzeigen.

Lösung:

- Projektstruktur bestehend aus dem Projekt *projektDienstwagen* und dem Paket *paketDienstwagen* anlegen. Mit dem Menübefehl **`File New Javaprojekt`** *projektDienstwagen* wird zuerst das Projekt und im Projekt dann mit dem Menübefehl **`File New Package`** *paketDienstwagen* das Paket angelegt. Es entsteht die nebenstehend abgebildete Projektstruktur.

- UML-Editor *eUML2* starten und die Klassen wie auf Seite 4 dargestellt modellieren. Zunächst werden nur die Attribute und Methoden, nicht die Assoziationen erstellt.

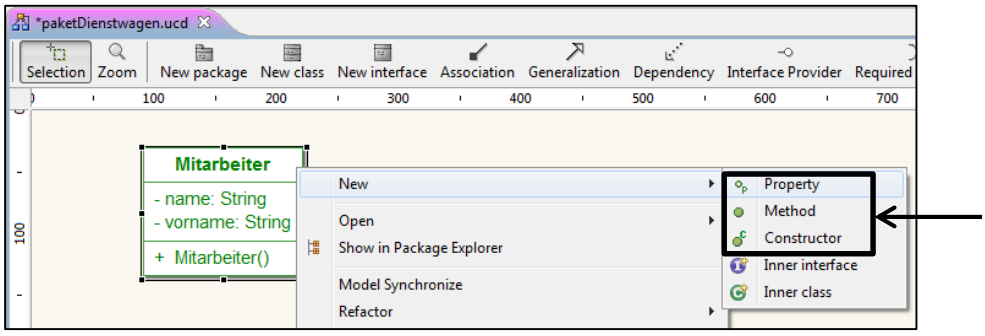


(1) Das Kontextmenü (Klick mit rechter Maustaste) zum Package *paketdienstwagen* öffnen und den Menüeintrag **`eUML2`** und darin den Menübefehl **`Class diagramm editor`** auswählen.

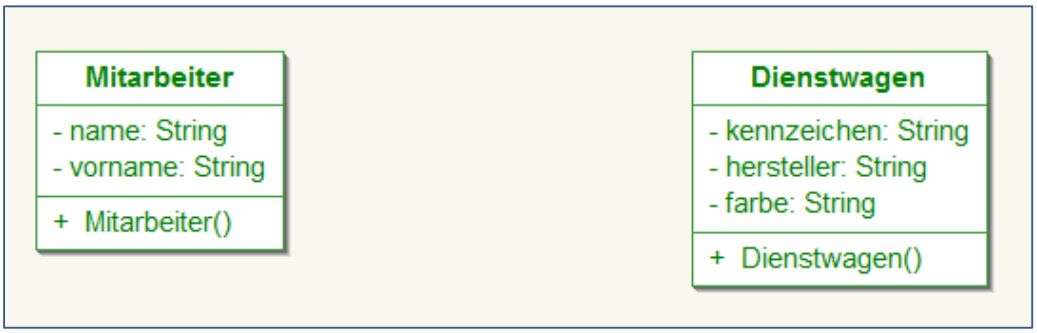
(2) Der UML-Editor wird gestartet und die Zeichenfläche, in der die Klassen modelliert werden können, wird bereitgestellt.

(3) Die Klassen können jetzt modelliert werden. Dazu wird der Button [**`New Class`**] gedrückt und die Klasse als Rechteck in der Zeichenfläche aufgezogen. Anschließend werden dann die Attribute und Methoden modelliert. Dazu wird die Klasse angeklickt und das Kontextmenü mit einem Klick auf die rechte Maustaste geöffnet.

Mit den Menübefehlen **`New Property`**, **`New Method`** und **`New Constructor`** (siehe nachfolgende Abbildung) wird die Klasse modelliert.



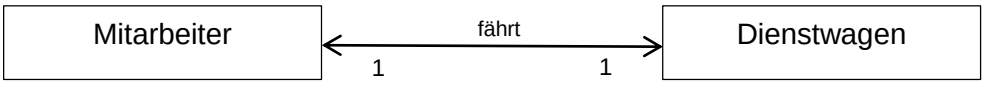
Als Ergebnis erhält man folgendes UML-Klassendiagramm.



➢ Assoziationen modellieren

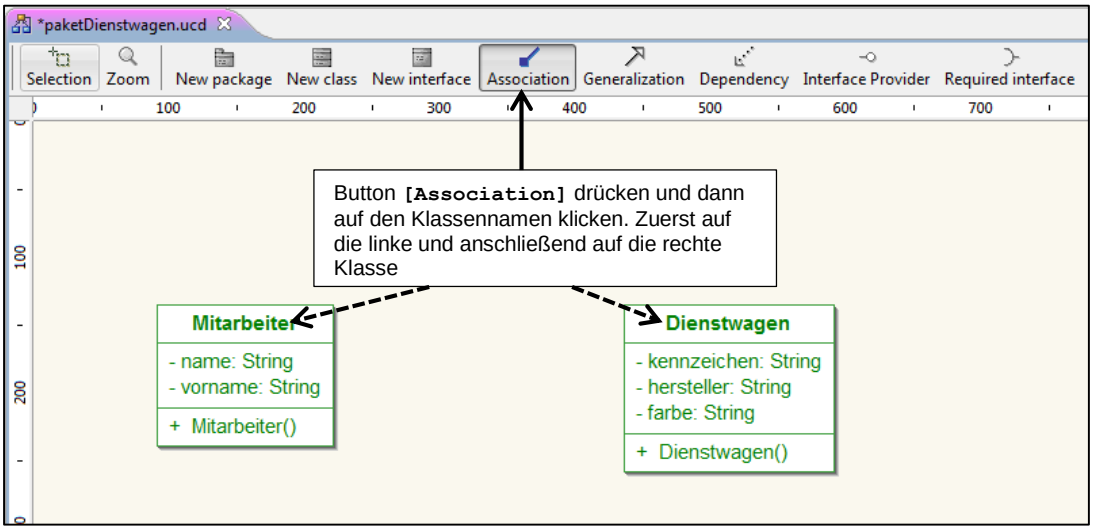
Die Assoziation wird ebenfalls über das grafische Werkzeug implementiert. Es werden dann die Referenzattribute deklariert und die zugehörigen Zugriffsmethoden *(„setter“* und *„getter“* erzeugt.

Im vorliegenden Beispiel wird eine bidirektionale 1 – 1 - Assoziation erzeugt.

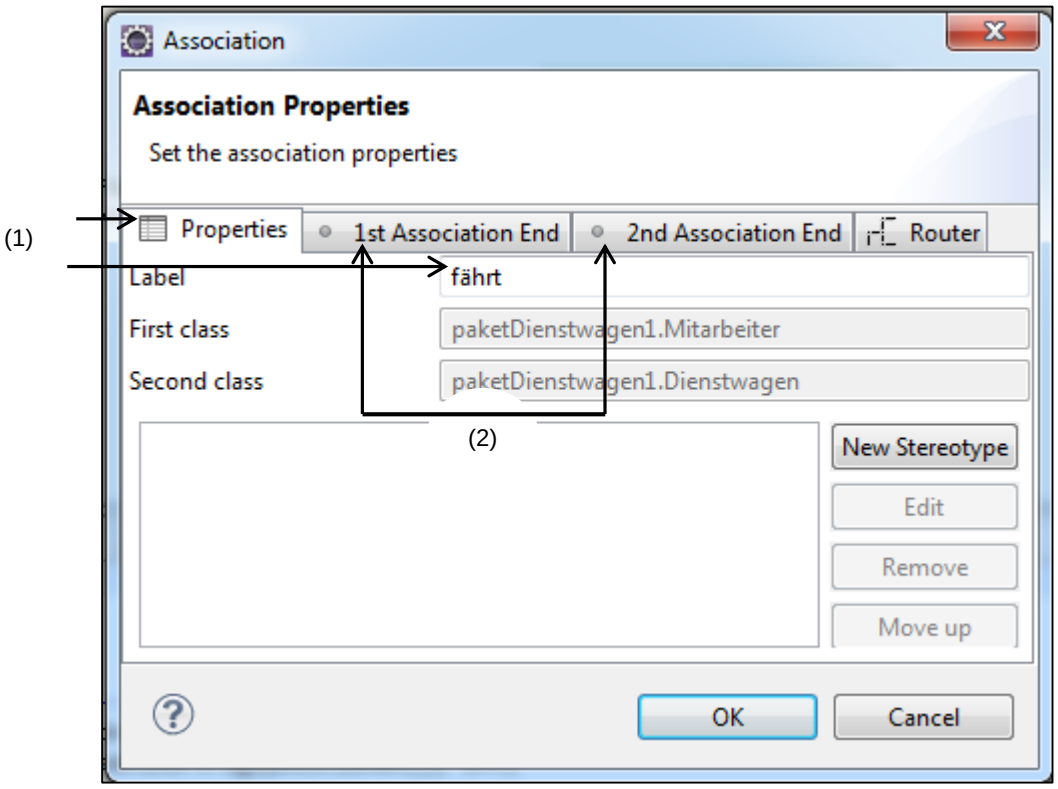


- Ein Mitarbeiterobjekt ist einem Dienstwagenobjekt zugeordnet, das bedeutet, ein Mitarbeiter fährt mit einem bestimmten Auto.
- Ein Dienstwagenobjekt ist einem Mitarbeiter zugeordnet, das bedeutet, ein Dienstwagen wird von einem bestimmten Mitarbeiter gefahren.

Um die Assoziation einzurichten, muss der Button **`[Association]`** gedrückt werden. Der Mauszeiger erscheint als Stecker, sobald er in die Zeichenfläche bewegt wird.
Bei der 1 – 1 - Assoziation wird zuerst auf den Klassennamen der linken und danach auf den Klassennamen der rechten Klasse geklickt.



Es öffnet sich ein Wizard, das es erlaubt die Assoziation zu spezifizieren. Der Label soll "*fährt*" lauten.

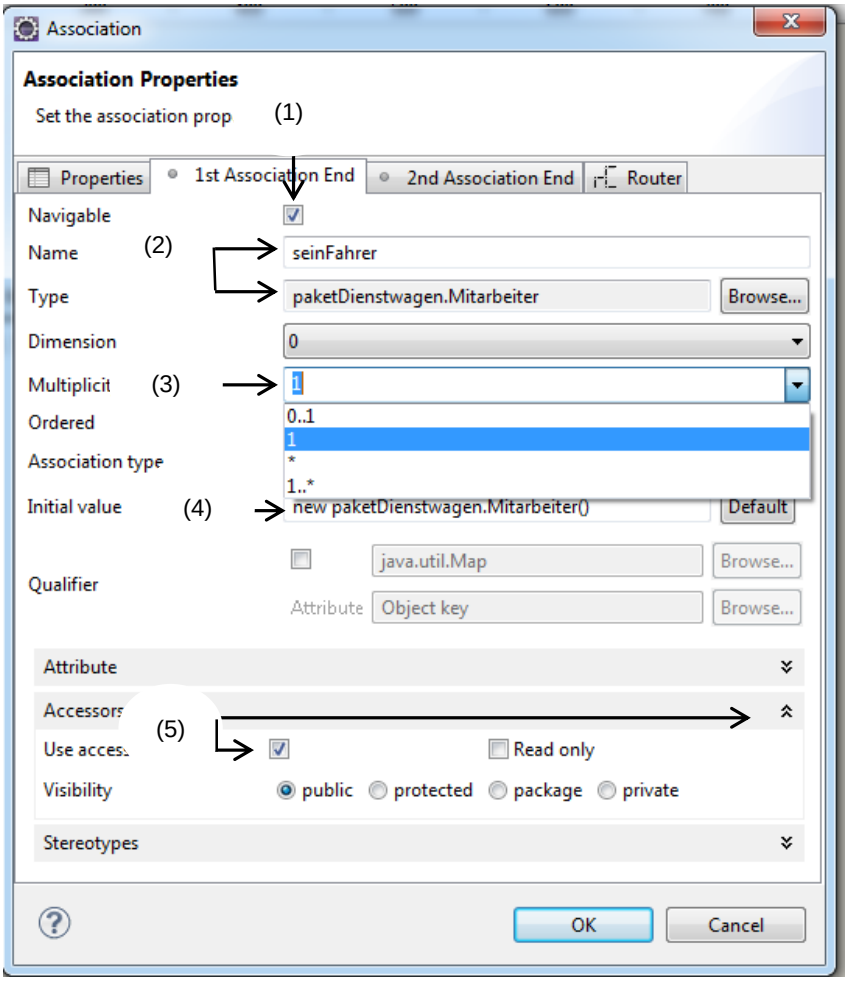


(1) Im Register **`Properties`** kann ein Name (Label) für die Assoziation eingetragen werden.

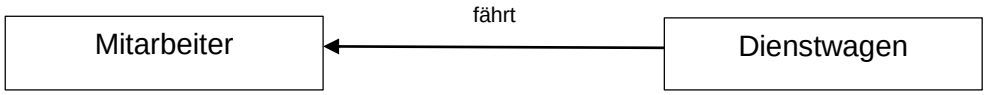
(2) Mit den Buttons [**`1st Association End`**] und [**`2nd Association End`**] werden die Assoziationen beschrieben.

Das erste Assoziationsende (1st Association End)

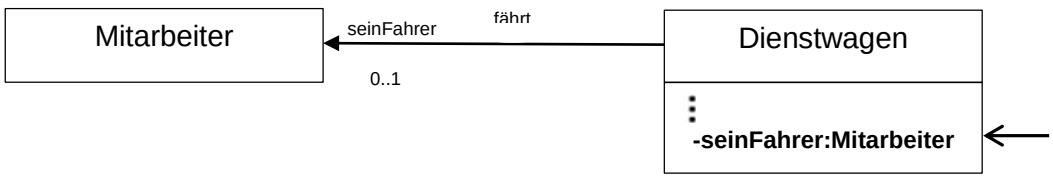
Wird der Button `[1st Association End]` geklickt, kann das Assoziationsende der ersten Klasse beschrieben werden.



(1) Mit dem Kontrollfeld *Navigable* wird die Navigationsrichtung freigegeben. Im vorliegenden Beispiel endet die Assoziation in der Klasse *Mitarbeiter*, es kann also von Dienstwagen zu Mitarbeiter navigiert werden.



(2) Im Eingabefeld *Name* wird die Rolle, die ein Objekt in der assoziierten Klasse spielt, festgelegt. Ein Mitarbeiter ist für ein Objekt der Klasse *Dienstwagen* sein Fahrer, weshalb als Rollenname *seinFahrer* gewählt wird. Der Rollenname wird auch als Name für das Referenzattribut benutzt. Mit dem Referenzattribut vom Typ der Klasse *Mitarbeiter* wird die Assoziation realisiert.



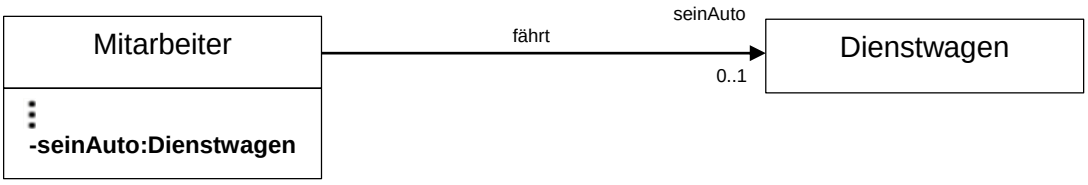
(3) Hier wird die Multiplizität ausgewählt. Zu einem Dienstwagen gehört ein Fahrer. Es ist aber auch denkbar, dass einem Dienstwagen kein Fahrer zugeordnet wird. Deshalb wird die Mulitplizität 0..1 ausgewählt.

(4) Im Eingabefeld **`Initial value`** wird durch Drücken des Buttons **`[Default]`** die Klasse angezeigt, aus der das an der Assoziation beteiligte Objekt stammt. Im vorliegenden Beispiel ist dies die Klasse *Mitarbeiter*.

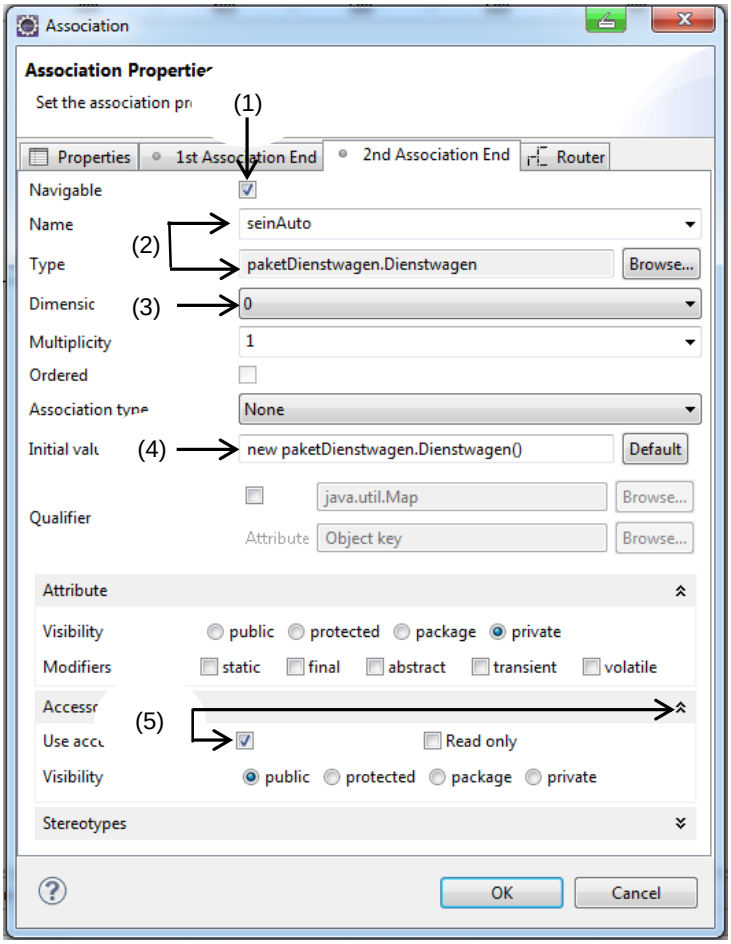
(5) Hier kann durch das Aktivieren beziehungsweise Deaktivieren des Kontrollfeldes **`Use accessors`** entschieden werden, ob zum erzeugten Referenzattribut die Zugriffsmethoden generiert werden sollen oder nicht. Standardmäßig wird vorgeschlagen, die Zugriffsmethoden zu generieren.

Das zweite Assoziationsende (2nd Association End)

Das 2. Assoziationsende wird nach einem Klick auf den Button **`[2nd Association End]`** beschrieben. Einem Mitarbeiter ist ein Dienstwagen zugeordnet. Dies wird in der Klasse *Mitarbeiter* durch das Attribut *seinAuto* vom Typ *Dienstwagen* realsiert.



Spezifizieren des Endes der zweiten Assoziation:



(1) Die Assoziation endet in der Klasse *Dienstwagen*, es kann also von Mitarbeiter zu Dienstwagen navigiert werden.

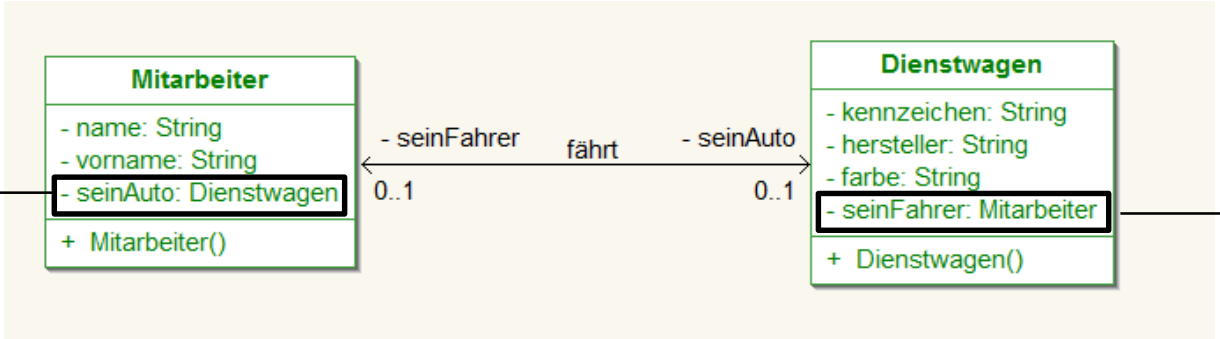
(2) Ein Dienstwagen ist für ein Objekt der Klasse *Mitarbeiter* sein Auto, weshalb als Rollenname *seinAuto* gewählt wird. Der Rollenname wird auch als Name für das Referenzattribut benutzt. Mit dem Referenzattribut vom Typ der Klasse *Dienstwagen* wird die Assoziation realisiert.

(3) Hier wird die Multiplizität ausgewählt. Zu einem Mitarbeiter gehört ein Auto, denkbar ist auch, dass ein Mitarbeiter nie ein Auto benutzt. Deshalb wird die Mulitplizität 0..1 ausgewählt.

(4) Im Eingabefeld **`Initial value`** wird durch Drücken des Buttons **`[Default]`** die Klasse angezeigt, aus der das an der Assoziation beteiligte Objekt stammt. Im vorliegenden Beispiel ist dies die Klasse *Dienstwagen*.

(5) Standardmäßig wird vorgeschlagen, die Zugriffsmethoden zu generieren.

Wenn beide Assoziationsenden modelliert sind, erhält man das folgende UML-Klassendiagramm:



Der generierte Javacode der modellierten Assoziationen:

Die Assoziation wird jeweils über das Referenzattribut realisiert. Im Referenzattribut können Objekte der assoziierten Klasse gespeichert und ausgelesen werden. Dazu werden die Zugriffsmethoden benötigt.

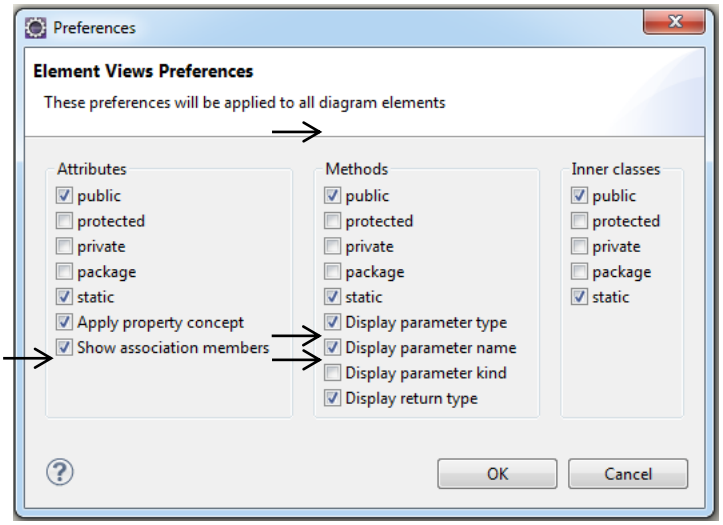
```
public class Mitarbeiter {
⋮
// Referenzattribut deklarieren
  private Dienstwagen seinAuto;

  // get-Methode (Attributwerte auslesen)

  public Dienstwagen getSeinAuto()
  {
       return seinAuto;
  }

  // set-Methode (Attributwerte setzen)
  public void setSeinAuto(Dienstwagen
                                seinAuto)
{
       this.seinAuto = seinAuto;
  }

}
```

```
public class Dienstwagen {
⋮
  // Referenzattribut deklarieren
  private Mitarbeiter seinFahrer ;

  // get-Methode (Attributwerte auslesen)

  public Mitarbeiter getSeinFahrer()
  {
       return seinFahrer;
  }
  // set-Methode (Attributwerte setzen)
  public void setSeinFahrer(Mitarbeiter
                              seinFahrer)
{
       this.seinFahrer = seinFahrer;
  }

}
```

Hinweis:
Wird das Klassendiagramm nicht korrekt dargestellt, weil die Referenzattribute nicht angezeigt werden, gibt es zwei Vorgehensweisen, um dies zu beheben.

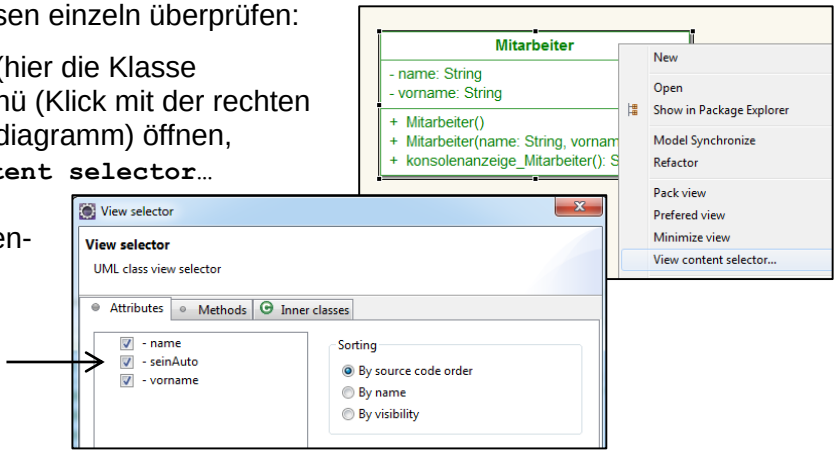
- Anpassen der Eigenschaften der Zeichnungsfläche:

  Dazu sind im Kontextmenü zur Entwurfsfläche des *eUML2*-Editors (mit der rechten Maustaste auf eine beliebige freie Stelle klicken) der Menübefehl **`Preferences ► Element views…`** auszuwählen und dann im Fenster **`Preferences`** die Option **`Show association members`** zu aktivieren. Neben dem Sichtbarmachen der Referenzattribute können hier auch noch weitere Festlegungen, wie das Anzeigen der Rückgabetypen der Methoden sowie der Namen und Typen der Parameter getroffen werden.



- Attributanzeige für die Klassen einzeln überprüfen:

  Für die betreffende Klasse (hier die Klasse *Mitarbeiter*) das Kontextmenü (Klick mit der rechten Maustaste auf das Klassendiagramm) öffnen, den Menübefehl **`View content selector…`** auswählen und im Fenster *View selector* die anzuzeigenden Attribute aktivieren.



2.3 Die 1 – 1 - Assoziationen in der Klasse *Startklasse* mit der main-Methode nutzen.

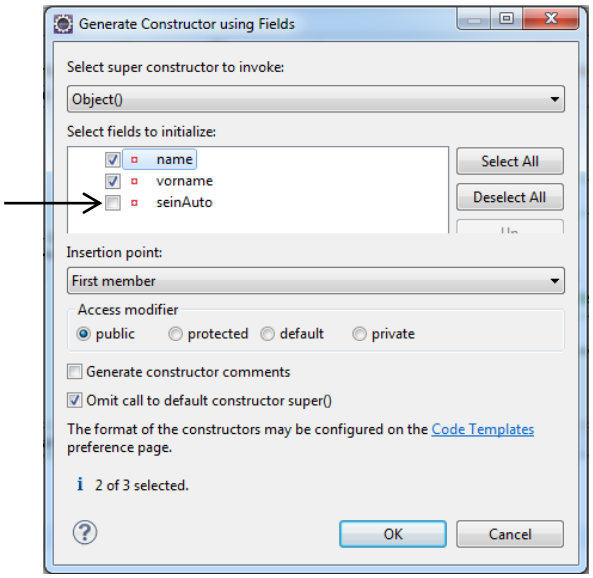
Die Klassen *Mitarbeiter* und *Dienstwagen* sollen nun benutzt werden. Dazu wird die Klasse *Startklasse*, welche die *main*-Methode enthält, verwendet.
In ihr werden Objekte erzeugt, verknüpft (assoziiert) und dann im Konsolenfenster ausgegeben.

Folgende Aufgaben sollen ausgeführt werden:

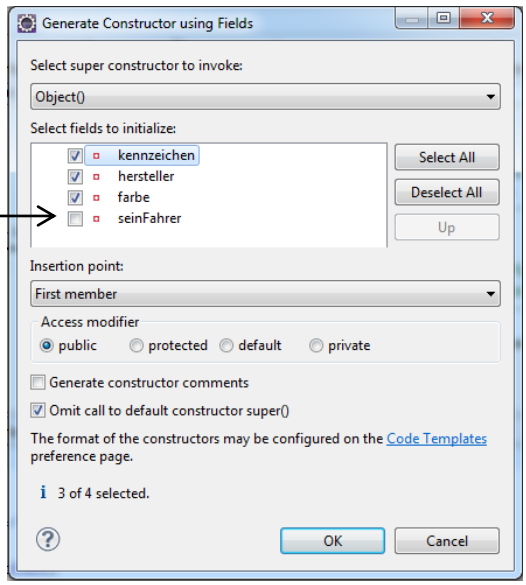
➢ Die im Sachverhalt auf Seite 3 dargestellten Objekte der Klassen *Mitarbeiter* und *Dienstwagen* sind zu erzeugen. Damit die Objekte mit ihren Attributwerten möglichst rasch erzeugt werden können, werden die beiden Klassen jeweils um einen angepassten Konstruktor erweitert. Angepasste Konstruktoren werden im Quellcode mit dem Menübefehl **`Generate Constructor using fields`** erzeugt.

Angepasste Konstruktoren für die Klassen:

Mitarbeiter



Dienstwagen



Javacode:

```
public Mitarbeiter(String name, String
                                 vorname)
{
  this.name = name;
  this.vorname = vorname;
}
```

```
public Dienstwagen(String kennzeichen,
      String hersteller, String farbe)
{
  this.kennzeichen = kennzeichen;
  this.hersteller = hersteller;
  this.farbe = farbe;
}
```

Hinweis:

„Muss-Assoziationen“ werden an dieser Stelle noch nicht thematisiert. Deshalb werden die Referenzattribute (*seinAuto* in der Klasse *Mitarbeiter* und *seinFahrer* in der Klasse *Dienstwagen*) nicht in den Konstruktor übernommen.

- Die beschriebenen Objektverbindungen (in beide Richtungen) sind vorzunehmen.
- Die Objektverbindungen sind im Konsolenfenster - wie in der nebenstehenden Abbildung dargestellt - auszugeben.

```
Mitarbeiter fährt mit Dienstwagen
---------------------------------
Uwe Fesch fährt mit Audi, UL-FE 1234,blau
Kurt Fent fährt mit BMW, BC-FE 4321,gelb
Ralf Heart fährt mit Mercedes, S - HE 1122,rot

Dienstwagen wird gefahren von Mitarbeiter
-----------------------------------------
Audi, UL-FE 1234 wird gefahren von Uwe Fesch
BMW, BC-FE 4321 wird gefahren von Kurt Fent
Mercedes, S - HE 1122 wird gefahren von Ralf Heart
```

Im Folgenden wird der kommentierte Quellcode der Klasse *Startklasse* aufgeführt.

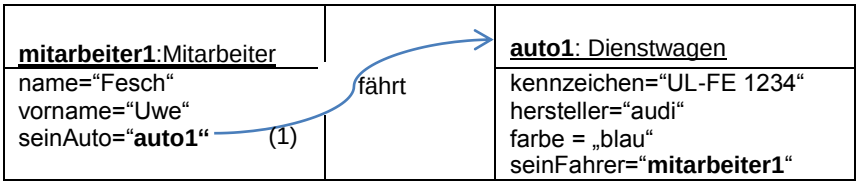
```
public static void main(String[] args)
        {
// Mitarbeiterobjekte mit dem angepassten  erzeugen
Mitarbeiter mitarbeiter1 = new Mitarbeiter("Fesch","Uwe");
Mitarbeiter mitarbeiter2 = new Mitarbeiter("Fent","Kurt");
Mitarbeiter mitarbeiter3 = new Mitarbeiter("Heart","Ralf");

// Dienstwagenobjekte mit dem angepassten Konstruktor erzeugen
Dienstwagen auto1 = new Dienstwagen("UL-FE 1234","Audi","blau");
Dienstwagen auto2 = new Dienstwagen("BC-FE 4321","BMW","gelb");
Dienstwagen auto3 = new Dienstwagen("S-HE 1122","Mercedes","rot");

// Assoziation 1: Mitarbeiter fährt mit Auto
mitarbeiter1.setSeinAuto(auto1);
mitarbeiter2.setSeinAuto(auto2);
mitarbeiter3.setSeinAuto(auto3);

// Assoziation 2: Auto wird gefahren von
auto1.setSeinFahrer(mitarbeiter1);
auto2.setSeinFahrer(mitarbeiter2);
auto3.setSeinFahrer(mitarbeiter3);

//Konsolenausgabe
System.out.println("Mitarbeiter fährt mit Dienstwagen ");
System.out.println("-----------------------------------");

// mitarbeiter1 fährt mit auto1
System.out.print(mitarbeiter1.getVorname()+" "+mitarbeiter1.getName()+" fährt mit ");
System.out.print(mitarbeiter1.getSeinAuto().getHersteller()+", ");
System.out.println(mitarbeiter1.getSeinAuto().getKennzeichen()+","+mitarbeiter1.getSeinAuto().
                                                                    getFarbe());
// mitarbeiter2 fährt mit auto2
System.out.print(mitarbeiter2.getVorname()+" "+mitarbeiter2.getName()+" fährt mit ");
System.out.print(mitarbeiter2.getSeinAuto().getHersteller()+", ");
System.out.println(mitarbeiter2.getSeinAuto().getKennzeichen()+","+mitarbeiter2.getSeinAuto().
getFarbe());
// mitarbeiter3 fährt mit auto3
System.out.print(mitarbeiter3.getVorname()+" "+mitarbeiter3.getName()+" fährt mit ");
System.out.print(mitarbeiter3.getSeinAuto().getHersteller()+", ");
System.out.println(mitarbeiter3.getSeinAuto().getKennzeichen()+","+mitarbeiter3.getSeinAuto().
getFarbe());

System.out.println();  // Leerzeile

System.out.println("Dienstwagen wird gefahren von Mitarbeiter");
System.out.println("-----------------------------------------");

// auto1 wird gefahren von mitarbeiter1
System.out.print(auto1.getHersteller()+", "+auto1.getKennzeichen()+" wird gefahren von ");
System.out.println(auto1.getSeinFahrer().getVorname()+ " "+ auto1.getSeinFahrer().getName());

// auto2 wird gefahren von mitarbeiter2
System.out.print(auto2.getHersteller()+", "+auto2.getKennzeichen()+" wird gefahren von ");
System.out.println(auto2.getSeinFahrer().getVorname()+ " "+ auto2.getSeinFahrer().getName());

// auto3 wird gefahren von mitarbeiter3
System.out.print(auto3.getHersteller()+", "+auto3.getKennzeichen()+" wird gefahren von ");
System.out.println(auto3.getSeinFahrer().getVorname()+ " "+ auto3.getSeinFahrer().getName());
        }
```

(1)

(2)

Erläuterung zu den Objektverbindungen und zum Zugriff auf die Attribute des assoziierten Objekts:

(1) Das Objekt *mitarbeiter1* ist mit dem Objekt *auto1* über das Attribut *seinAuto* assoziiert.



Das Referenzattribut *seinAuto* erhält als Wert das assoziierte Objekt *auto1.*

Javacode:

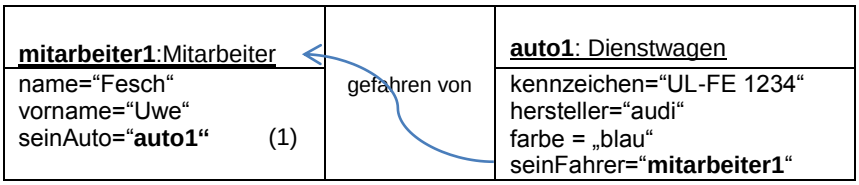
```
mitarbeiter1.setSeinAuto(auto1);
```

Auf die Attributwerte des assoziierten Objekts *auto1* wird mit Methoden zum Referenzattribut zugegriffen.

```
// mitarbeiter1 fährt mit auto1
System.out.print(mitarbeiter1.getVorname()+" "+mitarbeiter1.getName()+" fährt mit ");
System.out.print(mitarbeiter1.getSeinAuto().getHersteller()+", ");
System.out.print(mitarbeiter1.getSeinAuto().getKennzeichen());
System.out.println(mitarbeiter1.getSeinAuto().getFarbe());
```

Über die Methode getSeinAuto() können die Methoden des assoziierten Objekts *auto1* verwendet werden.

(2) Das Objekt auto1 ist mit dem Objekt *mitarbeiter1* über das Attribut *seinFahrer* assoziiert.



Das Referenzattribut *seinFahrer* erhält als Wert das assoziierte Objekt *mitarbeiter1.*

Javacode:

```
auto1.setSeinFahrer(mitarbeiter1);
```

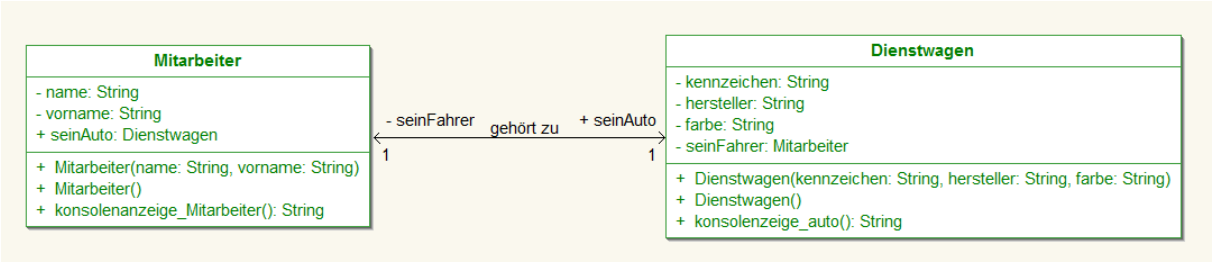
Auf die Attributwerte des assoziierten Objekts *mitarbeiter1* wird mit Methoden des

```
// auto1 wird gefahren von mitarbeiter1
System.out.print(auto1.getHersteller()+", "+auto1.getKennzeichen()+" wird gefahren
                                                                    von ");
System.out.print(auto1.getSeinFahrer().getVorname()+ " ");
System.out.println(auto1.getSeinFahrer().getName());
```

Über die Methode *getSeinFahrer()* können die Methoden des assoziierten Objekts *mitarbeiter1* verwendet werden.

Alternative:

Wird für die Konsolenanzeige der Attributwerte in den Klassen *Mitarbeiter* und *Dienstwagen* jeweils eine Methode erstellt, welche die Attributwerte als String-Werte liefert, können die Anweisungen zur Ausgabe in der Startklasse reduziert werden.



Methoden zur Rückgabe der Attributwerte als String:

```
public String konsolenanzeige_Mitarbeiter()
{
//Ausgabestring - Daten des Mitarbeiters
return vorname+" "+ this.name;
}
```

```
public String konsolenzeige_auto()
{
// Ausgabestring - Daten des Dienstwagens
return  this.kennzeichen+", "+
this.hersteller+", "+ this.farbe;
}
```

Anzeige der Objektverbindungen im Konsolenfenster am Beispiel der Objekte *mitarbeiter1* und auto1:

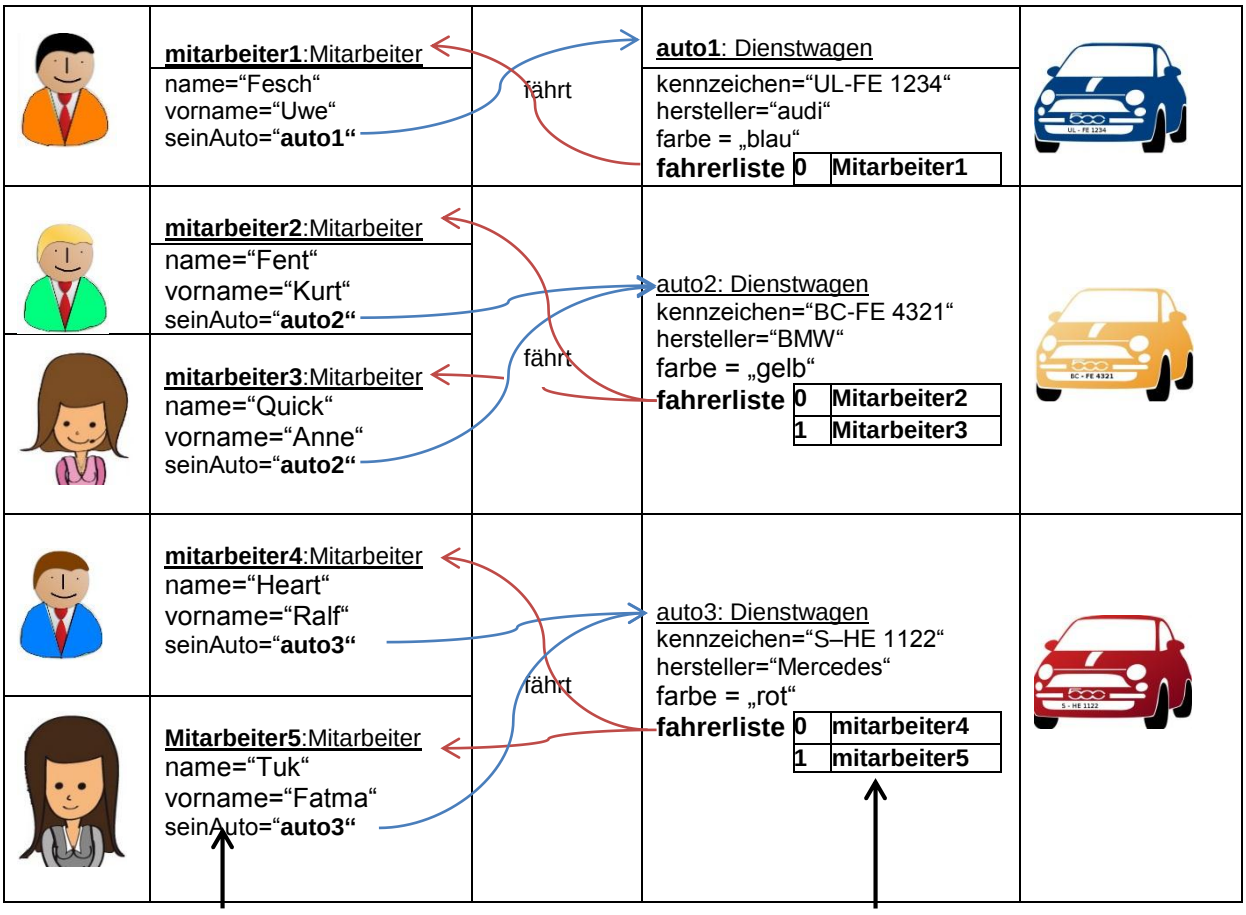
```
System.out.println("Mitarbeiter fährt mit Dienstwagen ");
System.out.println("-----------------------------------");
// mitarbeiter1 fährt mit auto1
System.out.print(mitarbeiter1.konsolenanzeige_Mitarbeiter()+" fährt mit ");
System.out.println(mitarbeiter1.getSeinAuto().konsolenzeige_auto());

// mitarbeiter2 fährt mit auto2
System.out.print(mitarbeiter2.konsolenanzeige_Mitarbeiter()+" fährt mit ");
System.out.println(mitarbeiter2.getSeinAuto().konsolenzeige_auto());

// mitarbeiter3 fährt mit auto3
System.out.print(mitarbeiter3.konsolenanzeige_Mitarbeiter()+" fährt mit ");
System.out.println(mitarbeiter3.getSeinAuto().konsolenzeige_auto());

System.out.println();  // Leerzeile

System.out.println("Dienstwagen wird gefahren von Mitarbeiter");
System.out.println("-----------------------------------------");
//auto1 wird gefahren von mitarbeiter1
System.out.print(auto1.konsolenzeige_auto()+" wird gefahren von ");
System.out.println(auto1.getSeinFahrer().konsolenanzeige_Mitarbeiter());

//auto2 wird gefahren von mitarbeiter2
System.out.print(auto2.konsolenzeige_auto()+" wird gefahren von ");
System.out.println(auto2.getSeinFahrer().konsolenanzeige_Mitarbeiter());
//auto3 wird gefahren von mitarbeiter3

System.out.print(auto3.konsolenzeige_auto()+" wird gefahren von ");
System.out.println(auto3.getSeinFahrer().konsolenanzeige_Mitarbeiter());
```

3 Die 1 – N - Assoziation

3.1 Sachverhalt - Beschreibung:

Die Firma GeLA GmbH stellt ihren Außendienstmitarbeitern Geschäftswagen zur Verfügung. Bisher konnte jeder Mitarbeiter immer nur ein und dasselbe Auto benutzen. Da sich in den Absatzgebieten Oberschwaben/Bodensee und Stuttgart die Nachfrage stark erhöht hat, werden für diese Absatzgebiete jeweils eine Mitarbeiterin (Anne Quick für Oberschwaben/ Bodensee und Fatma Tuk für Stuttgart) eingestellt. Da immer nur ein Mitarbeiter/eine Mitarbeiterin in diesen beiden Absatzgebieten Außendienst macht und der/die andere dann im Innendienst arbeitet, teilen sich Kurt Fend und Anne Quick sowie Ralf Heart und Fatma Tuk jeweils einen Dienstwagen.

Die untenstehende Darstellung zeigt, wie die Beziehung zwischen den Objekten jetzt aussieht.

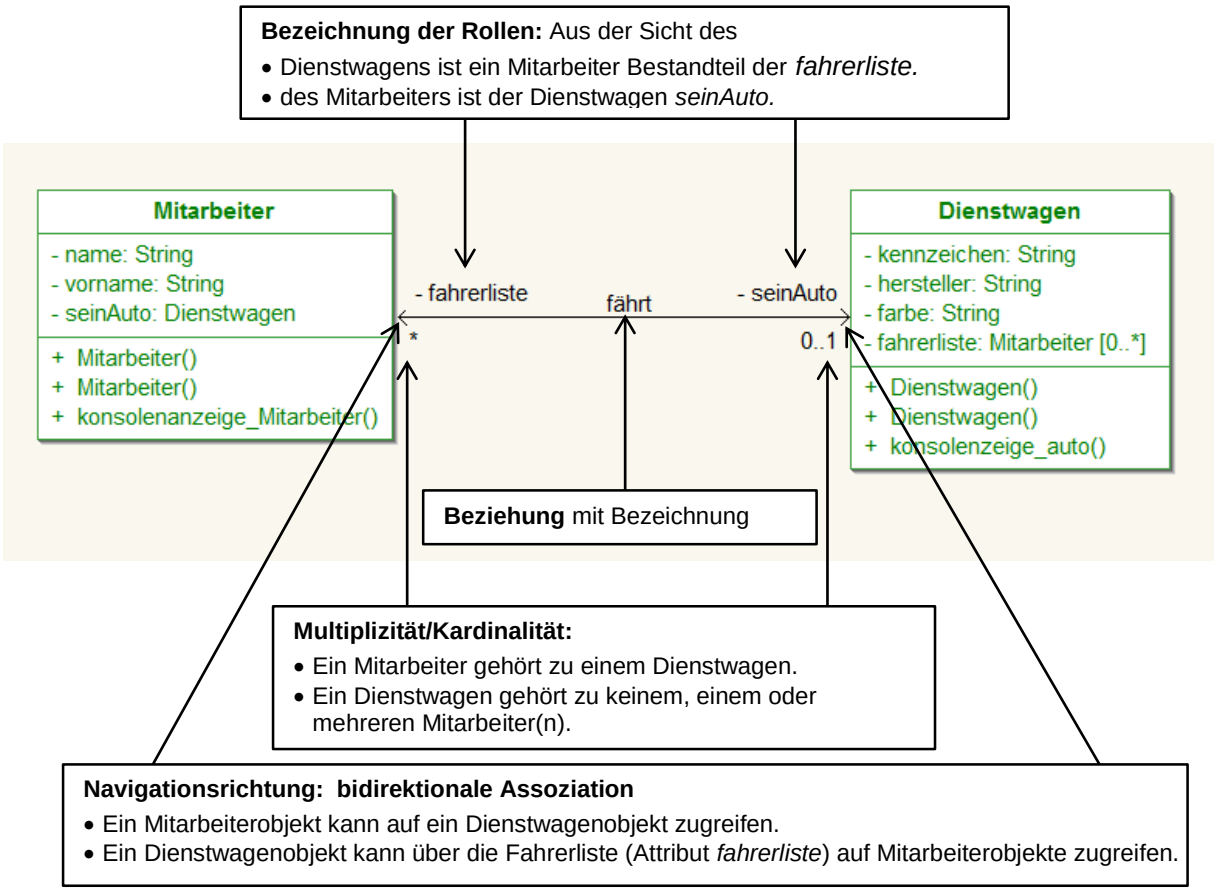


Soll von einem Mitarbeiterobjekt aus auf ein Autoobjekt zugegriffen werden, geschieht dies über das **Referenzattribut** *seinAuto*, dessen Attributwert das referenzierte Autoobjekt ist.

Sollen von einem Autoobjekt aus auf ein oder mehrere Mitarbeiterobjekte zugegriffen werden, geschieht dies über die Referenzliste *fahrerliste*, in der als Werte die referenzierten Mitarbeiterobjekte enthalten sind.

Die Attributtypen der Referenzattribute sind vom Typ der Klasse des verbundenen Objekts. Die Referenzliste *fahrerliste* ist ein Objekt der Containerklasse *ArrayList.*

In der nachfolgenden Abbildung werden die Spezifikationen beim Modellieren einer 1 – N - Assoziationen dargestellt und erläutert.



3.2 Implmentieren einer 1 – N – Assoziation

Die im Sachverhalts auf Seite 17 dargestellten Assoziationen zeigen, wie die Dienstwagen den Mitarbeitern und die Mitarbeiter den Dienstwagen zugeordnet sind.

Es soll die Zuordnung in beiden Richtungen möglich sein, das bedeutet, dass sowohl die Dienstwagen den Mitarbeitern, als auch die Mitarbeiter den Dienstwagen zugeordnet werden müssen (= **bidirektionale Assoziation**).

Die Assoziation *fährt* wird dadurch verwirklicht, dass Mitarbeiterobjekte ein Attribut *seinAuto* und damit einen Verweis zu den *Dienstwagen* erhalten. Dienstwagenobjekte erhalten als Attribut die Liste *fahrerliste*, in der alle Mitarbeiter aufgeführt sind, die mit dem entsprechenden Dienstwagen in Beziehung stehen (**1 – N - Assoziation**).

Folgende Aufgabenstellungen sind zu bearbeiten:

- Das Projekt projektDienstwagen2 mit dem Paket paketDienstwagen2 einrichten. Mithilfe des UML-Editors eUML2 die Klassen *Mitarbeiter* und *Dienstwagen* wie auf Seite 4 dargestellt, modellieren.
  Stattdessen kann auch aus dem Ordner *Objektorientierte Systementwicklung: Assoziationen (Projekte - Quellcodes) das Projekt projektDienstwagen* importiert werden und dann in projektDienstwagen3 umbenannt und weiter bearbeitet werden.
  (http://www.schule-bw.de/unterricht/faecher/informatik/material/softwareentwicklung/ooa-ood-oop/)

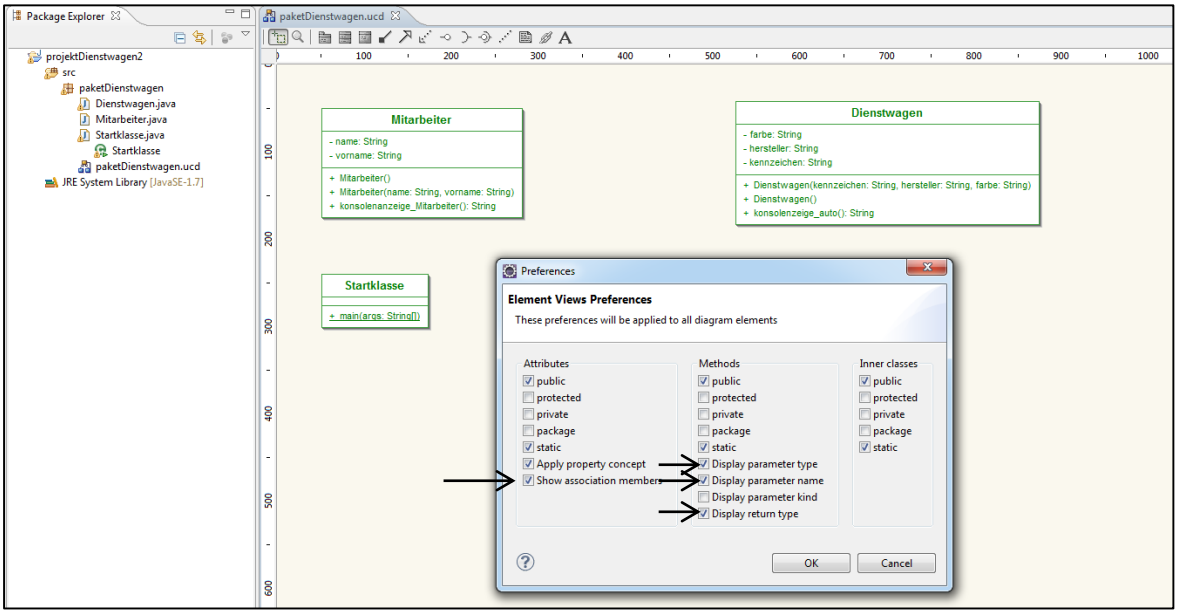
- Assoziationen modellieren. Dabei werden die Referenzattribute angelegt und die Zugriffmethoden generiert beziehungsweise programmiert.
  - Assoziation der Klasse *Mitarbeiter* zur Klasse *Dienstwagen* mit der *Multiplizität 1* modellieren. (Ein Mitarbeiter fährt mit einem Dienstwagen.)
  - Assoziation der Klasse *Dienstwagen* und *Mitarbeiter* mit der *Multiplizität N* modellieren. (Zu einem Dienstwagen gehören ein oder mehrere Fahrer.)

- In der Klasse *Startklasse* die Objekte mit den Objektverbindungen anlegen und im Konsolenfenster anzeigen.

Lösungen:

Das Projekt *projektDienstwagen* mit dem Paket *paketDienstwagen* importieren und in *projektDienstwagen3* umbenennen.
Zum Umbenennen des Projekts und des Pakets klickt man das Projekt beziehungsweise das Paket an, öffnet mit der rechten Maustaste das Kontextmenü zum Projekt/Paket, wählt den Menübefehl **`Refactor Rename`**… aus und vergibt die neuen Bezeichnungen.

Danach liegt das nachfolgend abgebildete Modell vor und die Assoziationen können modelliert werden.



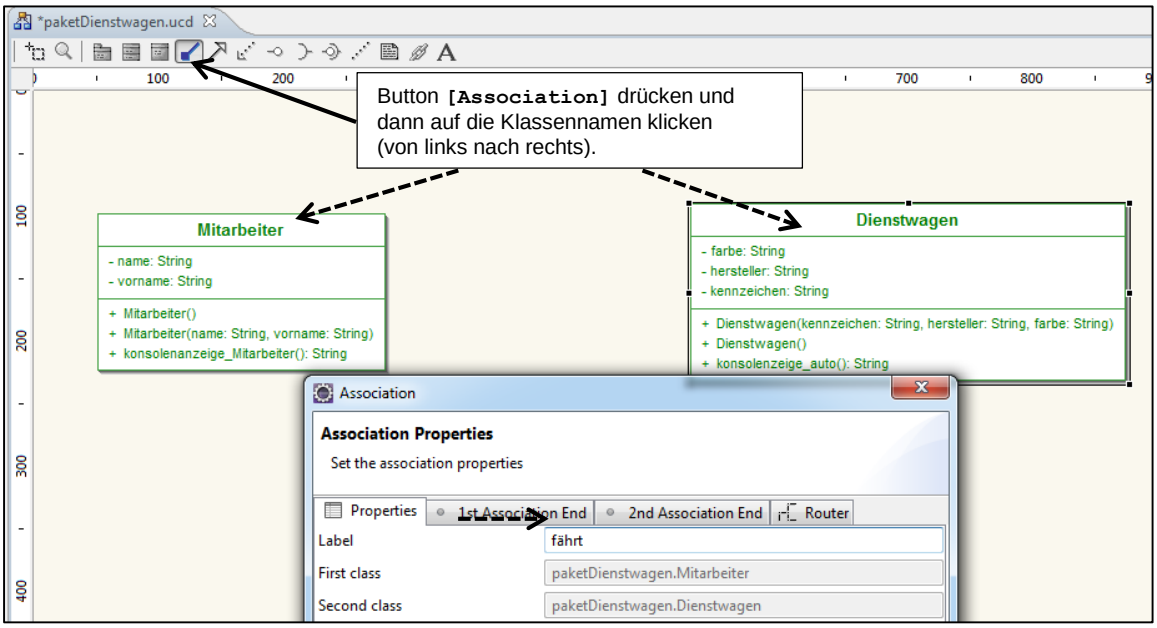
Hinweis:

Nach dem Öffnen des Kontextmenüs zur Zeichenfläche (Klick mit rechter Maustaste an einer beliebigen Stelle der Zeichenfläche) und der Auswahl des Menübefehls **`Preferences ► Element views`**…, können weitere Einstellungen für das Modell, wie das Sichtbarmachen der Referenzattribute und die Rückgabetypen der Methoden sowie der Namen und Typen der Parameter festlegen werden.

Assoziationen modellieren

➔ Assoziation herstellen und Name (=Label) vergeben.

Der Ablauf beim Modellieren von Assoziationen ist auf den Seiten 7 ff. beschrieben.



➔ **Das erste Assoziationsende**

Softwaretechnisch wird die N - Seite der Assoziation mithilfe eines Objekts einer Containerklasse (Sammlungsklasse) realisiert. Containerklassen werden im *Package java.util* bereitgestellt. Sie verfügen über die gängigen Operationen (einlesen, auslesen, ermitteln der Anzahl und entfernen der Elemente) zum Verwalten der Objekte (= Elemente der Sammlung).

Solche Containerklassen können sein:

- *Arrays*
- *ArrayList*
- *Vector*
- *LinkedList*

Soll eine Liste verwendet werden, stellt sich die Frage, welche der genannten Sammlungsklassen dafür am besten geeignet ist.

Während beim Einlesen von Elementen in Objekte der Klasse *Array* die Dimension des Bereichs definiert werden muss, passen sich die Objekte der Klassen *ArrayList*, *Vector* und *LinkedList* beim Einlesen von Elementen dynamisch an.

ArrayList liefert die besten Performanceergebnisse und wird daher in den folgenden Ausführungen verwendet.

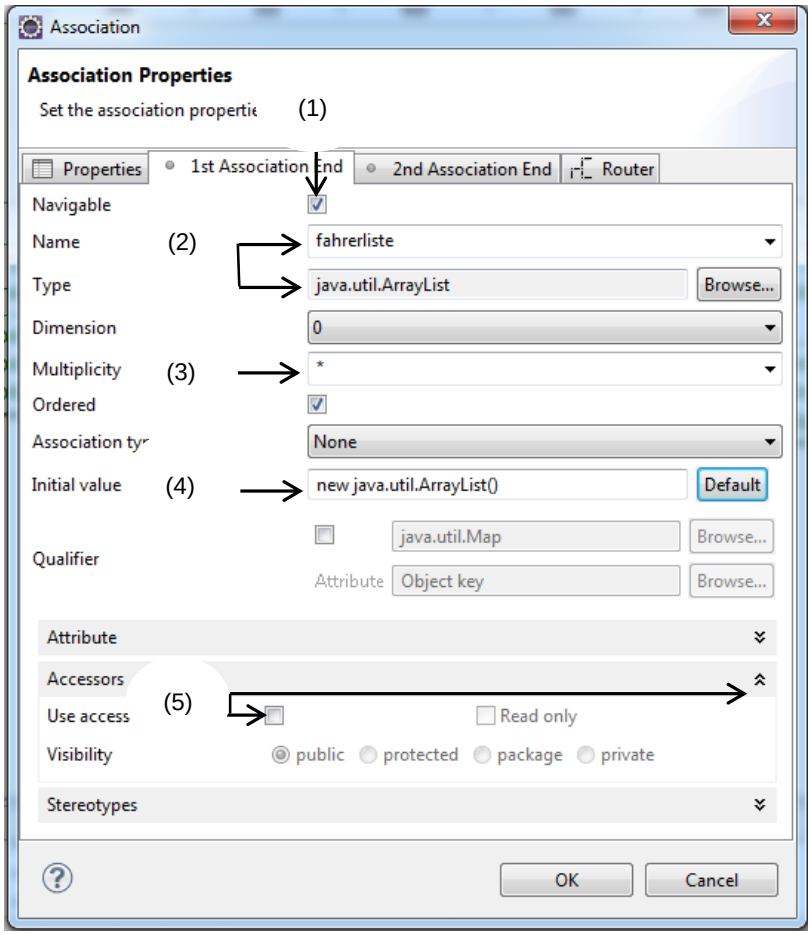
Für das Beispiel bedeutet dies, dass die Klasse *Dienstwagen* das Attribut *fahrerliste* vom Typ *ArrayList* erhält. Jedes Objekt der Klasse *Dienstwagen* erhält so beim Anlegen eine Fahrerliste (Attribut *fahrerliste*), in der auf die assoziierten Mitarbeiter verwiesen wird. Im Attribut *fahrerliste* sind die Mitarbeiterobjekte sequentiell angeordnet und nummeriert (indiziert). Die Indizierung beginnt mit Null. Wie in der Abbildung zum Sachverhalt auf Seite 17 dargestellt ist, sind dem Dienstwagenobjekt *auto3* zwei Mitarbeiterobjekte zugeordnet: *mitarbeiter4* und *mitarbeiter5*.

Das Attribut *fahrerliste* des Dienstwagenobjekts *auto3* hat daher folgenden Inhalt:

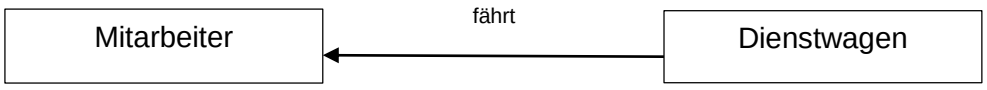
fahrerliste: ArrayList	
0	mitarbeiter4
1	mitarbeiter5

Index (Stelle) Attributwert

Mit einem Klick auf den Button `[1st Association End]`, kann das Assoziationsende der ersten Klasse beschrieben werden.

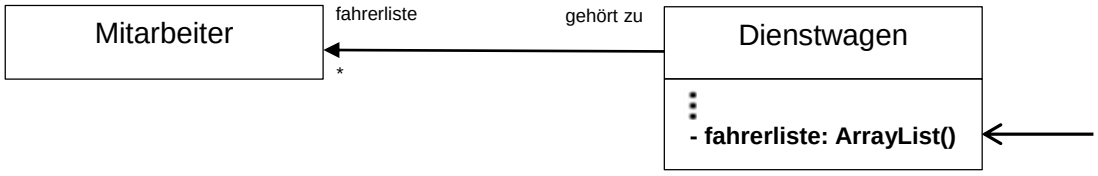


(1) Mit dem Kontrollfeld *Navigable* wird die Navigationsrichtung freigegeben. Im vorliegenden Beispiel endet die Assoziation in der Klasse *Mitarbeiter*, es kann also von Dienstwagen zu Mitarbeiter navigiert werden.

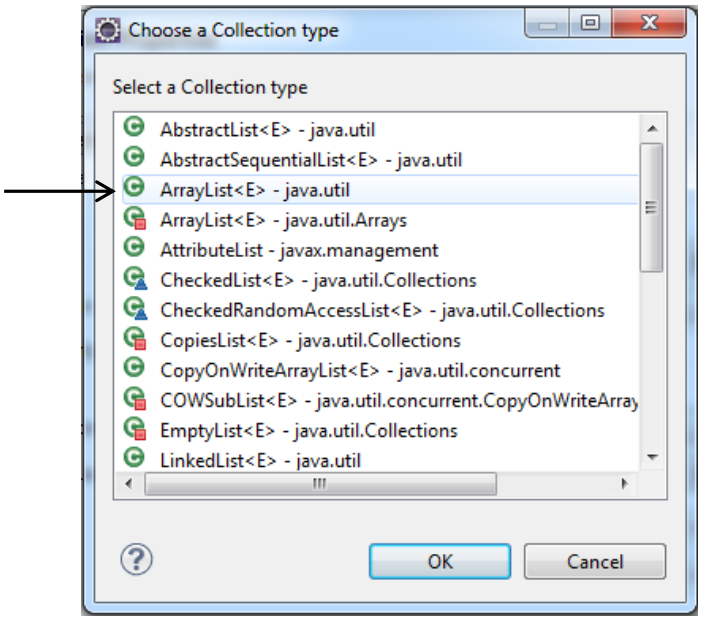


(2) Im Eingabefeld *Name* wird die Rolle, welche ein Objekt in der assoziierten Klasse spielt, festgelegt. Wie in der Abbildung des Sachverhalts (siehe Seite 17) zu erkennen ist, wird das Dienstwagenobjekt *auto3* von zwei Mitarbeitern (*mitarbeiter4* und *mitarbeiter5*) gefahren. In der Klasse *Dienstwagen* wird daher für die

Dienstwagenobjekte das Attribut *fahrerliste* vom Typ *ArrayList* eingerichtet. Darin werden die Mitarbeiterobjekte aufgenommen, die den Dienstwagen benutzen.



Da mehrere Mitarbeiter einen Dienstwagen benutzen können, wird das Attribut *fahrerliste* als Objekt einer Containerklasse deklariert. Bei der Auswahl von *Type* wird jetzt mit dem Button **`[Browse]`** der Containertyp *ArrayList* aus der Bibliothek *java.util* ausgewählt.



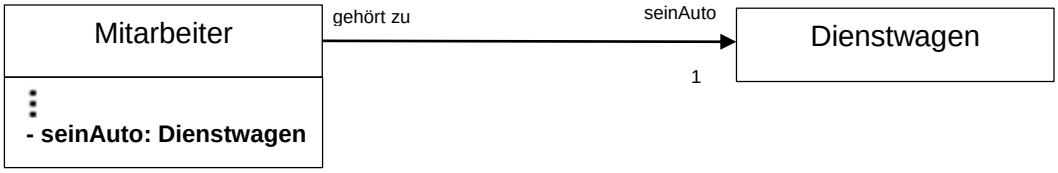
(3) Hier wird die Multiplizität ausgewählt. Zu einem Dienstwagen können kein, ein oder mehrere Fahrer gehören.

(4) Im Eingabefeld **`Initial value`** wird durch Drücken des Buttons **`[Default]`** die Klasse angezeigt, aus der das an der Assoziation beteiligte Objekt stammt. Im vorliegenden Beispiel ist dies die Klasse *ArrayList()* aus der Bibliothek *java.util*.

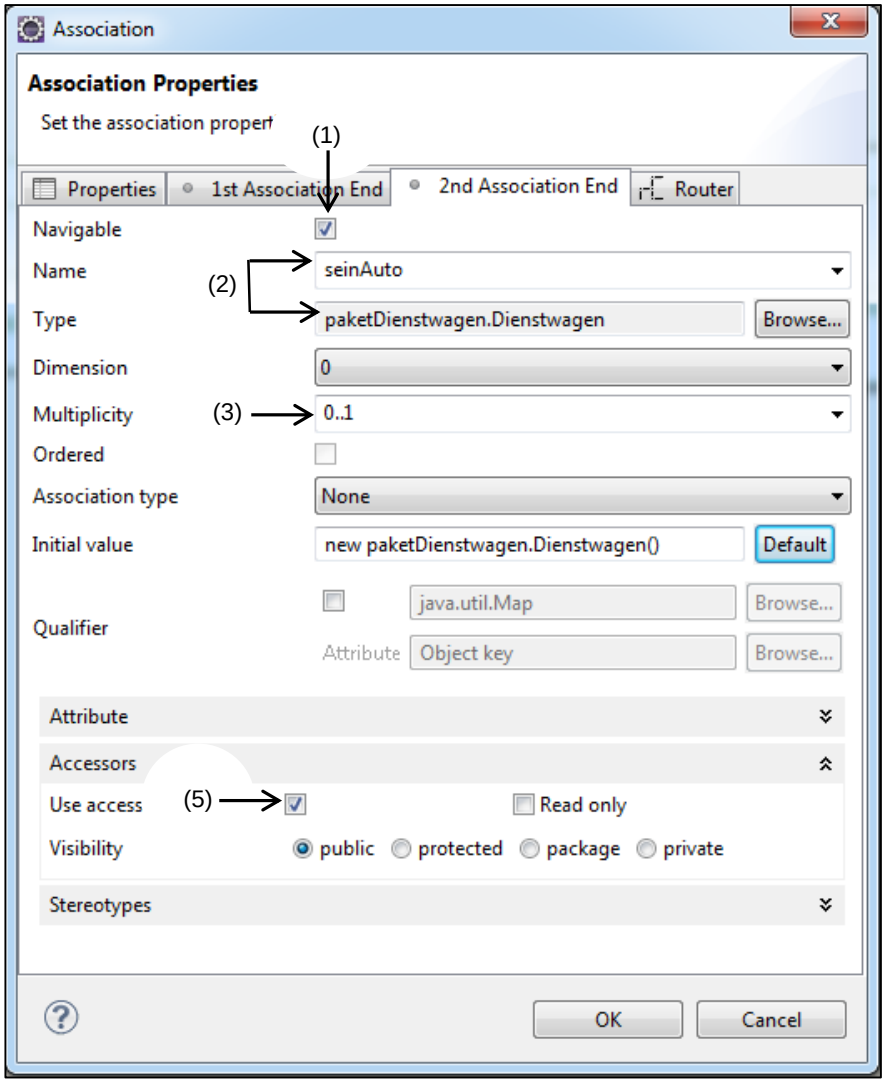
(5) Hier kann durch das Aktivieren beziehungsweise Deaktivieren des Kontrollfeldes **`Use accessors`** entschieden werden, ob zu dem erzeugten Referenzattribut *fahrerliste* Zugriffsmethoden generiert werden sollen oder nicht. Für die Objekte der Referenzliste werden anschließend eigene Zugriffsmethoden erstellt, weshalb an dieser Stelle die Option **`Use accessors`** **nicht gewählt (deaktiviert)** wird.

➔ Das zweite Assoziationsende (2n Association End)

Das 2. Assoziationsende wird nach einem Klick auf den Button **`[2nd Association End]`** beschrieben. Einem Mitarbeiter ist ein Dienstwagen zugeordnet. Dies wird in der Klasse *Mitarbeiter* durch das Attribut *seinAuto* vom Typ *Dienstwagen* realisiert.



Spezifizieren des Endes der zweiten Assoziation:



(1) Die Assoziation endet in der Klasse *Dienstwagen*, es kann also von Mitarbeiter zu Dienstwagen navigiert werden.

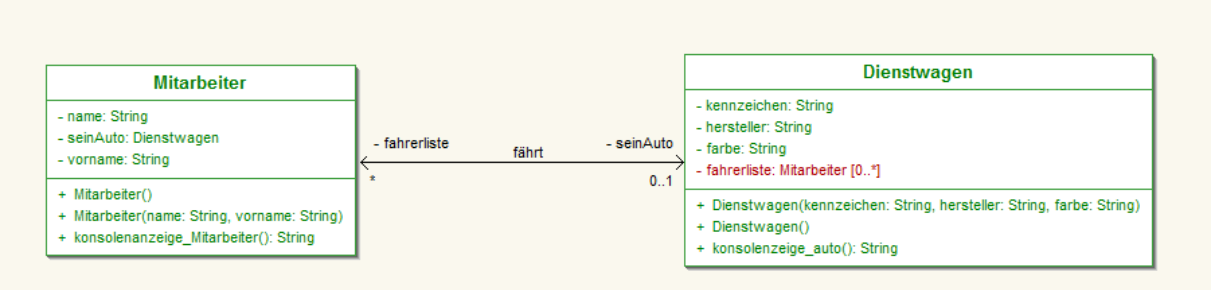
(2) Ein Dienstwagen ist für ein Objekt der Klasse *Mitarbeiter* sein Auto, weshalb als Rollenname *seinAuto* gewählt wird. Der Rollenname wird auch als Name für das Referenzattribut verwendet. Mit dem Referenzattribut *seinAuto* vom Typ der Klasse *Dienstwagen* wird die Assoziation realisiert.

(3) Hier wird die Multiplizität 0..1 ausgewählt. Zu einem Mitarbeiter kann kein oder ein Auto ghören.

(4) Im Eingabefeld **`Initial value`** wird durch Drücken des Buttons **`[Default]`** die Klasse angezeigt, aus der das an der Assoziation beteiligte Objekt stammt. Im vorliegenden Beispiel ist dies die Klasse *Dienstwagen*.

(5) Standardmäßig wird vorgeschlagen, die Zugriffsmethoden zu generieren.

Als Ergebnis erhält man folgendes UML-Diagramm:



Hinweis:

Wird das Klassendiagramm nicht korrekt dargestellt, weil die Referenzattribute nicht angezeigt werden, gibt es zwei Vorgehensweisen, um dies zu beheben:

- Anpassen der Eigenschaften der Zeichnungsfläche

  Kontextmenü zur Entwurfsfläche des *eUML2*-Editors öffnen (mit der rechten Maustaste auf eine beliebige freie Stelle klicken) und den Menübefehl **`Preferences`** ► **`Element views`**... auswählen und dann im Fenster **`Preferences`** die Option **`Show association members`** aktivieren.

- Attributanzeige für die Klassen einzeln überprüfen

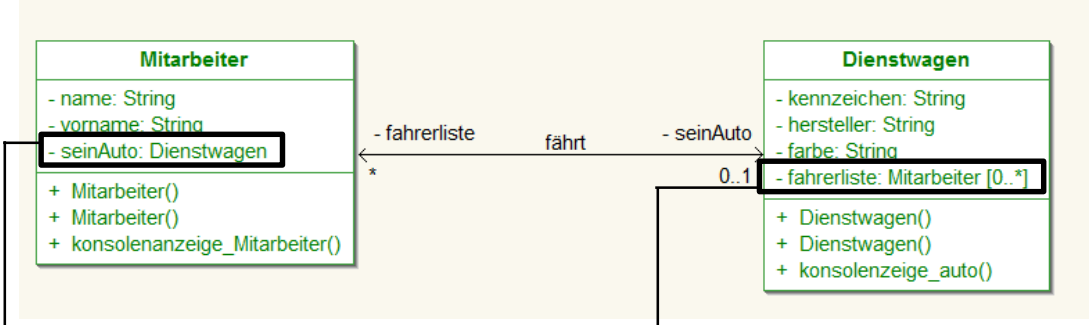
  Für die betreffende Klasse das Kontextmenü (Klick mit der rechten Maustaste auf das Klassendiagramm) öffnen, den Menübefehl **`View content selector`** auswählen, und im Fenster *View selector...* die anzuzeigenden Attribute aktivieren.

  Die beiden Vorgehensweisen sind bereits auf Seite 12 beschrieben.

Javacode der modellierten Assoziationen

Durch das Modellieren der Assoziationen im UML-Klassendiagramm wurde für die Referenzattribute Javacode generiert.

UML-Klassendiagramm



Java-Quellcode

```
public class Mitarbeiter {

    // Referenzattribut deklarieren
    private Dienstwagen seinAuto;

    //get-Methode (Attributwerte auslesen)

    public Dienstwagen getSeinAuto()
    {
         return seinAuto;
    }

    // set-Methode (Attributwerte setzen)
    public void setSeinAuto(Dienstwagen
                                seinAuto)
    {
         this.seinAuto = seinAuto;
    }
}
```

```
import java.util.ArrayList;


public class Dienstwagen {
    ⋮
    // Referenzliste erzeugen
    private ArrayList fahrerliste;
    ⋮
}
```

Die Assoziation mit der Multiplizität 1 wird über das Referenzattribut *seinAuto* realisiert. Mit den Zugriffsmethoden können Attributwerte vom Typ *Dienstwagen* ein- und ausgelesen werden.

Für die Assoziation mit der Multiplizität N wird die Referenzliste *fahrerliste* erzeugt. Sie ist ein Objekt der Containerklasse *ArrayList* aus der Bibliothek *java.util*.

Während der generierte Quellcode der Assoziation mit der Multiplizität 1 vollständig ist, muss er für die erzeugte *fahrerliste* (Assoziation mit der Multiplizität N) noch nachbearbeitet und ergänzt werden.

- Typisierung und Initialisierung des Attributs *fahrerliste*

  Damit in das Attribut *fahrerliste* nur Mitarbeiterobjekte aufgenommen werden können, wird die Quellcodezeile um den Typ der Objekte, die aufgenommen werden können, erweitert. Der aufzunehmende Typ wird in spitzen Klammern <…> angegeben.

```
generiert:  private ArrayList fahrerliste;
erweitert:  private ArrayList<Mitarbeiter> fahrerliste= new ArrayList<Mitarbeiter>();
```

- Zugriffsmethoden für das Attribut *fahrerliste*

  Für die wichtigsten Operationen zur Verwaltung von Mitarbeiterobjekten im Attribut *fahrerliste* eines Dienstwagenobjekts sind u. a. folgende Methoden bereitzustellen:

  - Aufnehmen eines Mitarbeiters,
  - Auslesen eines Mitarbeiters,
  - Ermitteln der Anzahl der Mitarbeiter in der Liste.

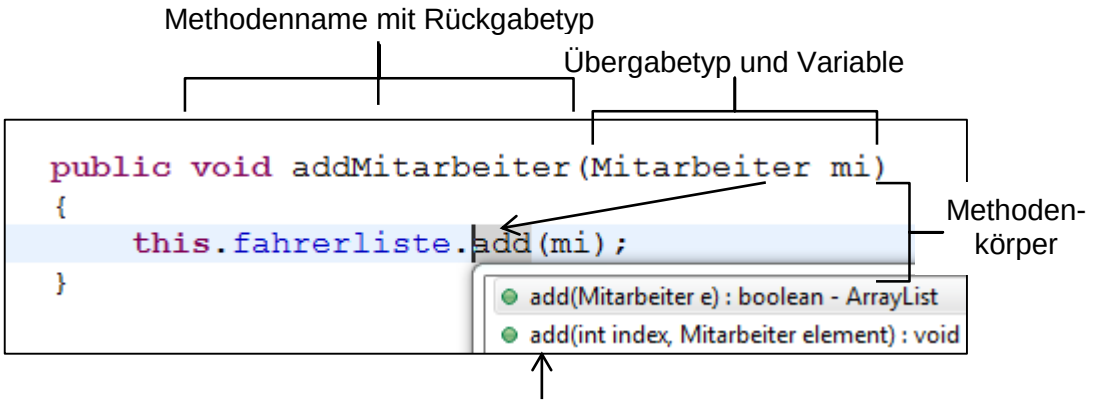
  Die Methoden können im
  - Java-Quellcode vollständig codiert werden.
  - eUML2-Editor softwareunterstützt erstellt und ausprogrammiert werden.
  - eUML2-Editor softwareunterstützt erstellt und im Java-Quellcode ausprogrammiert.

  In den folgenden Ausführungen wird jede der Vorgehensweisen vorgestellt.

➔ Aufnahme eines Mitarbeiters in die Fahrerliste des Dienstwagenobjekts

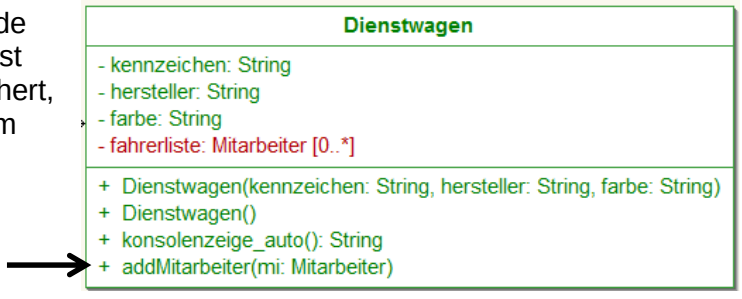
Mit der Methode **`addMitarbeiter(Mitarbeiter mi)`** soll ein Mitarbeiterobjekt in das Attribut *fahrerliste* des Dienstwagenobjekts aufgenommen werden.

Quellcode der Methode mit Erklärung



Die Methode **`public void addMitarbeiter(Mitarbeiter mi)`** hat den Rückgabetyp *void,* weil sie kein Ergebnis zurückgibt. Sie benötigt bei der Ausführung einen Übergabewert („Input“). Dafür wird die Variable *mi* vom Typ *Mitarbeiter* deklariert. Im Methodenkörper wird mit der Methode *add* der Wert der Variablen *mi* im Attribut *fahrerliste* angefügt.

Die Methode kann so im Quellcode für die Klasse *Dienstwagen* erfasst werden. Wird die Klasse gespeichert, erscheint sie im Klassendiagramm der Klasse *Dienstwagen*.



Die Methode **`public void addMitarbeiter(Mitarbeiter mi)`** kann auch mithilfe des *eUML2*-Editors generiert werden.

Vorgehensweise:

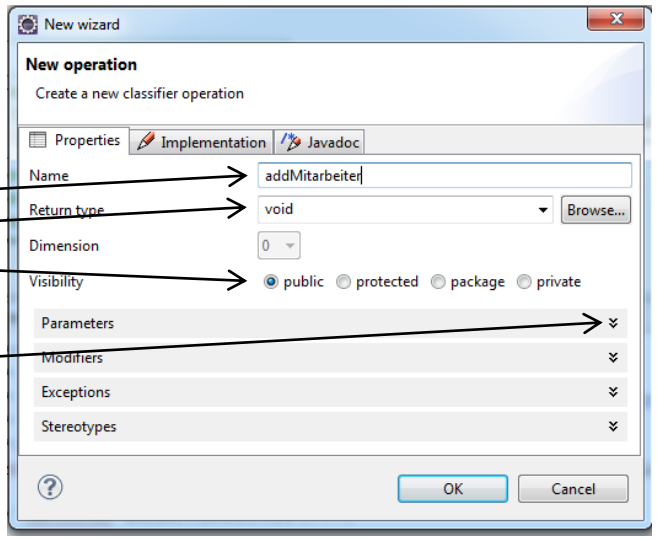
(1) Klasse *Dienstwagen* anklicken und über das Kontextmenü (rechte Maustaste) den Menübefehl **`New Method`** öffnen.

- Name der Methode,
- Rückgabetyp und
- Sichtbarkeit

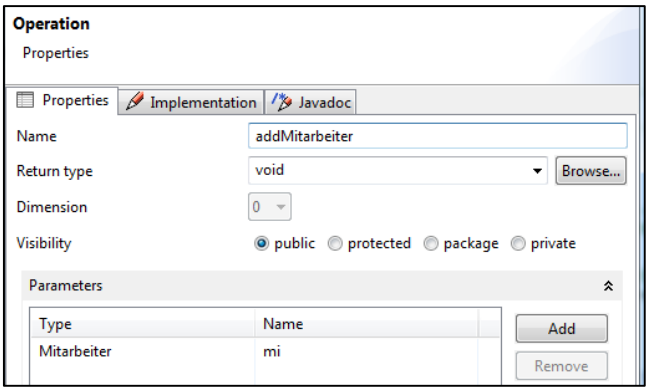
festlegen.

Anschließend das Feld Parameters erweitern. Es öffnet sich das Fenster **`New operation.`**

Screenshot



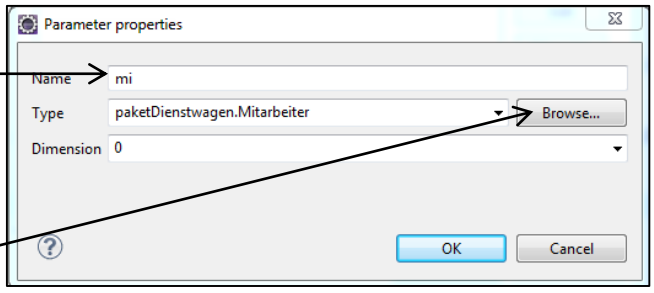
(2) Im Fenster **`Operation Properties`** wird dann mit dem Button **`[Add]`** das Fenster **`Parameter properties`** geöffnet.

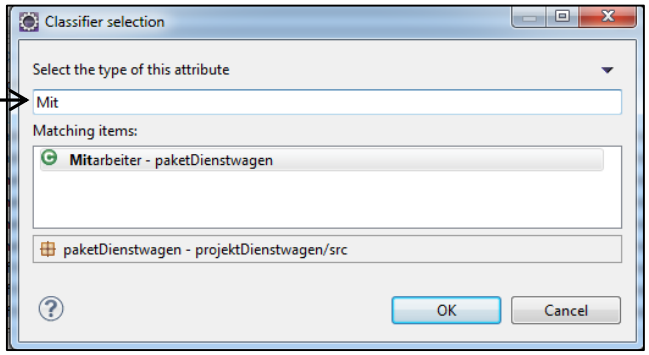


(3) Hier wird der Name des Parameters (hier *mi*) sowie sein Typ eingetragen.
Der Übergabeparameter *mi* muss ein Mitarbeiterobjekt in die Liste übergeben, er muss also vom Typ *Mitarbeiter* sein.

Mit dem Button **`[Browse]`** kann im Fenster **`Classifier selection`** der Datentyp aus einer Liste ausgewählt werden, nachdem seine Anfangszeichen eingegeben wurden.

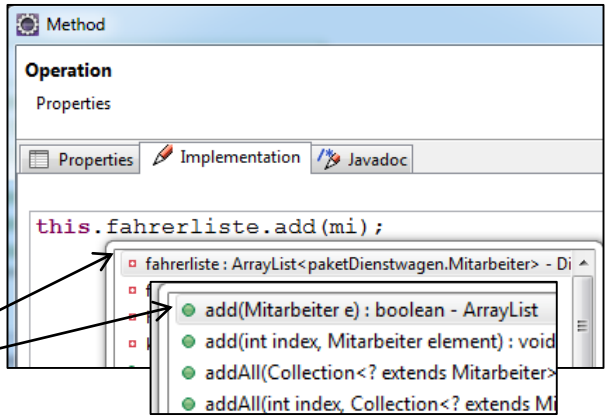
Werden die Fenster mit dem **`[OK]`**–Button geschlossen, ist der Methodenrumpf angelegt.





(4) Im Register **`Implementation`** wird nun der Methodenkörper „ausprogrammiert“, das heißt für die Methode **`addMitarbeiter(…)`**, dass jetzt der Übergabewert in das Attribut *fahrerliste* angefügt wird.

Sowohl das Attribut *fahrerliste* als auch die Methode *add(mi)* können mithilfe der Code-Ergänzungsfunktion erstellt werden: **`([Strg]+[Leertaste]`**-Taste).



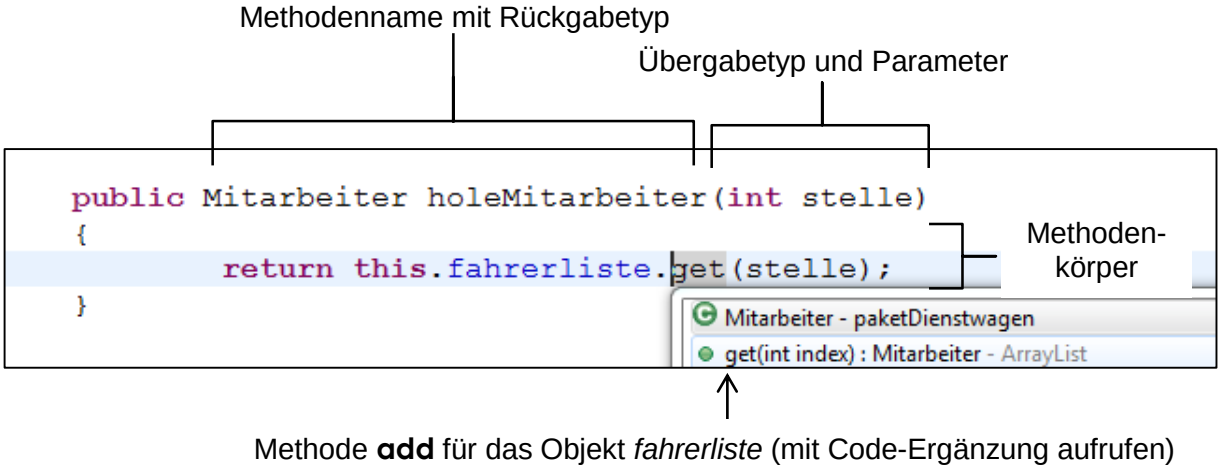
Als Ergebnis erhält man die fertige Methode;

```
public void addMitarbeiter(Mitarbeiter mi){
    this.fahrerliste.add(mi);
}
```

➔ Auslesen eines Mitarbeiters aus der fahrerliste des Dienstwagenobjekts

Mit der Methode **`public Mitarbeiter holeMitarbeiter(int Stelle)`** soll ein Mitarbeiterobjekt aus einer bestimmten Stelle des Attributs *fahrerliste* des Dienstwagenobjekts ausgelesen und zurückgegeben werden.

Quellcode der Methode mit Erklärung

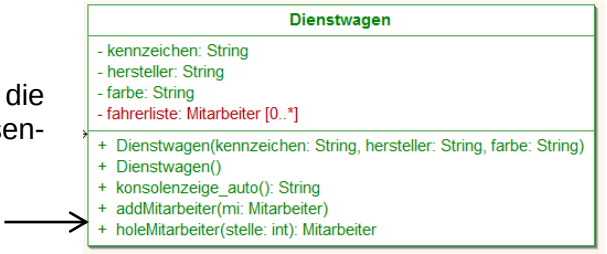


Mit der Methode **`public Mitarbieter holeMitarbeiter(int Stelle)`** wird ein Mitarbeiterobjekt aus dem Attribut *fahrerliste* ausgelesen. Sie hat deshalb den Rückgabetyp (*Mitarbeiter*).

Wie auf Seite 21 dargestellt, sind die Objekte in Attributen vom Typ *ArrayList* sequentiell angeordnet und indiziert. Die Indizierung beginnt mit 0. Soll ein Element aus der Liste ausgelesen werden, muss der Methode die Nummer der Stelle, an der das Element in der Liste steht, als Parameter (hier: *int stelle*) mitgegeben werden.

Im Methodenkörper wird mit der Methode *get(stelle)* im Attribut *fahrerliste* der Attributwert an dieser Stelle zurückgegeben.

Die Methode kann so im Quellcode für die Klasse *Dienstwagen* erfasst werden. Wird die Klasse gespeichert, erscheint sie im Klassendiagramm der Klasse *Dienstwagen*.



Eine häufige Vorgehensweise beim Erstellen von Methoden ist die Erzeugung des Methodenkopfes mithilfe des Wizards von *eUML2* (Sichtbarkeit, Rückgabetyp, Methodenname und Übergabeparameter) und die anschließende Ergänzung des Methodenkörpers im Quellcode.

Vorgehensweise:

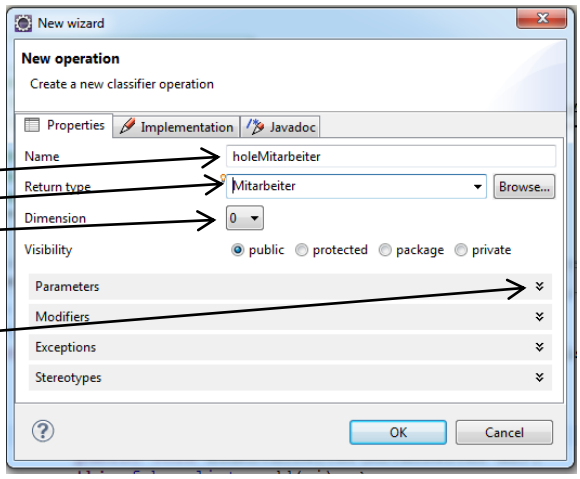
(1) Klasse *Dienstwagen* anklicken und über das Kontextmenü (rechte Maustaste) den Menübefehl **`New Method`** öffnen.

- Name der Methode,
- Rückgabetyp und
- Sichtbarkeit

festlegen.

Anschließend das Feld Parameters erweitern. Es öffnet sich das Fenster **`New operation.`**

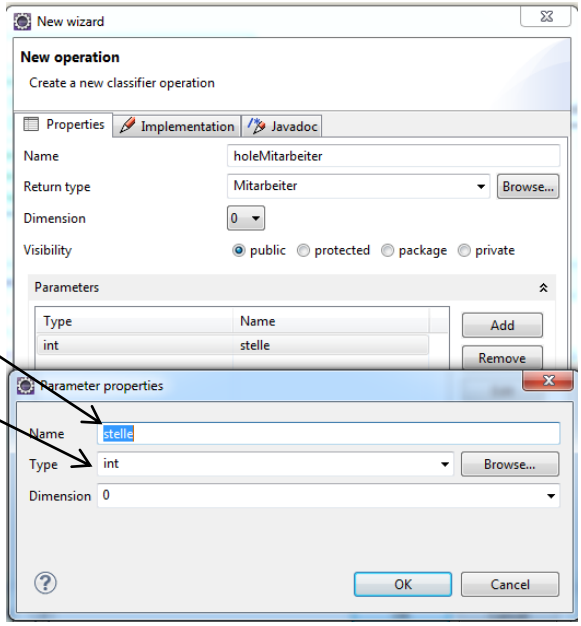
Screenshot



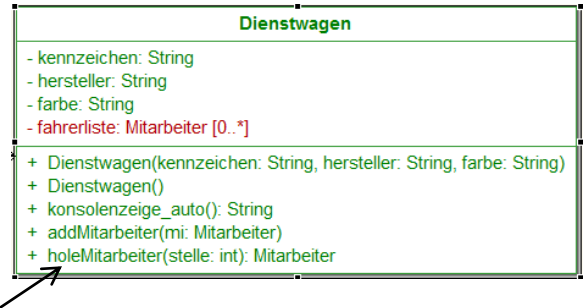
(2) Im Fenster **`New operation`** wird anschließend mit dem Button **`[Add]`** das Fenster **`Parameter properties`** geöffnet.

Hier wird der Name des Parameters (hier *stelle*) eingetragen und sein Datentyp mithilfe des **`[Browse]`**-Buttons ausgewählt.

Werden die Fenster mit dem **`[OK]`**-Button geschlossen, ist der Methodenrumpf angelegt.



Falls die neu erstellte Methode nicht sofort im Klassendiagramm erscheint, kann sie über den Menübefehl **`View content selector`** sichtbar gemacht werden. Dazu muss die Klasse angeklickt und das Kontextmenü geöffnet werden.



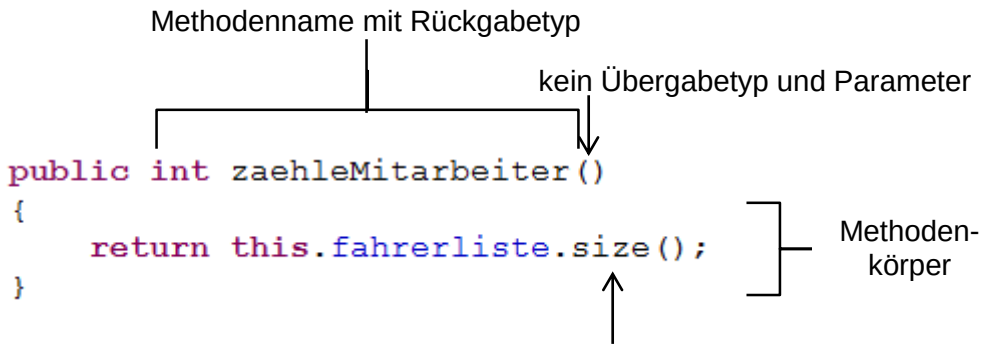
Mit einem Doppelklick auf den Methodennamen wird der Quellcode editiert und kann jetzt vervollständigt werden.

```
public Mitarbeiter holeMitarbeiter(int stelle)
{
     return this.fahrerliste.get(stelle);
}
```

➔ **Ermitteln der Anzahl der Elemente in der Fahrerliste des Dienstwagenobjekts**

Mit der Methode **`public zaehleMitarbeiter()`** soll die Anzahl der Elemente im Attribut *fahrerliste* des Dienstwagenobjekts ermittelt und zurück gegeben werden.

Quellcode der Methode mit Erklärung



Methode **size()** für das Objekt *fahrerliste* mit Code-Ergänzung aufrufen.
(Methode der Klasse *ArrayList*)

Mit der Methode **`public zaehleMitarbeiter()`** wird die Anzahl der Mitarbeiterobjekt im Attribut *fahrerliste* ermittelt und ausgelesen, sie hat deshalb den Rückgabetyp *int*.

Die Methode kann so im Quellcode für die Klasse *Dienstwagen* erfasst werden.

3.3 Die 1 – N - Assoziation in der Klasse *Startklasse* mit der *main*-Methode nutzen.

Die 1 – N - Assoziation zwischen den Klassen *Mitarbeiter* und *Dienstwagen* soll nun in der Klasse *Startklasse* genutzt werden. Objekte werden erzeugt, verknüpft (assoziiert) und dann im Konsolenfenster ausgegeben.

Folgende Aufgaben sollen ausgeführt werden:

- Die im Sachverhalt auf Seite 17 dargestellten Objekte der Klassen *Mitarbeiter* und *Dienstwagen* sind zu erzeugen.

- Die auf Seite 17 beschriebenen Objektverbindungen (in beide Richtungen) sind vorzunehmen.

- Ausgabe der Objektverbindungen im Konsolenfenster nach folgendem Beispiel:

```
Mitarbeiter     fahrt mit       Dienstwagen
-------------------------------------------------
Uwe Fesch       fährt mit       Audi UL-FE 1234
Kurt Fent       fährt mit       BMW BC-FE 4321
Anne Quick      fährt mit       BMW BC-FE 4321
Ralf Heart      fährt mit       Mercedes S - HE 1122
Fatma Tuk       fährt mit       Mercedes S - HE 1122

Fahrer des Dienstagen Audi, UL-FE 1234
----------------------
Uwe Fesch

Fahrer des Dienstagen BMW, BC-FE 4321
----------------------
Kurt Fent
Anne Quick

Fahrer des Dienstagen Mercedes, S - HE 1122
----------------------
Ralf Heart
Fatma Tuk
```

Im Folgenden wird der kommentierte Quellcode der Klasse *Startklasse* aufgeführt.

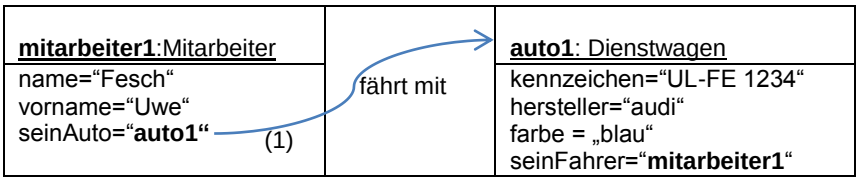
```
public static void main(String[] args)
{
// Mitarbeiterobjekte mit dem angepassten  erzeugen
Mitarbeiter mitarbeiter1 = new Mitarbeiter("Fesch","Uwe");
Mitarbeiter mitarbeiter2 = new Mitarbeiter("Fent","Kurt");
Mitarbeiter mitarbeiter3 = new Mitarbeiter("Quick","Anne");
Mitarbeiter mitarbeiter4 = new Mitarbeiter("Heart","Ralf");
Mitarbeiter mitarbeiter5 = new Mitarbeiter("Tuk","Fatma");
// Dienstwagenobjekte mit dem angepassten Konstruktor erzeugen
Dienstwagen auto1 = new Dienstwagen("UL-FE 1234","Audi","blau");
Dienstwagen auto2 = new Dienstwagen("BC-FE 4321","BMW","gelb");
Dienstwagen auto3 = new Dienstwagen("S-HE 1122","Mercedes","rot");
// Assoziation 1: Mitarbeiter fährt mit Auto
mitarbeiter1.setSeinAuto(auto1);
mitarbeiter2.setSeinAuto(auto2);
mitarbeiter3.setSeinAuto(auto2);
mitarbeiter4.setSeinAuto(auto3);
mitarbeiter5.setSeinAuto(auto3);
// Assoziation 2: Fahrerliste eines Autos
auto1.addMitarbeiter(mitarbeiter1);  //auto1 wird gefahren von mitarbeiter1
auto2.addMitarbeiter(mitarbeiter2);  //auto2 wird gefahren von mitarbeiter2 u. mitarbeiter3
auto2.addMitarbeiter(mitarbeiter3);
auto3.addMitarbeiter(mitarbeiter4);  //auto3 wird gefahren von mitarbeiter4 u. mitarbeiter5
auto3.addMitarbeiter(mitarbeiter5);
//Konsolenausgabe
String tab = "\t";       // Tabulatorsprung in der variablen tab speichern
System.out.println("Mitarbeiter"+tab + "fahrt mit"+ tab + "Dienstwagen");
System.out.println("-----------------------------------------------------");
// mitarbeiter1 fährt mit auto1
System.out.print(mitarbeiter1.getVorname()+" "+mitarbeiter1.getName()+tab+"fährt mit"+tab);
System.out.println(mitarbeiter1.getSeinAuto().getHersteller()+"
                                    "+mitarbeiter1.getSeinAuto().getKennzeichen());
// mitarbeiter2 fährt mit auto2
System.out.print(mitarbeiter2.getVorname()+" "+mitarbeiter2.getName()+tab+"fährt mit"+tab);
System.out.println(mitarbeiter2.getSeinAuto().getHersteller()+"
                                    "+mitarbeiter2.getSeinAuto().getKennzeichen());
⋮

// Fahrerliste des Dienstwagen auto1
System.out.println("Fahrer des Dienstagen "+auto1.getHersteller()+",
                                                    "+auto1.getKennzeichen());
System.out.println("----------------------");
for (int i = 0; i < auto1.zaehleMitarbeiter(); i++)
{
    System.out.println(auto1.holeMitarbeiter(i).getVorname()+"
                                                +auto1.holeMitarbeiter(i).getName());
}
System.out.println();
// Fahrerliste des Dienstwagen auto2
⋮

// Fahrerliste des Dienstwagen auto3
System.out.println("Fahrer des Dienstagen "+auto3.getHersteller()+",
"+auto3.getKennzeichen());
System.out.println("----------------------");
for (int i = 0; i <auto3.zaehleMitarbeiter(); i++)
{
       System.out.println(auto3.holeMitarbeiter(i).getVorname()+"
"+auto3.holeMitarbeiter(i).getName());
}
```

(1)

(2)

Erläuterung zu den Objektverbindungen und dem Zugriff auf die Attribute der assoziierten Objekte:

(1) Das Objekt *mitarbeiter1* ist mit dem Objekt *auto1* über das Attribut *seinAuto* assoziiert.



Das Referenzattribut *seinAuto* erhält als Wert das assoziierte Objekt *auto1.*

Javacode:

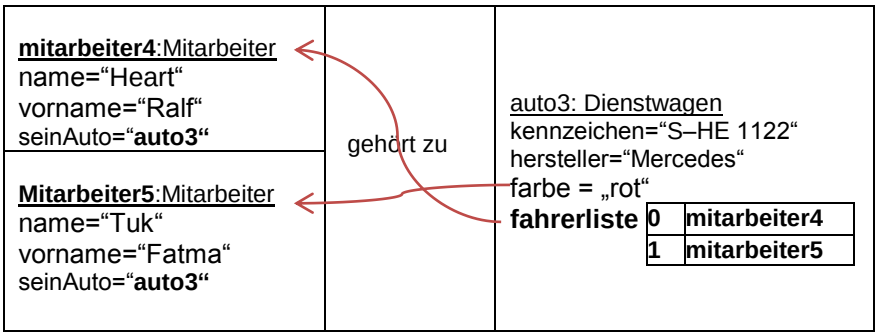
```
mitarbeiter1.setSeinAuto(auto1);
```

Auf die Attributwerte des assoziierten Objekts *auto1* wird mit Methoden des Referenzattributs zugegriffen.

```
// mitarbeiter1 fährt mit auto1
System.out.print(mitarbeiter1.getVorname()+" "+mitarbeiter1.getName()+" fährt mit ");
System.out.print(mitarbeiter1.getSeinAuto().getHersteller()+", ");
System.out.print(mitarbeiter1.getSeinAuto().getKennzeichen());
System.out.println(mitarbeiter1.getSeinAuto().getFarbe());
```

Über die Methode *getSeinAuto()* können die Methoden des assoziierten Objekts *auto1* verwendet werden.

(2) das Objekt *auto3* ist mit den Objekten *mitarbeiter4* und *mitarbeiter5* über das Attribut *fahrerliste* assoziiert.



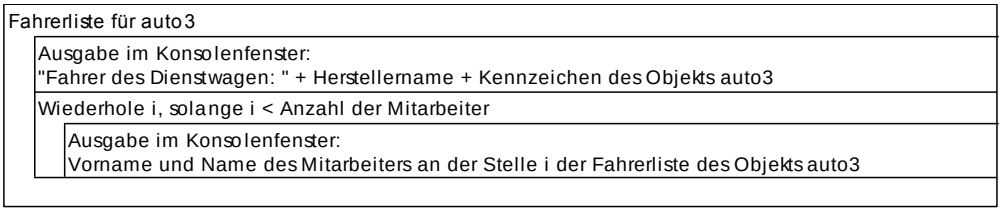
Im Attribut *fahrerliste* sind als Wert die assoziierten Objekte *mitarbeiter4* und *mitarbeiter5* enthalten. Sie sind hintereinander angeordnet und nummeriert (indiziert).

Javacode:

```
// Auto 3 wird gefahren von Mitarbeiter 4 und Mitarbeiter 5
auto3.addMitarbeiter(mitarbeiter4);
auto3.addMitarbeiter(mitarbeiter5);
```

Die Elemente des Attributs *fahrerliste* eines Objekts (hier des Objekts *auto3*) werden im Konsolenfenster angezeigt. Da eine Fahrerliste ein oder mehrere Elemente (Mitarbeiter) enthält, wird für das Auslesen der Elemente der Liste eine Wiederholungsstruktur benötigt.

Struktogramm:

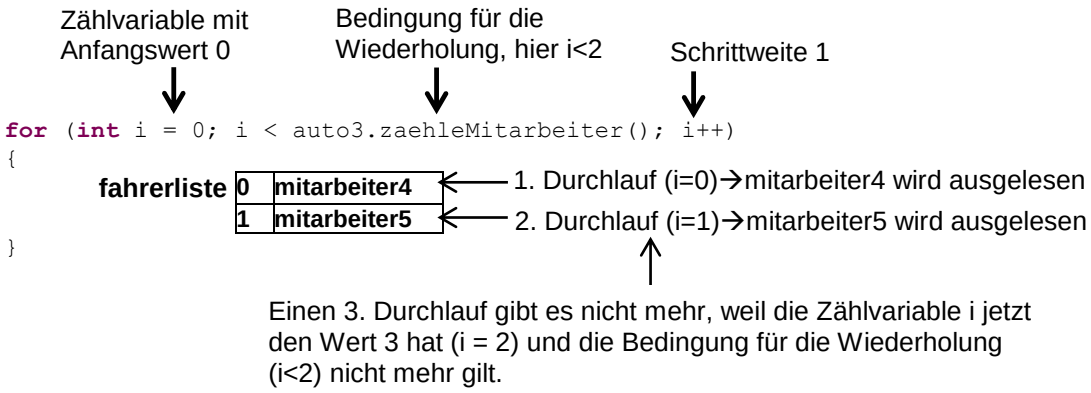


Javacode:

```
// Fahrerliste des Dienstwagen auto3
System.out.println("Fahrer des Dienstagen "+auto3.getHersteller()+",
                                              "+auto3.getKennzeichen());
System.out.println("----------------------");
for (int i = 0; i <auto3.zaehleMitarbeiter(); i++)        <-- (1)
    {
       System.out.println(auto3.holeMitarbeiter(i).getVorname()+" "
                                 +auto3.holeMitarbeiter(i).getName());
    }
```

(2)

(1) Das Attribut *fahrerliste* enthält in der Regel mehrere Elemente. Sollen diese ausgelesen werden, muss die Liste von der ersten Stelle (i = 0) bis zur letzten Stelle durchlaufen werden. Diese Steuerung erfolgt mit einer Schleife (Wiederholungsstruktur).



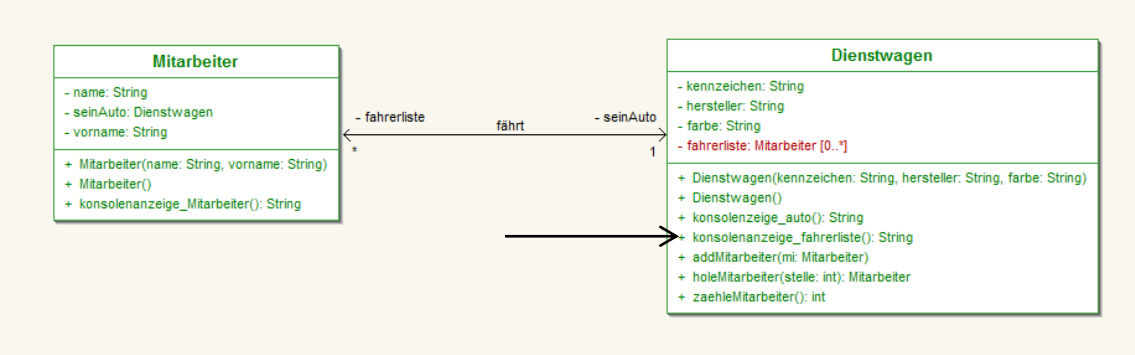
(2) Mit der Anweisung **`auto3.holeMitarbeiter(i)`** erfolgt der Zugriff auf das Mitarbeiterobjekt, das sich im Attribut *fahrerliste* an der Stelle *i* befindet.

Dadurch stehen alle Methoden des Mitarbeiterobjekts zur Verfügung, beispielsweise die Methode **`getName()`** zum Auslesen des Namens des Mitarbeiters.

Mit der Anweisung **`auto3.holeMitarbeiter(i).getName()`** wird also der Name des Mitarbeiters, der sich im Attribut *fahrerliste* des Objekts *auto3* an der *i*-ten Stelle befindet, ausgelesen.

Alternative:

Wie schon bei der Ausgabe der Mitarbeiter- und der Dienstwagendaten (siehe Seite 16) gezeigt wird, kann auch für die Daten der Mitarbeiter aus dem Attribut *fahrerliste* eine Methode in der Klasse *Dienstwagen* erstellt werden, welche die Fahrer eines Dienstwagens als Ausgabestring zurückgeben.



Mit der Methode **`public String konsolenanzeige_fahrerliste()`** wird der Ausgabestring *fahrer* aufgebaut. In der Wiederholungsstruktur werden im Referenzattribut *fahrerliste* für jedes assoziierte Mitarbeiterobjekt mit der Methode `konsolenanzeige_Mitarbeiter()` die Mitarbeiterdaten ausgelesen und zum Ausgabestring *fahrer* hinzugefügt.

```
public String konsolenanzeige_fahrerliste()
{
  String fahrer="";
  for (int i = 0; i < this.zaehleMitarbeiter(); i++)
  {
    Fahrer=fahrer + this.holeMitarbeiter(i).konsolenanzeige_Mitarbeiter()+"\n";
  }
  return fahrer;
}
```

Dadurch vereinfacht sich in der *Startklasse* der Javacode zum Ausgeben der Fahrerlisten der einzelnen Dienstwagen:

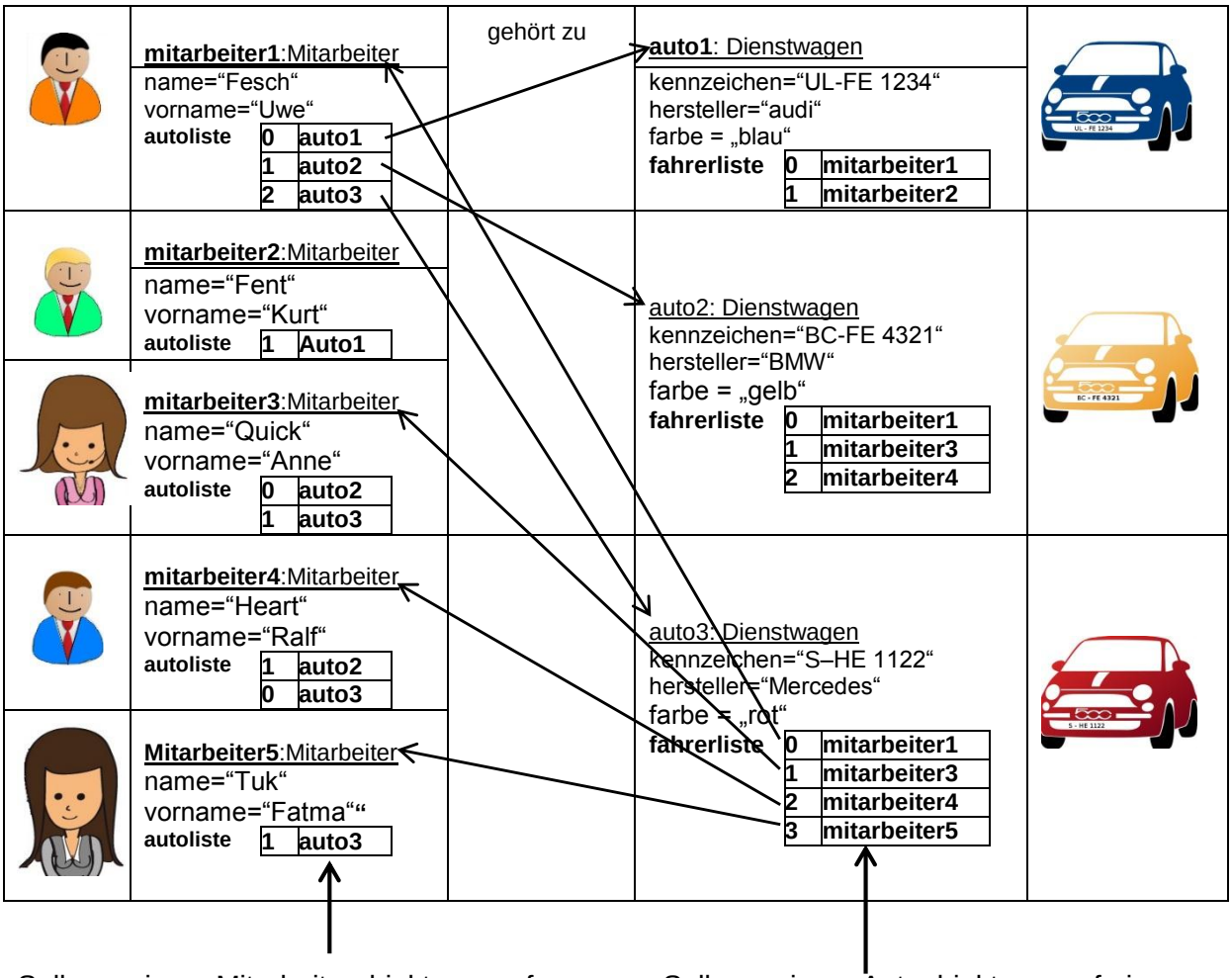
```
// Fahrerliste des Dienstwagen auto1
System.out.println("Fahrer des Dienstagen "+auto1.konsolenzeige_auto());
System.out.println("----------------------------------------");
System.out.println(auto1.konsolenanzeige_fahrerliste());
System.out.println();
// Fahrerliste des Dienstwagen auto2
System.out.println("Fahrer des Dienstagen "+auto2.konsolenzeige_auto());
System.out.println("----------------------------------------");
System.out.println(auto2.konsolenanzeige_fahrerliste());
System.out.println();
// Fahrerliste des Dienstwagen auto3
System.out.println("Fahrer des Dienstagen "+auto3.konsolenzeige_auto());
System.out.println("----------------------------------------");
System.out.println(auto3.konsolenanzeige_fahrerliste());
System.out.println();
```

4 Die M - N - Assoziation - jeder Mitarbeiter kann jedes Auto benutzen

4.1 Sachverhalt:

Die Firma GeLA GmbH stellt ihren Außendienstmitarbeitern Geschäftswagen zur Verfügung. Bisher stand den Mitarbeitern in jedem Absatzgebiet jeweils ein Dienstwagen zur Verfügung. Aus organisatorischen Gründen entschloss sich die Geschäftsleitung, den Getränkeabsatz im Standort Ehingen zu zentralisieren.
Jedem Mitarbeiter steht jetzt jeder Dienstwagen zur Verfügung. Es soll möglich sein, zu ermitteln, welcher Mitarbeiter mit welchem Dienstwagen gefahren ist, ebenso sollen nach wie vor für jeden Dienstwagen die Mitarbeiter aufgelistet werden können, die mit ihm gefahren sind.

Die untenstehende Darstellung zeigt, welche Dienstwagen der Mitarbeiter Uwe Fesch (Objekt *mitarbeiter1*) im Monat Mai 20XX gefahren hat und welche Mitarbeiter den Dienstwagen3 (Objekt *auto3*) mit dem Kennzeichen S – HE 1122 benutzt haben.

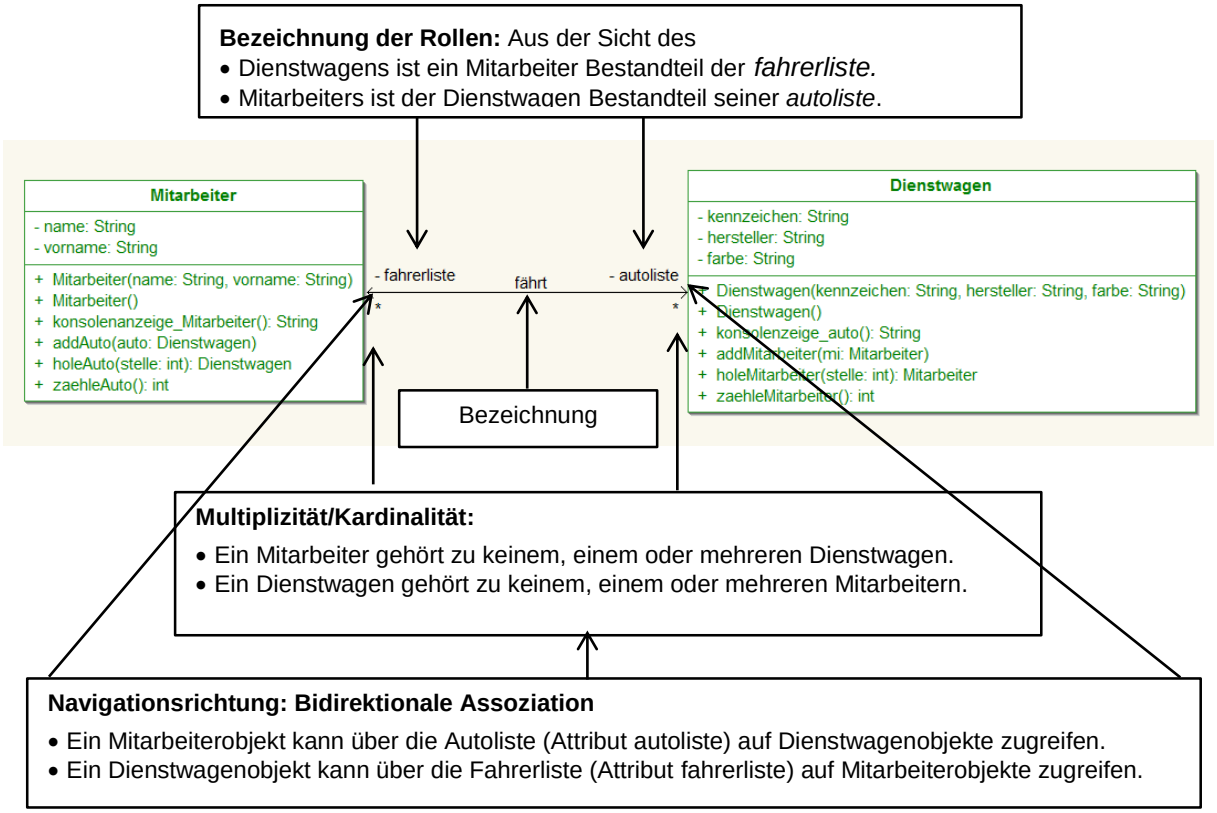


Soll von einem Mitarbeiterobjekt aus auf ein oder mehrere Dienstwagenobjekte zugegriffen werden, geschieht dies über die **Referenzliste** *autoliste*, in der als Werte die referenzierten Dienstwagenobjekte enthalten sind.

Soll von einem Autoobjekt aus auf ein oder mehrere Mitarbeiterobjekte zugegriffen werden, geschieht dies über die **Referenzliste** *fahrerliste*, in der als Werte die referenzierten Mitarbeiterobjekte enthalten sind.

Die Referenzlisten sind Objekte der Containerklasse *ArrayList.*

In der nachfolgenden Abbildung werden die Spezifikationen beim Modellieren einer M – N - Assoziation dargestellt und erläutert.



4.2 Implmentieren von M – N – Assoziationen

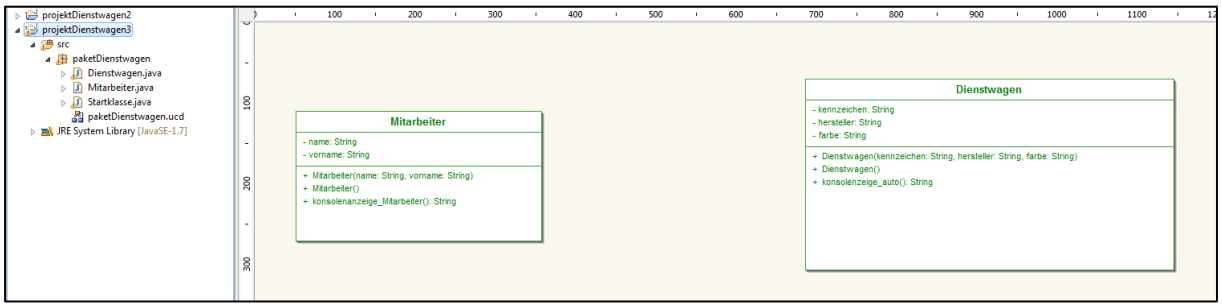
Folgende Aufgabestellungen sind zu bearbeiten:

- Das Projekt *projektDienstwagen3* mit dem Paket *paketDienstwagen* einrichten.
  Stattdessen kann auch aus dem Ordner *Objektorientierte Systementwicklung: Assoziationen (Projekte - Quellcodes)*
  das Projekt projektDienstwagen importiert und dann in *projektDienstwagen3* umbenannt und weiter bearbeitet werden (Menübefehl **`Refaktor` ➔ `Rename`**…).
  (http://www.schule-bw.de/unterricht/faecher/informatik/material/softwareentwicklung/ooa-ood-oop/)
- Mithilfe des UML-Editors *eUML2* die Klassen *Mitarbeiter* und *Dienstwagen,* wie auf Seite 4 dargestellt, modellieren.
- Assoziationen modellieren. Dabei werden die Referenzattribute angelegt und die Zugriff-methoden generiert beziehungsweise programmiert.
  - Assoziation der Klassen *Mitarbeiter* und *Dienstwagen* mit der *Multiplizität M* modellieren (ein Mitarbeiter fährt mit keinem, einem oder mehreren Dienstwagen).
  - Assoziation der Klassen *Dienstwagen* und *Mitarbeiter* mit der *Multiplizität N* modellieren (zu einem Dienstwagen gehören kein, ein oder mehrere Fahrer).
- In der Klasse *Startklasse* die Objekte mit den Objektverbindungen anlegen und im Konsolenfenster anzeigen.

Lösungen:

Das Projekt *projektDienstwagen* mit dem Paket *paketDienstwagen* importieren und in *projektDienstwagen3* umbenennen.
Zum Umbenennen des Projekts und des Pakets wird das Projekt beziehungsweise das Paket angeklickt und mit der rechten Maustaste das Kontextmenü zum Projekt/Paket geöffnet. Hier erfolgt die Auswahl des Menübefehls **`Refactor`** ➔ **`Rename`**…, um Projekt und Paket umzubenennen.

Auf Grundlage des vorliegenden Modelles *projektDienstwagen3* (siehe Abb.) werden nun die Assoziationen modelliert.



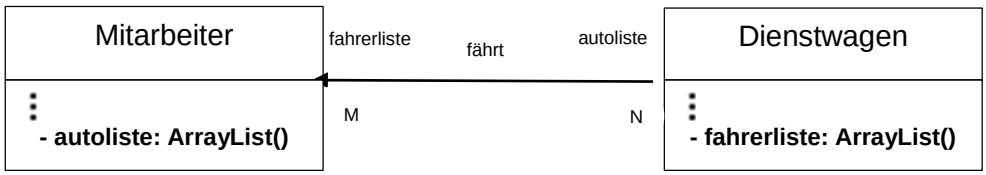
Hinweis: Mithilfe des Kontextmenüs zur Zeichenfläche (Klick mit rechter Maustaste an einer beliebigen Stelle der Zeichenfläche), wird der Menübefehl **`Preferences`** ► **`Element views`**… ausgewählt, und weitere Einstellungen für das Modell, wie das Sichtbarmachen der Referenzattribute und Rückgabetypen der Methoden sowie der Namen und Typen der Parameter festlegen (siehe auch Seite 12).

Assoziationen modellieren

Softwaretechnisch erhält jede Seite der Assoziation ein Objekt einer Containerklasse (Sammlungsklasse - *ArrayList*, siehe Seite 20).

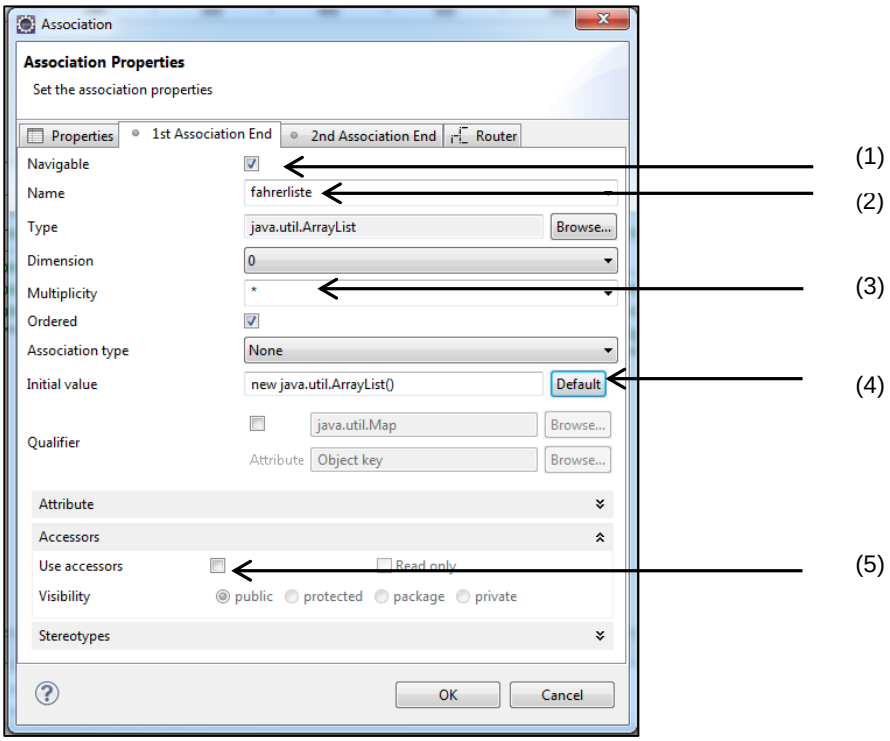
In der Darstellung des Sachverhalts auf Seite 36 wird gezeigt, wie die Dienstwagen den Mitarbeitern und die Mitarbeiter den Dienstwagen zugeordnet sind.

- Diese Zuordnung in beiden Richtungen ist auch mithilfe der Software möglich. Das bedeutet, dass sowohl die Dienstwagen den Mitarbeitern, als auch die Mitarbeiter den Dienstwagen zuzuordnen sind (= **bidirektionale Assoziation**).
- Die Assoziation *fährt* ist dadurch verwirklicht, dass Mitarbeiterobjekte ein Attribut *autoliste* besitzen, in dem alle Dienstwagen aufgeführt sind, die der Mitarbeiter benutzt hat. Die Dienstwagenobjekte besitzen ein Attribut *fahrerliste*, in der alle Mitarbeiter aufgeführt sind, die mit dem entsprechenden Dienstwagen in Beziehung stehen (**M – N - Assoziation**).

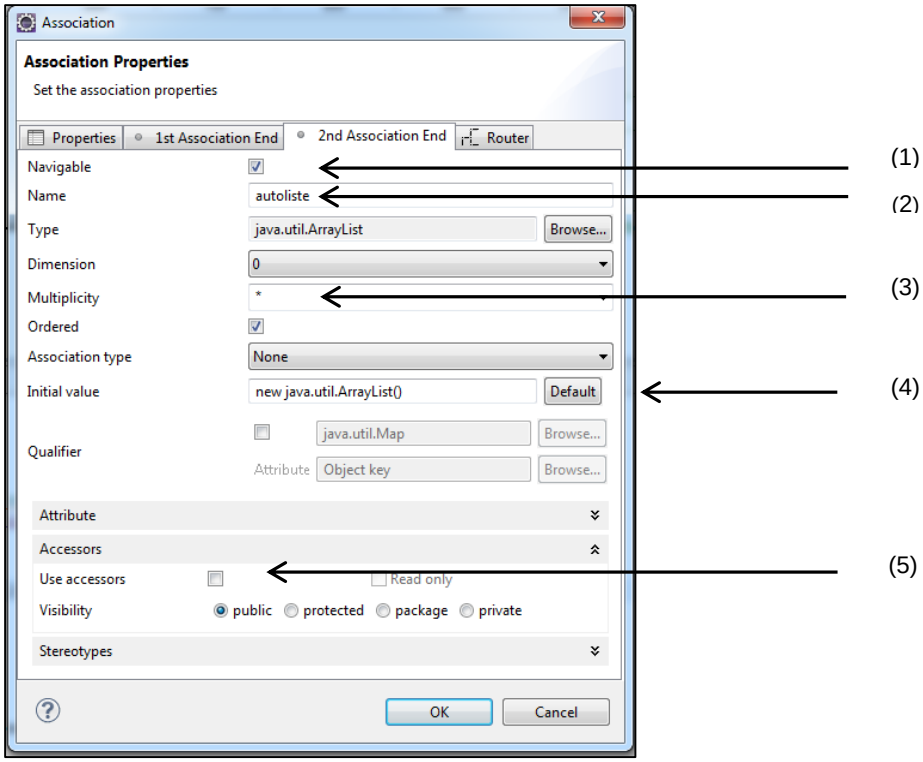


Der Ablauf beim Modellieren von 1 – N - Assoziationen ist auf den Seiten 20 ff. beschrieben, so dass im Folgenden lediglich die Entwurfsfenster der beiden Assoziationsenden gezeigt werden.

- **Das erste Assoziationsende (**Register **`[1st Association End]`)**



- **Das zweite Assoziationsende (**Register **`[2nd Association End]`)**



(1) Mit dem Kontrollfeld *Navigable* wird die Navigationsrichtung freigegeben. Im vorliegenden Beispiel endet die erste Assoziation in der Klasse *Mitarbeiter* und die zweite Assoziation in der Klasse *Dienstwagen,* so dass in beide Richtungen navigiert werden kann.

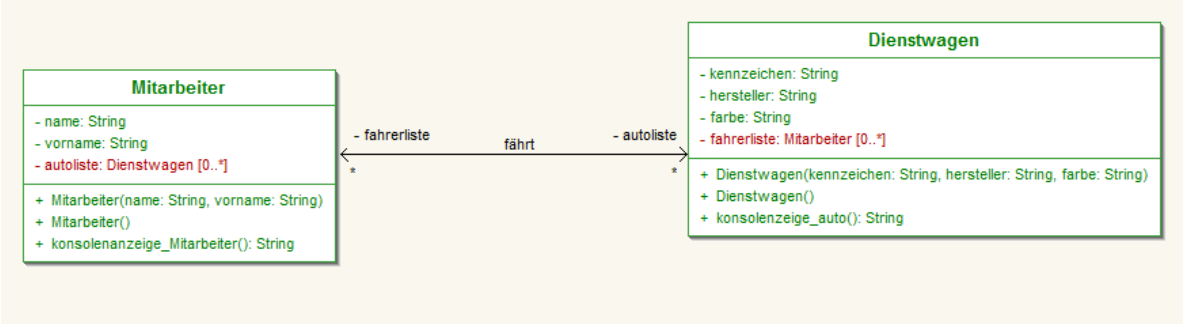
(2) Im Eingabefeld *Name* wird die Rolle, die ein Objekt in der jeweils assoziierten Klasse spielt, festgelegt. Wie in der Abbildung des Sachverhalts (siehe Seite 36) zu erkennen ist, wird das Dienstwagenobjekt *auto3* von zwei Mitarbeitern (*mitarbeiter4* und *mitarbeiter5*) gefahren. Für die aufzunehmenden Mitarbeiterobjekte wird in der Klasse *Dienstwagen* das Attribut *fahrerliste* vom Typ *ArrayList* eingerichtet. Für die aufzunehmenden Dienstwagenobjekte wird in der Klasse *Mitarbeiter* das Attribut *autoliste* vom *Typ ArrayList* aufgenommen.

(3) Hier wird die Multiplizität ausgewählt. Zu einem Dienstwagen können kein, ein oder mehrere Fahrer gehören, ein Mitarbeiter fährt mit keinem, einem oder mit mehreren Dienstwagen.

(4) Im Eingabefeld **`Initial value`** wird durch Drücken des Buttons **`[Default]`** die Klasse angezeigt, aus der das an der Assoziation beteiligte Objekt stammt. Im vorliegenden Beispiel ist dies die Klasse *ArrayList()* aus der Bibliothek *java.util*.

(5) Hier kann durch das Aktivieren beziehungsweise Deaktivieren des Kontrollfeldes **`Use accessors`** entschieden werden, ob zu der erzeugten Referenzliste *fahrerliste* Zugriffsmethoden generiert werden sollen oder nicht. Für die Objekte der Referenzliste werden anschließend eigene Zugriffsmethoden erstellt, weshalb an dieser Stelle die Option **`Use accessors`** **nicht gewählt** wird.

Als Ergebnis erhält man das UML-Diagramm mit den modellierten M – N – Assoziationen.



Javacode der modellierten Assoziationen

Durch die modellierten Assoziationen im UML-Klassendiagramm wurde für die Referenzattribute Javacode generiert.

Java-Quellcode:

```
import java.util.ArrayList;

public class Mitarbeiter {
   ⋮
   // Referenzliste erzeugen
   private ArrayList autoliste = new
                  java.util.ArrayList();
   ⋮
}
```

```
import java.util.ArrayList;

public class Dienstwagen {
   ⋮
   // Referenzliste erzeugen
   private ArrayList fahrerliste = new
                  java.util.ArrayList();
   ⋮
}
```

Für die Assoziation wird jeweils ein Attribut erzeugt, das ein Objekt der Containerklasse *ArrayList* aus der Bibliothik *java.util* ist.

Der für die Attribute erzeugte Quellcode muss noch ergänzt werden um die

- Typisierung der Attribute

  Damit in das Attribut *fahrerliste* nur Mitarbeiterobjekte und in das Attribut *autoliste* nur Dienstwagenobjekte aufgenommen werden können, wird die Quellcodezeile um den Typ der Objekte, die aufgenommen werden können, erweitert. Der aufzunehmende Typ wird in spitzen Klammern <…> angegeben.

  Typisierung des Attributs *fahrerliste* der Klasse *Dienstwagen*

```
generiert: private ArrayList fahrerliste = new java.util.ArrayList();
           ↓                             ↓                          ↓
erweitert: private ArrayList<Mitarbeiter> fahrerliste=new ArrayList<Mitarbeiter>();
```

  Typisierung des Attributs *autoliste* der Klasse *Mitarbeiter*

```
generiert: private ArrayList autoliste = new java.util.ArrayList();
           ↓                           ↓                        ↓
erweitert: private ArrayList<Dienstwagen> fahrerliste=new ArrayList<Dienstwagen>();
```

- Zugriffsmethoden für die Containerattribute

  Für die wichtigsten Operationen zur Verwaltung von Mitarbeiterobjekten im Attribut *fahrerliste* eines Dienstwagenobjekts beziehungsweise von Dienstwagenobjekte im Attribut *autoliste* eines Mitarbeiterobjekts sind Methoden für die folgenden Aufgaben bereitzustellen:
  - Aufnehmen eines Objekts in die Liste.
  - Auslesen eines Objekts aus der Liste.
  - Ermitteln der Anzahl der Objekte der Liste.

  Die Erstellung dieser Methoden wird für das Attribut *fahrerliste* auf den Seiten 26 ff ausführlich beschrieben und erläutert. Die Methoden des Attributs *autoliste* können entsprechend erstellt werden.

  Methode `public void addAuto(Dienstwagen auto)`

```
public void addAuto(Dienstwagen auto)
      {
         this.autoliste.add(auto);
      }
```

  Methode `public Dienstwagen holeAuto(int stelle)`

```
public Dienstwagen holeAuto(int stelle)
      {
            return this.autoliste.get(stelle);
      }
```

Methode `public int zaehleAuto()`

```
public int zaehleAuto()
{
      return this.autoliste.size();
}
```

4.3 Die M – N - Assoziationen in der Klasse *Startklasse* mit der main-Methode nutzen.

Die M – N - Assoziationen zwischen den Klassen *Mitarbeiter* und *Dienstwagen* sollen nun in der Startklasse benutzt werden.

Folgende Aufgabenstellungen sind zu bearbeiten:

Für die im Sachverhalt auf Seite 36 dargestellten Objekte der Klassen *Mitarbeiter* und *Dienstwagen* müssen/muss

- die angeführten Mitarbeiter- und Autoobjekte angelegt werden.
  (Auszug aus der Klasse *Startklasse* mit den erzeugten Objekten:)

```
// Mitarbeiterobjekte mit dem angepassten  erzeugen
Mitarbeiter mitarbeiter1 = new Mitarbeiter("Fesch","Uwe");
Mitarbeiter mitarbeiter2 = new Mitarbeiter("Fent","Kurt");
Mitarbeiter mitarbeiter3 = new Mitarbeiter("Quick","Anne");
Mitarbeiter mitarbeiter4 = new Mitarbeiter("Heart","Ralf");
Mitarbeiter mitarbeiter5 = new Mitarbeiter("Tuk","Fatma");

// Dienstwagenobjekte mit dem angepassten Konstruktor erzeugen
Dienstwagen auto1 = new Dienstwagen("UL-FE 1234","Audi","blau");
Dienstwagen auto2 = new Dienstwagen("BC-FE 4321","BMW","gelb");
Dienstwagen auto3 = new Dienstwagen("S-HE 1122","Mercedes","rot");
```

- die im Sachverhalt dargestellten Objektverbindungen für den Mitarbeiter Uwe Fesch (Objekt *mitarbeiter1* der Klasse *Mitarbeiter*) und den Dienstwagen Mercedes mit Kennzeichen S – HE 1122 (Objekt *auto3* der Klasse *Dienstwagen*) gebildet werden.

- die Ausgabe der Fahrzeuge, die Uwe Fesch gefahren hat, beziehungsweise die Mitarbeiter, die mit dem Dienstwagen Mercedes mit Kennzeichen S – HE 1122 gefahren sind, im Konsolenfenster vorgenommen werden.

  Konsolenanzeige der Dienstwagen des Mitarbeiters Uwe Fesch:

```
Dienstwagen von Uwe Fesch
--------------------------------
Audi, UL-FE 1234, blau
BMW, BC-FE 4321, gelb
Mercedes, S - HE 1122, rot
```

  Konsolenanzeige der Fahrer des Dienstwagens mit dem Kennzeichen S – HE 1122:

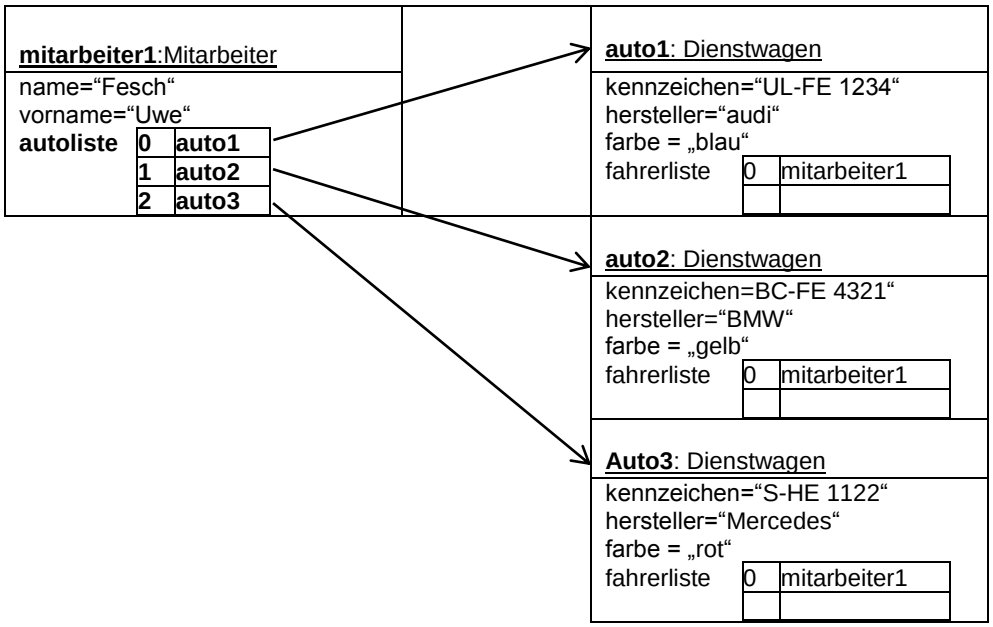
```
Fahrer des Dienstwagen  Mercedes, S - HE 1122, rot
---------------------------------------
Uwe Fesch
Anne Quick
Ralf Heart
Fatma Tuk
```

Nachfolgend wird beschrieben, wie die Objektverbindungen in der Startklasse realisiert werden:

- *Ein Mitarbeiter „kennt" die von ihm gefahrenen Dienstwagen.*

  Uwe Fensch hat drei Dienstwagen benutzt. Das Objekt *mitarbeiter1* ist daher mit den Objekten *auto1, auto2 und auto3* über das Attribut *autoliste* assoziiert.

  Im Attribut *autoliste* sind als Werte die assoziierten Objekte *auto1, auto2* und *auto3* enthalten. Sie sind hintereinander angeordnet und nummeriert (indiziert).



Javacode für die Assoziationen

```
// mitarbeiter1 fährt mit aut1, auto2 und auto3
mitarbeiter1.addAuto(auto1);
mitarbeiter1.addAuto(auto2);
mitarbeiter1.addAuto(auto3)
```

Anzeige der Elemente des Attributs *autoliste* des Objekts *mitarbeiter1*

```
// Autoliste des Mitarbeiters 1
System.out.println("Autoliste "+ mitarbeiter1.konsolenanzeige_Mitarbeiter());
System.out.println("----------------------");
for (int i = 0; i <mitarbeiter1.zaehleAuto(); i++)        (1)
    {
       System.out.println(mitarbeiter1.holeAuto(i).konsolenzeige_auto());
    }
                                                          (2)
```

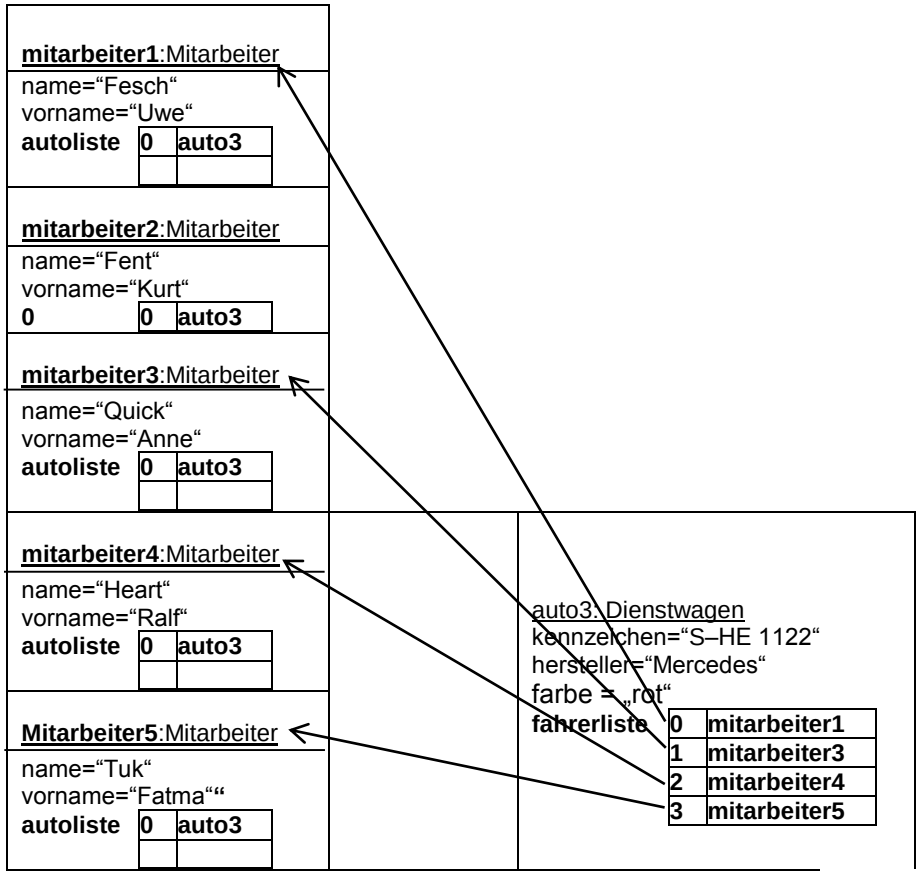
Mit der Methode *zaehleAuto()* wird die Anzahl der Elemente ermittelt, die das Attribut *autoliste* enthält. Beginnend mit dem Wert i = 0 werden die Einträge des Attributs *autoliste* von der ersten bis zur letzten Stelle durchlaufen (1).

Mit der Anweisung `mitarbeiter1.holeAuto(i)` erfolgt der Zugriff auf das Autoobjekt, das sich im Attribut *autoliste* an der Stelle *i* befindet. Über das Autoobjekt stehen alle Methoden der Klasse *Auto* zur Verfügung, auch die Methode **`konsolenanzeige_auto()`**, welche die Daten des Dienstwagens als Ausgabestring liefert (2).
(Siehe auch Seite 16)

- *Ein Dienstwagen „kennt“ seine Fahrer (Mitarbeiter*).

  Mit dem Dienstwagen Mercedes sind die Mitarbeiter Uwe Fesch, Anne Quick, Ralf Heart und Fatma Tuk gefahren. Das Objekt *auto3* ist daher mit den Objekten *mitarbeiter1, mitarbeiter3, mitarbeiter4 und mitareiter5* über das Attribut *fahrerliste* assoziiert.

  Im Attribut *fahrerliste* sind als Werte die assoziierten Objekte *mitarbeiter1, mitarbeiter3, mitarbeiter4* und *mitareiter5* enthalten. Sie sind hintereinander angeordnet und nummeriert (indiziert).



Javacode für die Assoziationen

```
// Assoziationen 2: Fahrer von Dienstwagen 3
auto3.addMitarbeiter(mitarbeiter1); auto3.addMitarbeiter(mitarbeiter3);
auto3.addMitarbeiter(mitarbeiter4); auto3.addMitarbeiter(mitarbeiter5);
```

Anzeige der Elemente des Attributs *fahrerliste* des Objekts *auto3*

```
System.out.println("Fahrer des Dienstwagen  "+auto3.konsolenzeige_auto());
System.out.println("----------------------------------------");
for (int i = 0; i < auto3.zaehleMitarbeiter(); i++)        (1)
   {
     System.out.println(auto3.holeMitarbeiter(i).konsolenanzeige_Mitarbeiter());
   }
```

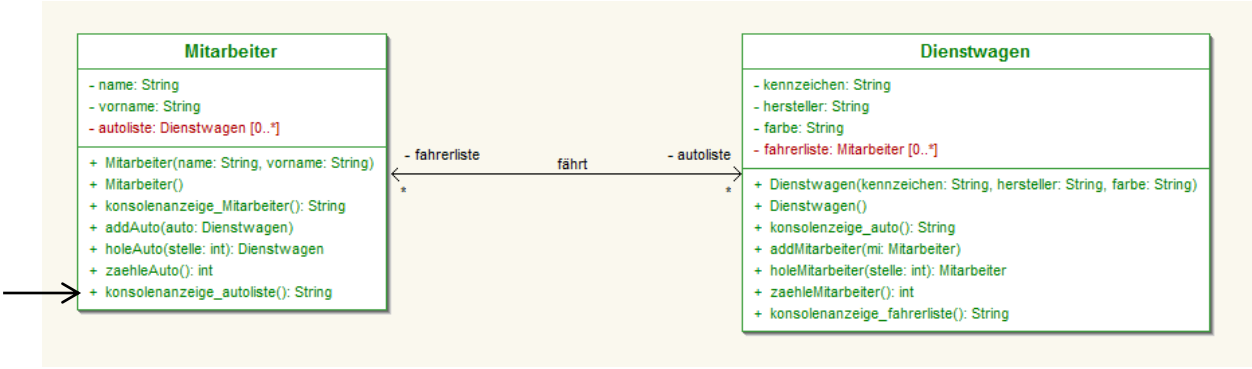
(2)

Mit der Methode *zaehleMitarbeiter()* wird die Anzahl der Elemente ermittelt, die das Attribut *fahrerliste* enthält. Beginnend mit dem Wert i = 0 werden die Einträge des Attributs *fahrerliste* von der ersten bis zur letzten Stelle durchlaufen (1).

Mit der Anweisung `auto3.holeMitarbeiter(i)` erfolgt der Zugriff auf das Autoobjekt, das sich im Attribut *fahrerliste* an der Stelle *i* befindet. Über das Mitarbeiterobjekt stehen alle Methoden dieses Objekts zur Verfügung, beispielsweise die Methode `konsolenanzeige_Mitarbeiter()`, welche die Daten des Mitarbeiters als Ausgabestring liefert (2).
(siehe auch Seite 16)

Alternative:

Wie bei der Ausgabe der Mitarbeiterdaten aus dem Attribut *fahrerliste* der Klasse *Dienstwagen* (siehe Seite 35), wird auch für die Daten der Dienstwagen aus dem Attribut *autoliste* eine Methode in der Klasse *Mitarbeiter* erstellt, welche die Dienstwagen eines Mitarbeiters als Ausgabestring zurückgeben.



Mit der Methode **`public String konsolenanzeige_autoliste()`** wird der Ausgabestring *auto* aufgebaut. In der Wiederholungsstruktur werden im Referenzattribut *autoliste* für jedes assoziierte Dienstwagenobjekt mit der Methode `konsolenanzeige_Auto()` die Dienstwagendaten ausgelesen und zum Ausgabestring *auto* hinzugefügt.

```
public String konsolenanzeige_autoliste() {
String auto="";
for (int i = 0; i < this.zaehleAuto(); i++)
      {
            auto=auto + this.holeAuto(i).konsolenzeige_auto() + "\n";
      }
return auto;
}
```

Dadurch vereinfacht sich in der Startklasse der Javacode zum Ausgeben der Fahrerlisten der einzelnen Dienstwagen und der Autolisten der Mitarbeiter:

```
System.out.println("Dienstwagen von
                              "+mitarbeiter1.konsolenanzeige_Mitarbeiter());
System.out.println("----------------------------------------");
System.out.println(mitarbeiter1.konsolenanzeige_autoliste());
System.out.println();
System.out.println("Fahrer des Dienstwagen  "+auto3.konsolenzeige_auto());
System.out.println("----------------------------------------");
System.out.println(auto3.konsolenanzeige_fahrerliste());
```

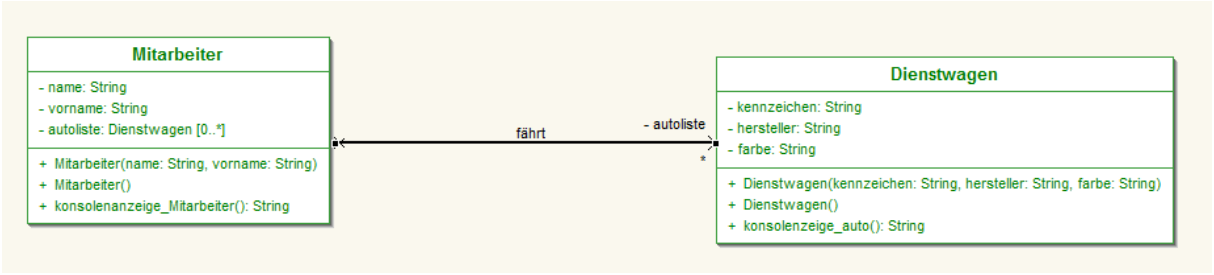
5 N - Assoziation ohne Kapselung des Containerzugriffs

Bei der bisherigen Vorgehensweise wurden beim Modellieren der Assoziationen mit der Multiplizität N die Zugriffsmethoden für die Attribute *fahrerliste* und *autoliste* in den Methoden *addMitarbeiter()/addAuto*,() *holeMitarbeiter()/holeAuto()* sowie *zaehleMitarbeiter()/zaehleAuto()* gekapselt und können nur mit diesen Methoden manipuliert werden. Deshalb wurden beim Modellieren der Attribute *fahrerliste* und *autoliste* keine Zugriffsmethoden (*Accessors*) angelegt. Die entsprechende Option wurde deaktiviert (siehe Seite 21).

Am Beispiel der Dienstwagen, mit denen der Mitarbeiter Fesch gefahren ist (Objekt *mitarbeiter1,* siehe vorherige Seite), soll gezeigt werden, wie auf das Attribut *autoliste* direkt zugegriffen werden kann.

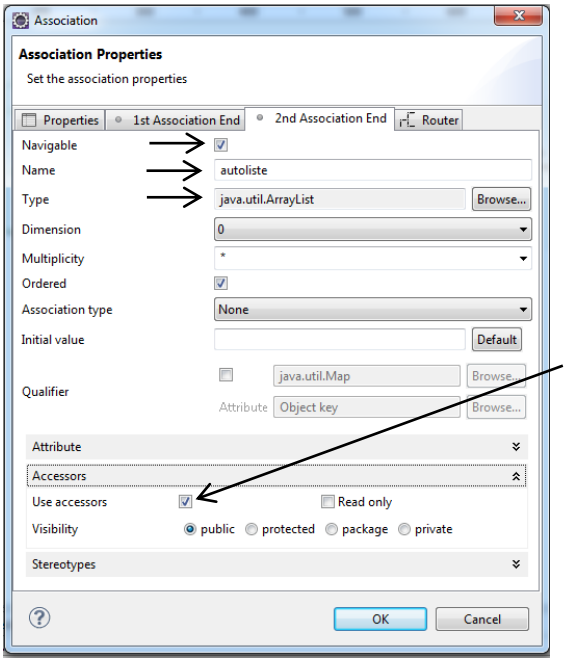
Folgende Aufgaben sind zu bearbeiten:

- Das Projekt *projektDienstwagen4* mit dem Paket *paketDienstwagen* einrichten. Mithilfe des UML-Editors *eUML2* die Klassen *Mitarbeiter* und *Dienstwagen* wie auf Seite 4 dargestellt, modellieren. Stattdessen kann auch aus dem Ordner *Objektorientierte Systementwicklung: Assoziationen (Projekte - Quellcodes) das Projekt projektDienstwagen* importiert werden und dann in *projektDienstwagen4* umbenannt und weiter bearbeitet werden. (Menübefehl **`Refaktor` ➔ `Rename`**…)
  (http://www.schule-bw.de/unterricht/faecher/informatik/material/softwareentwicklung/ooa-ood-oop/)
- Modellieren der unidirektionalen Assoziation mit der Multiplizität N. Ein Mitarbeiter fährt mit keinem, einem oder mehreren Dienstwagen.



Lösung:

Mit dem Symbol  wird das Modellieren der Assoziation gestartet und zuerst die Klasse *Mitarbeiter* und dann die Klasse *Dienstwagen* angeklickt („von links nach rechts“). Anschließend wird die Assoziation beschriftet, danach werden die „Assoziationsenden“ modelliert.



Während im Register **`1st Association End`** in diesem Beispiel die Option **`Navigable`** deaktiviert bleibt, wird

- im Register **`2nd Association End`**
  - die Option **Navigable** aktiviert,
  - der Attributname *autoliste* vergeben,
  - als Attributtype die Klasse *ArrayList* ausgewählt (Button **`[Browse]`**).
- in der Eigenschaft **`Accessors`** die Option **`Use Accessors`** **aktiviert** belassen.

Für das Attribut *autoliste* wird in der Klasse *Mitarbeiter* der folgende Javacode generiert, der dann noch vervollständigt werden muss.

generierter Javacode	**vervollständigter Javacode**

`// Attribut autoliste der Klasse ArrayList`
`// deklarieren`
 	
`private ArrayList autoliste;` **(1)** →
 	
`// Zugriffsmethoden`
 	
`// get-Methode`
 	
`public ArrayList getAutoliste() {`
`  return autoliste;`
`}`
 	
`// set-Methode`
 	
`public void setAutoliste(ArrayList autoliste)`
`{`
`  this.autoliste = autoliste;`
`}`
	`private ArrayList<Dienstwagen> autoliste  =`
	`new ArrayList<Dienstwagen>();`
	
	`public ArrayList<Dienstwagen>`
	`  getAutoliste()`
	`{`
	`  return autoliste;`
	`}`
	(2) ↕
	`public void setAutoliste(`
	`  ArrayList<Dienstwagen> autoliste)`
	`{`
	`  this.autoliste = autoliste;`
	`}`

Folgende Ergänzungen müssen vorgenommen werden:

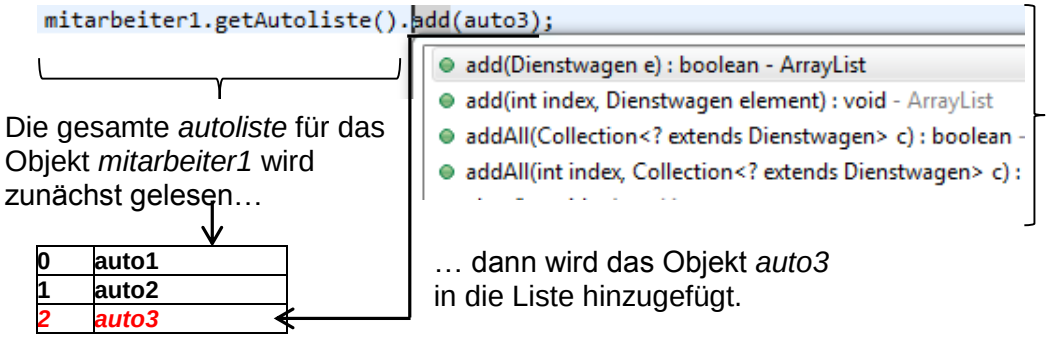
(1) „Typisierung“ und Initialisierung des Attributs *autoliste* aus der Klasse *ArrayList.* In das Attribut *autoliste* dürfen nur Dienstwagenobjekte aufgenommen werden.

(2) „Typisierung“ des Attributs *autoliste* in den Zugriffsmethoden.

Zugriff auf das Attribut *autoliste* in der *Startklasse*

Soll das Attribut *autoliste* bearbeiten werden (hinzufügen, lesen, löschen eines Dienstwagenobjekts), muss zuerst die gesamte Liste gelesen werden, um dann die Methoden der Klasse *ArrayList* auf die gesamte Liste anwenden zu können.

Beispiel: Das Dienstwagenobjekt *auto* soll in das Attribut *autoliste* des Mitarbeiterobjekts *mitarbeiter1* aufgenommen werden.



Hinweis:

Für das Attribut *autoliste* stehen alle Methoden der Klasse *ArrayList* zur Verfügung. Dies bedeutet eine Verletzung des Prinzips der „Kapselung“.

Javacode der Klasse *Startklasse*

```
public class Startklasse {

   public static void main(String[] args)
   {
      // Mitarbeiterobjekte mit dem angepassten erzeugen
      Mitarbeiter mitarbeiter1 = new Mitarbeiter("Fesch","Uwe");

      // Dienstwagenobjekte mit dem angepassten Konstruktor erzeugen
      Dienstwagen auto1 = new Dienstwagen("UL-FE 1234","Audi","blau");
      Dienstwagen auto2 = new Dienstwagen("BC-FE 4321","BMW","gelb");
      Dienstwagen auto3 = new Dienstwagen("S - HE 1122","Mercedes","rot");

      // Assoziation: Autoliste des Objekts mitarbeiter1
      mitarbeiter1.getAutoliste().add(auto1);
      mitarbeiter1.getAutoliste().add(auto2);
      mitarbeiter1.getAutoliste().add(auto3);

      // Konsolenausgabe: autoliste des Objekts mitarbeiter1
      System.out.println("Autoliste "+mitarbeiter1.konsolenanzeige_Mitarbeiter());
      System.out.println("-----------------------------------");
      for (int i = 0; i < mitarbeiter1.getAutoliste().size(); i++)
      {
         System.out.println(mitarbeiter1.getAutoliste().get(i).konsolenzeige_auto());
      }
   }
}
```

(1) (2) (3)

Erläuterung:

(1) Initialisierung des Mitarbeiter- und der Dienstwagenobjekte mithilfe angepasster Konstruktoren.

(2) Zuordung der Dienstwagenobjekte zum Mitarbeiterobjekt *mitarbeiter1*. Der Mitarbeiter fährt mit mehreren Dienstwagen.

(3) Auslesen und Anzeigen der Objekte des Attributs *autoliste*.

```
System.out.println(mitarbeiter1.getAutoliste().get(i).konsolenzeige_auto());
```

In der Wiederholungsstruktur wird

die gesamte *autoliste* für das Objekt *mitarbeiter1* gelesen…

… und auf die Objekte an der jeweiligen Stelle i zugegriffen, um die Methode für die Konsolenanzeige der Autodaten aufzurufen.

6 Assoziation mit Ausprägung - Assoziationsklasse

6.1 Sachverhalt

Um einen besseren Überblick über die Benutzung der Dienstfahrzeuge zu erhalten, wünscht die Vertriebsleitung der GeLa GmbH, dass neben der Nutzung der Fahrzeuge zusätzlich noch festgehalten wird, an welchem Termin welcher Mitarbeiter ein Auto fuhr und wie viele Kilometer zurückgelegt wurden. Außerdem soll festgehalten werden, ob die Fahrt dienstliche Gründe hatte oder nicht.

Das Fahrtenbuch für den Dienstwagen 3 (Objekt *auto3*) weist beispielsweise folgende Eintragungen auf:

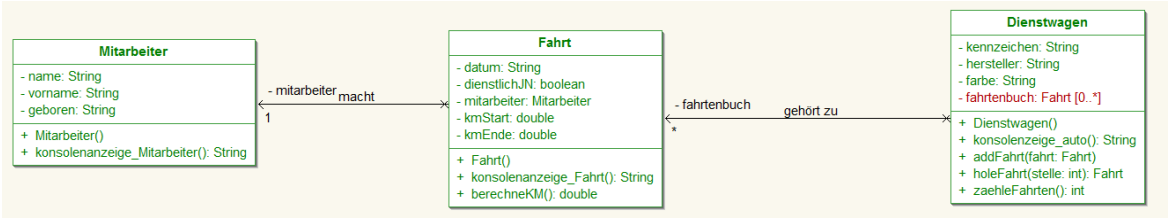
GeLa GmbH Fahrtenbuch Seite 1

Dienstwagen:	**S – HE 1122**	Fabrikat	**Mercedes**	Farbe	**rot**

Fahrten:

Nr	Datum	Name, Vorname	KM Start	KM Ende	dienstlich (X)
1	18.04.2013	Heart, Ralf	10500	11000	
2	21.04.2013	Quick, Anne	11000	11300	
3	22.04.2013	Fesch, Uwe	11300	11450	
4	23.04.2013	Tuk, Fatma	11450	12050	X

Würden die Attribute *datum*, *gefahrenekm* und *dienstlich* den Mitarbeitern oder den Dienstwagen zugeordnet, könnten für jedes Mitarbeiterobjekt beziehungsweise für jedes Dienstwagenobjekt nur Werte für eine Fahrt gespeichert werden. Die nicht eindeutig zuordenbaren Attribute werden deshalb in eigenen Objekten gespeichert, für die eine weitere Klasse *Fahrt* modelliert wird.



In der Klasse *Dienstwagen* werden die Fahrten im Attribut *fahrtenbuch* gesammelt. Das Attribut *fahrtenbuch* ist ein Objekt der Klasse *ArrayList*. Dadurch können mehrere Fahrten gespeichert werden. Es besteht zur Klasse *Fahrt* eine unidirektionale Assoziation mit der Multiplizität N. Die Klasse *Fahrt* enthält zur Klasse *Mitarbeiter* das Referenzattribut *mitarbeiter* vom Typ der Klasse *Mitarbeiter*. Es besteht also von der Klasse *Fahrt* zur Klasse *Mitarbeiter* eine unidirektionale Assoziation mit der Multiplizität 1.
Im Attribut *fahrtenbuch* werden die Objekte der Klasse *Fahrt* aufgenommen. Auf die Daten der Mitarbeiter kann dann über das Attribut *mitarbeiter* der Klasse *Fahrt* zugegriffen werden.

Die Klasse *Fahrt* verfügt sowohl über die Eigenschaften einer Klasse als auch über die einer Assoziation, da sie sowohl mit der Klasse *Mitarbeiter* als auch mit der Klasse *Dienstwagen* verbunden ist. Man spricht daher auch von einer *attribuierten Assoziation* beziehungsweise von einer *Assoziationsklasse* [1].

[1] Siehe Bernd Oestereich, Analyse und Design mit UML2.1, Oldenbourg Verlag München Wien, 8. Auflage, S. 349

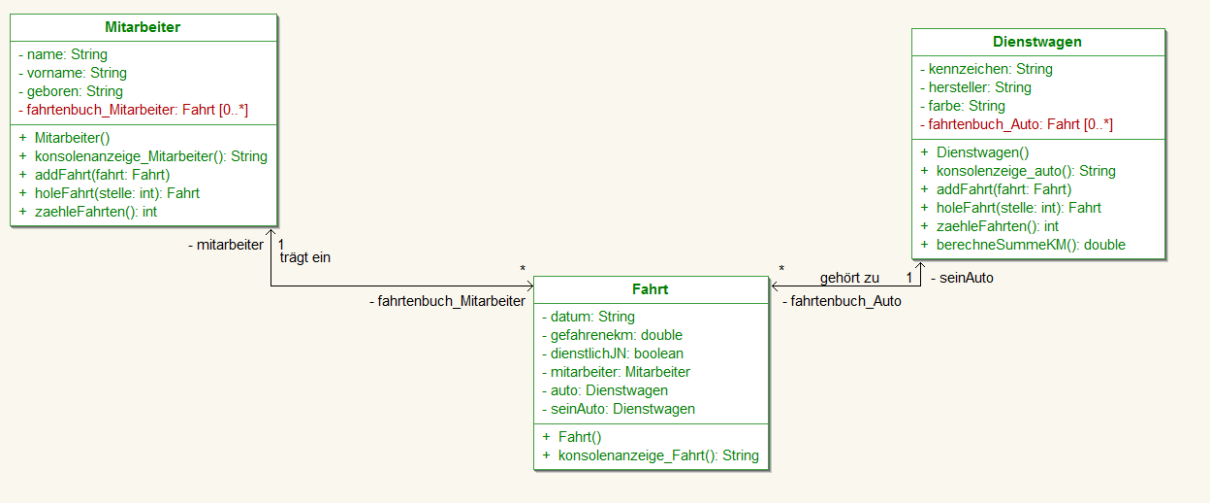
Hinweis:

Im oben dargestellten Klassendiagramm besteht jeweils eine unidirektionale Assoziation von der Klasse *Dienstwagen* zur Klasse *Fahrt* und von der Klasse *Fahrt* zur Klasse *Mitarbeiter*.

- Mit einem Auto (Objekt der Klasse *Dienstwagen*) werden keine, eine oder mehrere Fahrten gemacht.
- Ein Eintrag im Fahrtenbuch (Objekt der Klasse *Fahrt*) wird immer von einem Mitarbeiter gemacht.

Von den Mitarbeiterobjekten aus kann nicht auf Objekte der Klasse *Fahrt* und von den Objekten der Klasse *Fahrt* nicht auf Objekte der Klasse *Dienstwagen* zugegriffen werden.

Denkbar wären auch bidirektionale Assoziationen zwischen den Klassen *Mitarbeiter* und *Fahrt* und den Klassen *Dienstwagen* und *Fahrt*. In diesem Falle müssen auch die Mitarbeiter ein Fahrtenbuch führen (Attribut *fahrtenbuch_Mitarbeiter* vom Typ *ArrayList*). Außerdem muss in der Klasse *Fahrt* das Attribut *auto* vom Typ *Dienstwagen* eingefügt werden (siehe nachfolgendes UML-Klassendiagramm).



Im erweiterten Modell

- kann ein Mitarbeiter (Objekt der Klasse *Mitarbeiter*) keine, eine oder mehrere Fahrten machen.
- gehört ein Eintrag im Fahrtenbuch des Mitarbeiters (Objekt der Klasse *Fahrt*) immer zu einem Auto.

In den weiteren Ausführungen werden nur die unidirektionale Assoziationen zur Führung des Fahrtenbuches für die Dienstwagen, wie sie auf Seite 49 beschrieben sind, dokumentiert. Das Modell kann dann an späterer Stelle als Wiederholung und Vertiefung erweitert werden.

6.2 Implementieren der Assoziation mit Ausprägung

Folgende Aufgaben sind zu bearbeiten:

- Das Projekt *projektDienstwagen5* mit dem Paket *paketDienstwagen* einrichten. Mithilfe des UML-Editors *eUML2* die Klassen *Mitarbeiter, Dienstwagen* und *Fahrt,* wie auf Seite 49 dargestellt, modellieren.
  Stattdessen kann auch aus dem Ordner *Objektorientierte Systementwicklung: Assoziationen (Projekte - Quellcodes) das Projekt projektDienstwagen* importiert werden und dann in *projektDienstwagen5* umbenannt und weiter bearbeitet werden (Menübefehl **`Refaktor Rename`**…).
  (http://www.schule-bw.de/unterricht/faecher/informatik/material/softwareentwicklung/ooa-ood-oop/)
- Modellieren der Klasse *Fahrt* sowie der Assoziationen wie im Modell auf Seite 49 dargestellt.
  - Für einen Dienstwagen wird ein Fahrtenbuch geführt, in dem mehrere (keine, eine oder mehrere) Fahrten eingetragen werden können.
  - Eine Fahrt muss von einem Mitarbeiter gemacht werden.
  - Aus den Daten einer Fahrt müssen die gefahrenen Kilometer errechnet werden (Methode berechneKM).

Lösungen

- Nachdem das Projekt *projektDienstwagen* importiert und in *projektDienstwagen_5* umbenannt ist, wird die Klasse *Fahrt* ohne die Assoziationen modelliert.

```
public class Fahrt {
	// Attribut datum mit Zugriffsmethoden
	private String datum;
	public String getDatum() {
		return datum;
	}
	public void setDatum(String datum) {
		this.datum = datum;
	}
	// Attribut dienstlich mit Zugriffsmethoden
	private boolean dienstlich;
	public boolean isDienstlich() {
		return dienstlich;
	}
	public void setDienstlich(boolean dienstlich) {
		this.dienstlich = dienstlich;
	}
	// Attribut kmStart mit Zugriffsmethoden
	private double kmStart;
	public double getKmStart() {
		return kmStart;
	}
	public void setKmStart(double kmStart) {
		this.kmStart = kmStart;
	}
	// Attribut kmEnde mit Zugriffsmethoden
	private double kmEnde;
	public double getKmEnde() {
		return kmEnde;
	}
	public void setKmEnde(double kmEnde) {
		this.kmEnde = kmEnde;
	}
	// Standardkonstruktor
	public Fahrt(){
	}
	// Methode berechnen zum Berechnen der gefahrenen KM
	public double berechneKM(){
		return this.kmEnde - this.kmStart;
	}
}
```

- Modellieren der Assoziationen

Die Klasse *Fahrt* dient als Verbindungsklasse. Mit ihr ist die Klasse *Dienstwagen* verbunden (Referenzattribut *fahrtenbuch*) und sie ist mit der Klasse *Mitarbeiter* verbunden (Referenzattribut *mitarbeiter).*

Während beim Modellieren der Assoziation mit der Multiplizität 1 zwischen den Klassen *Fahrt* und *Mitarbeiter* (eine Fahrt muss von einem Mitarbeiter gemacht werden) beim Anlegen des Referenzattributs *mitarbeiter* die Zugriffsmethoden generiert werden, muss der Quellcode für das Referenzattribut *fahrtenbuch* der Assoziation mit der Multiplizität N zwischen den Klassen *Dienstwagen* und *Fahrt* (mit einem Dienstwagen werden mehrere Fahrten gemacht) noch ergänzt werden (die Vorgehensweise ist auf der Seite 20 beschrieben).

Im nachfolgenden Quellcode der Klasse *Fahrt* sind lediglich der angepasste Konstruktor sowie die für die Herstellung der jeweiligen Assoziation benötigten Referenzattribute mit deren Zugriffsmethoden dargestellt.

Klasse *Fahrt*

```
public class Fahrt {
//angepasster Konstrukor
     public Fahrt (Mitarbeiter mitarbeiter, String datum, int kmStart, int kmEnde,
                                                                  boolean dienstlichJN)
     {
          this.mitarbeiter = mitarbeiter;
          this.datum = datum;
          this.kmStart=kmStart;
          this.kmEnde=kmEnde;
          this.dienstlichJN = dienstlichJN;
     }
     //Referenzattribut mitarbeiter für die
     //Verbindung zur Klasse Mitarbeiter
     private Mitarbeiter mitarbeiter;
     //Zugriffsmethoden
     public Mitarbeiter getMitarbeiter()
     {
          return mitarbeiter;
     }
     public void setMitarbeiter(Mitarbeiter mitarbeiter)
     {
          this.mitarbeiter = mitarbeiter;
     }
}
```

Durch die Aufnahme des Referenzattributs *mitarbeiter* in den Konstruktor wird erreicht, dass beim Erzeugen eines Objekts der Klasse *Fahrt* (also beim Eintragen einer Fahrt) ein Mitarbeiter mit angegeben werden muss. (***Muss-Assoziation***).

Klasse *Dienstwagen*

```
public class Dienstwagen {

   //Referenzattribut fahrtenbuch (Objekt der Klasse ArrayList)
   private ArrayList<Fahrt> fahrtenbuch = new ArrayList<Fahrt>();
   //Zugriffsmethoden
   //Anfügen von Objekten der Klasse Fahrt in das Attribut fahrtenbuch der Klasse
   //Dienstwagen
   public void addFahrt(Fahrt fahrt)
   {
        fahrtenbuch.add(fahrt);
   }
   //Auslesen von Objekten der Klasse Fahrt aus dem Attribut fahrtenbuch der Klasse
                                                                        Dienstwagen
   public Fahrt holeFahrt(int stelle)
   {
        return this.fahrtenbuch.get(stelle);
   }
   //Ermitteln der Anzahl der Objekte im Attribut fahrtenbuchder Klasse Dienstwagen
   public int zaehleFahrten()
   {
        return fahrtenbuch.size();
   }
}
```

6.3 Anwenden der Assoziationsklasse in der *Startklasse*

Aufgabenstellung

In der *Startklasse* sollen

(1) die Mitarbeiter- und Dienstwagenobjekte, wie im Sachverhalt auf Seite 36 dargestellt, erzeugt und die Attributwerte eingelesen werden.

(2) die Fahrten für das Dienstfahrzeug mit dem Kennzeichen S – HE 1122 (Objekt *auto3*), wie im Fahrtenbuch auf Seite 49 dargestellt, in das Attribut *fahrtenbuch* des Objekts *auto3* der Klasse *Dienstfahrzeug* aufgenommen werden.

(3) alle Fahrten, die mit dem Dienstwagen *auto3* (Kennzeichen S-HE 1122) gemacht wurden, aufgelistet werden. Die Liste soll folgenden Aufbau haben.

```
Fahrten des Dienstwagen Mercedes, S - HE 1122, rot
Datum           KM     dienstl. Fahrer
18.04.2013      500.0   Nein    Ralf Heart
21.04.2013      300.0   Nein    Anne Quick
22.04.2013      150.0   Nein    Uwe Fesch
23.04.2013      600.0   Ja      Fatma Tuk
```

Hinweis:

In der Klasse *Fahrt* hat das Attribut dienstlich den Datentyp *boolean*. Das bedeutet, dass als Attributwerte die Werte „*true*“ oder „*false*“ möglich sind. Die Methode *private String jn()* in der Klasse *Fahrt* wird von der Methode *String konsolenanzeige_Fahrt()* aufgerufen. Dadurch wird erreicht, dass der Text „Ja“ zurückgegeben wird, sofern das Attribut *dienstlichJN* den boole’schen Wert *true* aufweist. Enthält das Attribut *dienstlichJN* den boole’schen Wert *false,* liefert die Methode den Text „Nein“ zurück.

```
private String jn()
{
if (this.dienstlichJN)
       {
              return "Ja";
       }
else
       {
               return "Nein";
       }
}

public String konsolenanzeige_Fahrt()
{
       return this.datum+"\t"+ this.berechneKM()+"\t"+this.jn();
}
```

Lösungsvorschlag für den Quellcode der *Startklasse*

```
public class Startklasse {
public static void main(String[] args)
{
  // Mitarbeiterobjekte mit dem angepassten  erzeugen
  Mitarbeiter mitarbeiter1 = new Mitarbeiter("Fesch","Uwe");
  Mitarbeiter mitarbeiter2 = new Mitarbeiter("Fent","Kurt");
  Mitarbeiter mitarbeiter3 = new Mitarbeiter("Quick","Anne");
  Mitarbeiter mitarbeiter4 = new Mitarbeiter("Heart","Ralf");
  Mitarbeiter mitarbeiter5 = new Mitarbeiter("Tuk","Fatma");

  // Dienstwagenobjekte mit dem angepassten Konstruktor erzeugen
  Dienstwagen auto1 = new Dienstwagen("UL-FE 1234","Audi","blau");
  Dienstwagen auto2 = new Dienstwagen("BC-FE 4321","BMW","gelb");
  Dienstwagen auto3 = new Dienstwagen("S-HE 1122","Mercedes","rot");

  // Fahrten anlegen
  Fahrt fahrt1 = new Fahrt(mitarbeiter4, "18.04.2013", 10500,11000, false);
  Fahrt fahrt2 = new Fahrt(mitarbeiter3, "21.04.2013", 11000,11300, false);
  Fahrt fahrt3 = new Fahrt(mitarbeiter1, "22.04.2013", 11300,11450, false);
  Fahrt fahrt4 = new Fahrt(mitarbeiter5, "23.04.2013", 11450,12050, true);

  // Fahrtenbucheinträge für den Dienstwagen auto3
  auto3.addFahrt(fahrt1);
  auto3.addFahrt(fahrt2);
  auto3.addFahrt(fahrt3);
  auto3.addFahrt(fahrt4);

  // Alle Fahrten von Dienstwagen 3
  // Hersteller, kennzeichen, Datum, KM, Fahrer
  System.out.println("Fahrten des Dienstwagen "+auto3.konsolenzeige_auto());
  System.out.println("Datum           KM     dienstl. Fahrer");
  for (int i = 0; i < auto3.zaehleFahrten(); i++)
    {
     System.out.println(auto3.holeFahrt(i).konsolenanzeige_Fahrt() + "\t"
        + auto3.holeFahrt(i).getMitarbeiter().konsolenanzeige_Mitarbeiter());
    }
  }
}
```

Alternative:

In die Klasse *Dienstwagen* wird die Methode **`konsolenausgabe_fahrtenbuch()`** aufge nommen. Diese Methode baut in einer Wiederholungsstruktur den Ausgabestring *fb* auf. Im Referenzattribut *fahrtenbuch* werden für jedes assoziierte Objekt der Klasse *Fahrt* mit der Methode `konsolenanzeige_Fahrt()` die Daten der Fahrt ausgelesen und zum Ausgabestring *fb* hinzugefügt:

```
public String konsolenausgabe_fahrtenbuch()
{
String fb="";    // Speichervariable für den „Ausgabestring"
fb = fb+"Datum           KM     dienstl. Fahrer" + "\n";
for (int i = 0; i < this.zaehleFahrten(); i++)
   {
     Fb = fb + this.holeFahrt(i).konsolenanzeige_Fahrt() + "\t" +
         this.holeFahrt(i).getMitarbeiter().konsolenanzeige_Mitarbeiter()+"\n";
   }
       return fb;
}
```

In der *Startklasse* wird dann die Methode zur Anzeige des Fahrtenbuchs im Konsolenfenster aufgerufen:

```
// Alle Fahrten von Dienstwagen 3
// Hersteller, kennzeichen, Datum, KM, Fahrer
System.out.println("Fahrten des Dienstwagen " + auto3.konsolenzeige_auto());
System.out.println(auto3.konsolenausgabe_fahrtenbuch());
```

Weitere Beispiele

Sachverhalt

Die Fuhrparkverwaltung der GeLa GmbH wünscht eine Auswertung der Fahrten des Dienstwagens mit dem Kennzeichen S – HE 1122 (Objekt *auto3*).
Für die gewünschten Auswertungen ist jeweils eine Methode in der Klasse *Dienstwagen* zu erstellen. Folgende Auswertungen für den Dienstwagen *auto3* sollen im Einzelnen vorgenommen werden können:

Auswertung	**Name der Methode**
- Summe der gefahrenen Kilometer	berechneSummeKM()
- Summe der gefahrenen Privat-KM	berechneSummeKM_Privat()
- Längste Fahrt mit dem Dienstwagen *auto3*	berechneKM_max()
- Kürzeste Fahrt mit dem Dienstwagen *auto3*	berechneKM_min()

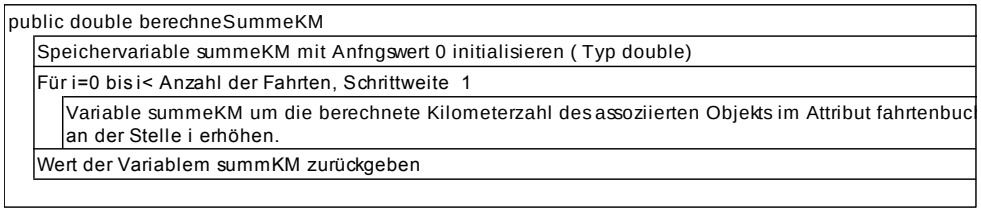
Aufgabenstellung:

Für den Dienstwagen mit dem Kennzeichen S – HE 1122 (Dienstwagenobjekt *auto3*) soll die Summe der gefahrenen Kilometer ermittelt und am Ende der Liste im Konsolenfenster angezeigt werden.

```
Fahrten des Dienstwagen Mercedes, S - HE 1122, rot
Datum           KM     dienstl. Fahrer
18.04.2013      500     Nein    Ralf Heart
21.04.2013      300     Nein    Anne Quick
22.04.2013      150     Nein    Uwe Fesch
23.04.2013      600     Ja      Fatma Tuk
Summe der gefahrenen Kilometer: 1550.0
```

➔ Methode berechneSummeKM(): Summe der gefahrenen Kilometer

Struktogramm:



Java Quellcode

```
public double berechneSummeKM()
{
    double summeKM=0;
    for (int i = 0; i < this.zaehleFahrten(); i++)
    {
       summeKM=summeKM+this.holeFahrt(i).berechneKM();
    }
    return summeKM;
}
```

Anwenden der Methode *double berechneSummeKM* in der *Startklasse*

```
System.out.println("Summe    " +"\t" + auto3.berechneSummeKM());
```

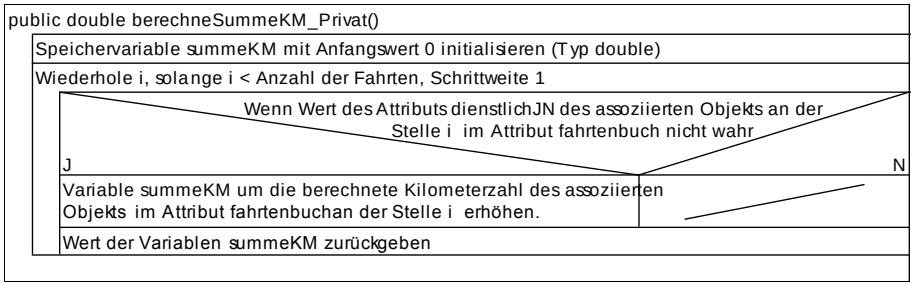
Aufgabenstellung:

Für den Dienstwagen mit dem Kennzeichen S – HE 1122 (Dienstwagenobjekt *auto3*) soll die Summe der privat gefahrenen Kilometer ermittelt und am Ende der Liste im Konsolenfenster angezeigt werden.

```
Fahrten des Dienstwagen Mercedes, S - HE 1122, rot
Datum           KM     dienstl. Fahrer
18.04.2013      500.0   Nein    Ralf Heart
21.04.2013      300.0   Nein    Anne Quick
22.04.2013      150.0   Nein    Uwe Fesch
23.04.2013      600.0   Ja      Fatma Tuk
Summe der gefahrenen Kilometer: 1550.0
davon privat: 950.0
```

➔ **Methode berechneSummeKM_Privat(): Summe der gefahrenen Privatkilometer**

Struktrogramm



Java Quellcode

```
public double berechneKM_Privat(){
   double summeKM_privat=0;
   for (int i = 0; i < this.zaehleFahrten(); i++)
   {
      if (! this.holeFahrt(i).isDienstlichJN())
      {
        summeKM_privat=summeKM_privat+this.holeFahrt(i).berechneKM();
      }
   }
   return summeKM_privat;
}
```

!	NOT
<	kleiner als
<=	kleiner oder gleich
>	größer als
>=	größer oder gleich
.equals(…)	Textvergleiche

Anwenden der Methode double *berechneSummeKM_Privat* () in der *Startklasse*

```
System.out.println("PrivatKM " +"\t" + auto3.berechneKM_Privat());
```

Aufgabenstellung:

Für den Dienstwagen mit dem Kennzeichen S – HE 1122 (Dienstwagenobjekt *auto3*) soll die Fahrt mit der höchsten Kilometerzahl ermittelt und am Ende der Liste im Konsolenfenster angezeigt werden.

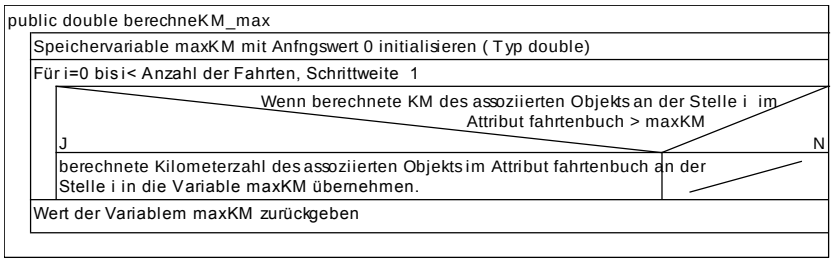
```
Fahrten des Dienstwagen Mercedes, S - HE 1122, rot
Datum           KM     dienstl. Fahrer
18.04.2013      500.0   Nein    Ralf Heart
21.04.2013      300.0   Nein    Anne Quick
22.04.2013      150.0   Nein    Uwe Fesch
23.04.2013      600.0   Ja      Fatma Tuk
Summe der gefahrenen Kilometer: 1550.0
davon privat: 950.0

höchste Kilometerzahl: 600.0
```

➔ Methode berechneSummeKM_max(): Längste Fahrt mit dem Dienstwagen *auto3*

Vorgehensweise:

Es wird eine Variable (*maxKM*) für die höchste Kilometerzahl und den Anfangswert 0 initialisiert. Dann werden für alle Fahrten die gefahrenen Kilometer der einzelnen Fahrten aus dem Fahrtenbuch ermittelt und mit dem Wert der Variablen *maxKM* verglichen. Wenn der errechnete Wert der Fahrt größer ist, als der bisherige Wert der Variablen *maxKM*, wird der errechnete Wert in die Variable übernommen. Wenn alle Fahrten bearbeitet sind, enthält die Variable *maxKM* die höchste berechnete Fahrt und kann als Ergebnis zurückgegeben werden.

Struktrogramm



Java Quellcode

```
public double berechneKM_max(){
// die Variable kmMax mit dem Anfangswert 0 initialisieren
double maxKM=0;
for (int i = 0; i < this.zaehleFahrten(); i++)
   {
      if (this.holeFahrt(i).berechneKM()> maxKM)
      {
         maxKM = this.holeFahrt(i).berechneKM();
      }
   return maxKM;
   }
}
```

Anwenden der Methode double *berechneSummeKM_max* in der *Startklasse*

```
System.out.println("höchste Kilometerzahl: " +"\t"+ auto3.berechneKM_max());
```

Aufgabenstellung:

Für den Dienstwagen mit dem Kennzeichen S – HE 1122 (Dienstwagenobjekt *auto3*) soll die Fahrt mit der niedrigsten Kilometerzahl ermittelt und am Ende der Liste im Konsolenfenster angezeigt werden.

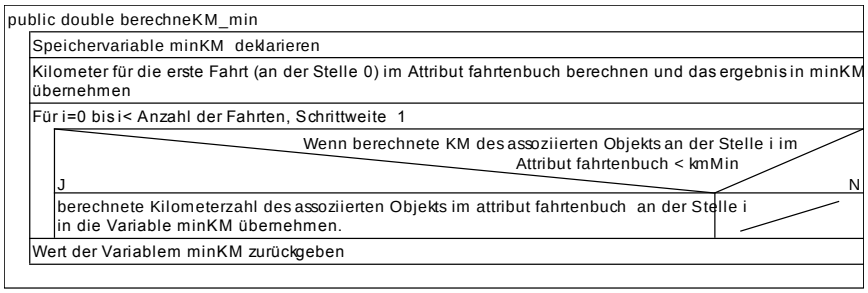
```
Fahrten des Dienstwagen Mercedes, S - HE 1122, rot
Datum           KM     dienstl. Fahrer
18.04.2013      500.0   Nein    Ralf Heart
21.04.2013      300.0   Nein    Anne Quick
22.04.2013      150.0   Nein    Uwe Fesch
23.04.2013      600.0   Ja      Fatma Tuk
Summe der gefahrenen Kilometer: 1550.0
davon privat: 950.0

höchste Kilometerzahl: 600.0
niedrigste Kilometerzahl: 150.0 <----------
```

➔ **Methode berechneSummeKM_min(): Kürzeste Fahrt mit dem Dienstwagen *auto3***

Vorgehensweise:

Es wird eine Variable *kmMin* für die niedrigste Kilometerzahl deklariert. In diese Variable werden die berechneten Kilometer der ersten Fahrt aus dem Fahrtenbuch (Stelle 0 im Attribut *fahrtenbuch*) übernommen. Anschließend werden für alle Fahrten die gefahrenen Kilometer der einzelnen Fahrten ermittelt und mit dem Wert der Variablen *kmMin* verglichen. Wenn der errechnete Wert der Fahrt kleiner ist, als der bisherige Wert der Variablen *kmMin*, wird der errechnete Wert in die Variable übernommen. Wenn alle Fahrten aus im Attribut *fahrtenbuch* bearbeitet sind, enthält die Variable *kmMin* den niedrigsten Wert und kann als Ergebnis zurückgegeben werden.

Struktrogramm



Java Quellcode

```
public double berechneKM_min(){
// km der ersten Fahrt als niedrigste Fahrt initialisieren
double minKM = this.holeFahrt(0).berechneKM();
for (int i = 0; i < this.zaehleFahrten(); i++)
      {
         if (this.holeFahrt(i).berechneKM()< minKM)
         {
               minKM= this.holeFahrt(i).berechneKM();
         }
      }
return minKM;
}
```

Anwenden der Methode double *berechneSummeKM_min* in der *Startklasse*

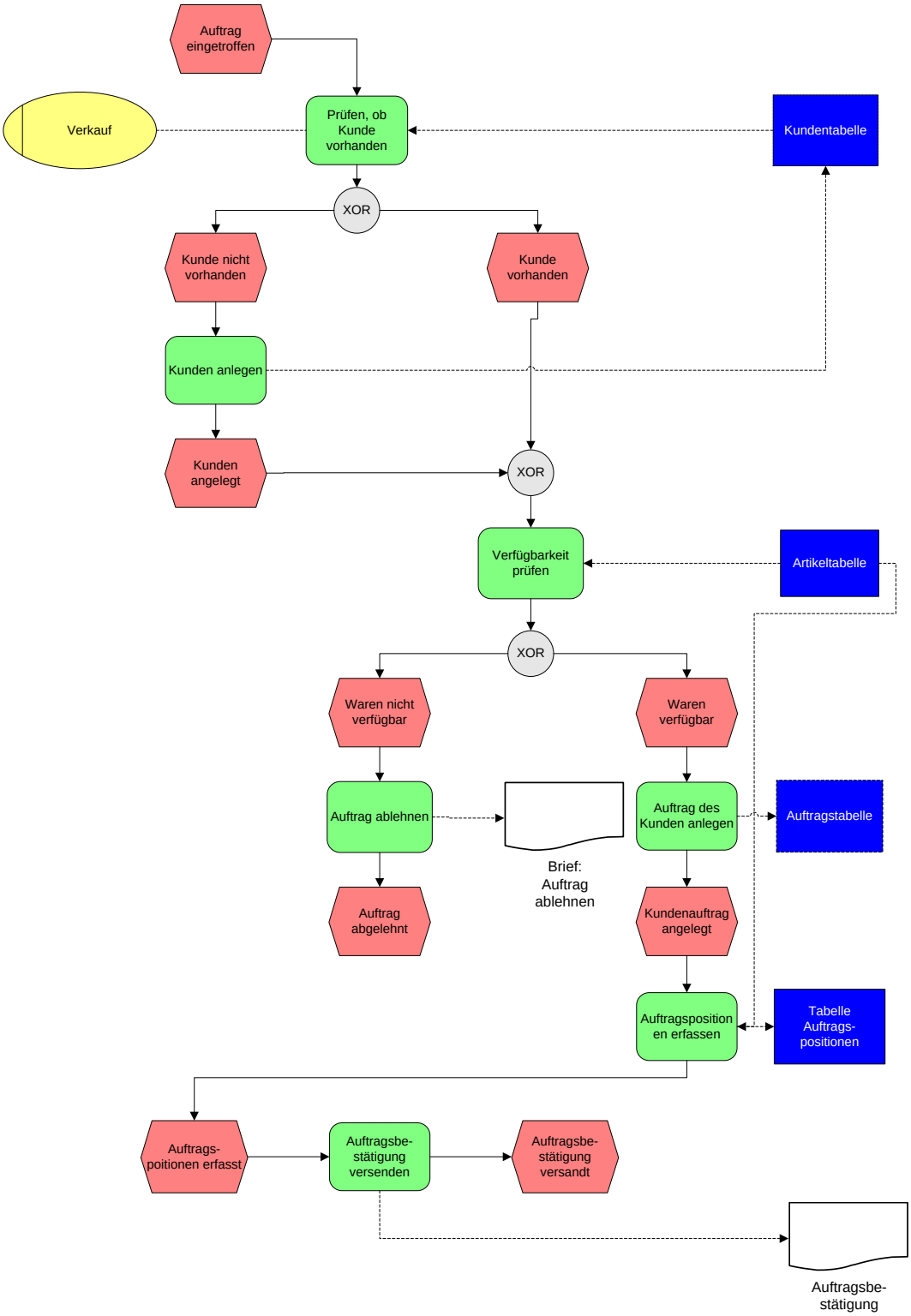
```
System.out.println("niedrigste Kilometerzahl: " + "\t" + auto3.berechneKM_min());
```

7 Kann- und Muss-Assoziationen

7.1 Sachverhalt/Problemstellung:

In der Auftragsabwicklung der GeLa GmbH wird zur Verwaltung der Kundenaufträge eine objektorientierte Anwendung eingesetzt.

Mit der nachfolgenden ereignisgesteuerten Prozesskette (EPK) wird der Ablauf des Geschäftsprozesses „Kundenauftrag“ beschrieben:

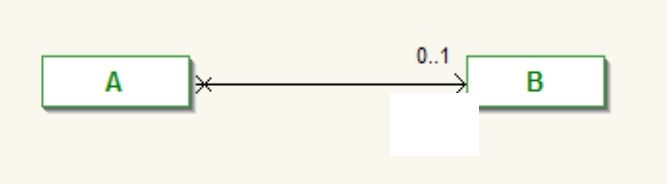
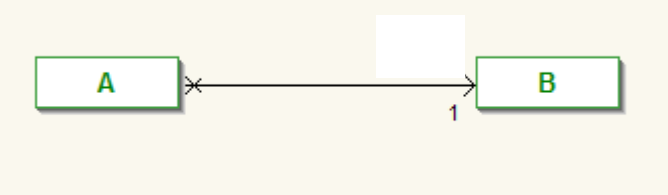
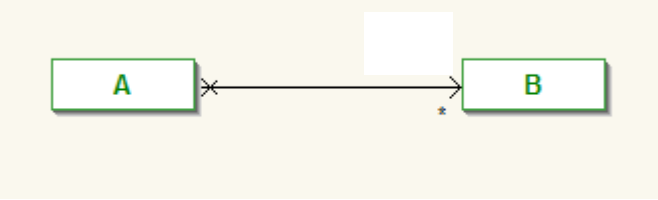
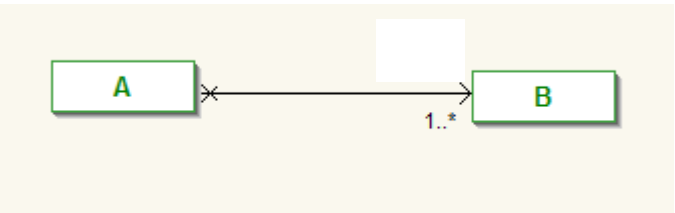


Erläuterung:

Wenn der Kundenauftrag eingetroffen ist, wird in der Verkaufsabteilung geprüft, ob die Kundendaten vorhanden sind. Wenn sie nicht vorhanden sind, werden sie neu angelegt. Wenn die Kundendaten angelegt sind oder schon vorhanden waren, kann die Verfügbarkeit der Artikel geprüft werden. Wenn nicht genügend Waren verfügbar sind, muss der Auftrag abgelehnt und dem Kunden die Ablehnung mitgeteilt werden. Sind die Waren verfügbar, kann der Auftrag angelegt und die Auftragspositionen können erfasst werden. Jede Auftragsposition bezieht sich auf einen Artikel. Wenn die Auftragspositionen erfasst sind, wird eine Auftragsbestätigung erzeugt und an den Kunden versandt.

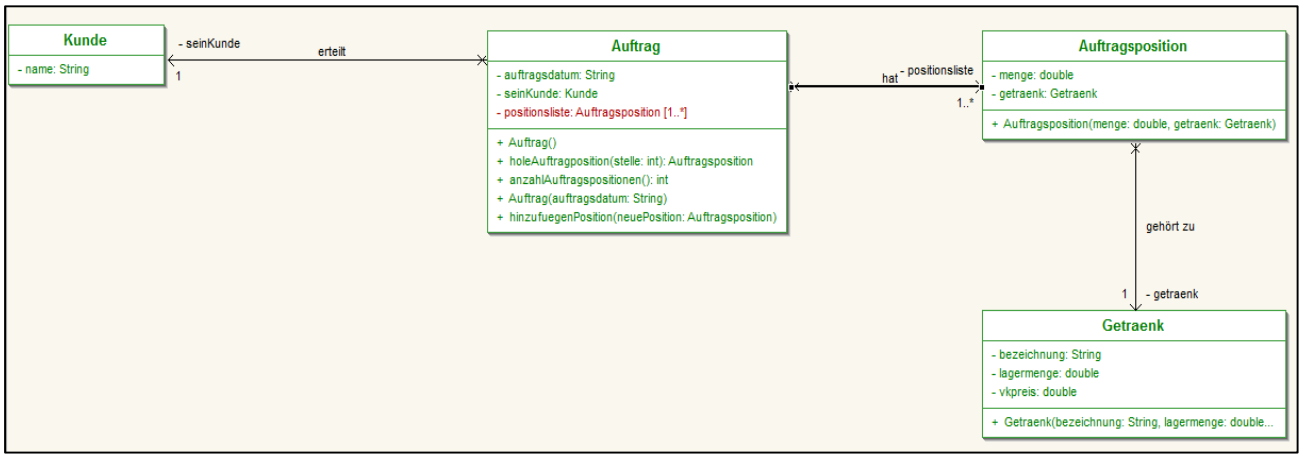
7.2 Übersicht über Kann- und Muss-Assoziationen

Übersicht über Kann- und Muss-Assoziationen

	Beschreibung
	**Kann-Assoziation mit Muliplizität 1** Zu einem Objekt der Klasse *A* gibt es kein oder ein assoziiertes Objekt der Klasse *B*.
	**Muss-Assoziation mit Muliplizität 1** Zu einem Objekt der Klasse *A* gibt es ein assoziiertes Objekt der Klasse *B*.
	**Kann-Assoziation mit Muliplizität N** Zu einem Objekt der Klasse *A* gibt es kein, ein oder mehrere assoziierte Objekte der Klasse *B*.
	**Muss-Assoziation mit Muliplizität N** Zu einem Objekt der Klasse *A* gibt es ein oder mehrere assoziierte Objekte der Klasse *B*.

7.3 Implementieren der Muss-Assoziationen

Aus dem Geschäftsprozess (siehe Seite 59) wurde das folgende (vereinfachte) Klassendiagramm entworfen. Das Modell ist aus didaktisch-methodischen Gründen reduziert. Nur die Klassen und deren Attribute, die zum Verständnis unerlässlich sind, werden verwendet.



Die Klasse *Auftrag* steht im Mittelpunkt. Wie in der ereignisgesteuerten Prozesskette beschrieben, kann ein Auftrag erst angelegt werden, wenn die Kundendaten vorliegen und er mindestens eine Position hat. Eine Auftragsposition muss immer zu einem Artikel führen.

Aus diesem Sachverhalt ergeben sich folgende Assoziationen:

- Von der Klasse *Auftrag* zur Klasse *Kunde* besteht somit eine unidirektionale **Muss-Assoziation** mit der **Multiplizität 1**. Die Assoziation wird mit dem Referenzattribut *seinKunde* realisiert.
- Von der Klasse *Auftrag* zur Klasse *Auftragsposition* besteht eine unidirektionale **Muss-Assoziation** mit der **Multiplizität N** und wird mit dem Referenzattribut *positionsliste* realisiert.
- Von der Klasse *Auftragsposition* zur Klasse *Getraenk* besteht eine **Muss-Assoziation** mit der **Multiplizität 1**. Sie wird mit dem Referenzattribut *getraenk* realisiert.

Bei einer **Muss-Assoziation** ist Multiplizität größer als Null.
Dies bedeutet, dass mindestens ein Objekt assoziiert sein muss. Eine Muss-Assoziation kann programmiertechnisch umgesetzt werden, indem

- der Konstruktor des Objekts ein bereits existierendes Objekt der assoziierten Klasse als Übergabewert einfordert oder
- mit dem Konstruktor des Objektes das Referenzattribut in der assoziierten Klasse erzeugt wird.

Problemstellung

In der Auftragsbearbeitung der GeLa GmbH geht ein Auftrag des Kunden Finke KG zur Lieferung von 30 Flaschen Apfelschorle, 20 Flaschen Mineralwasser Blauquelle natur und 10 Flaschen Himbeersaft ein.
Nachdem geprüft ist, dass der Kunde schon angelegt wurde und die Getränke verfügbar sind (siehe EPK auf Seite 59) soll der Auftrag mithilfe einer objektorientierten Software angelegt sowie die Auftragsbestätigung ausgedruckt und versandt werden.

Folgende Aufgabenstellungen sind zu erledigen:

1 Ein Javaprojekt *projektGeLa_Auftragsbearbeitung* und darin ein Paket *paketGela_Auftragsbearbeitung* sind anzulegen und das UML-Klassendiagramm mit den Klassen, Attributen und Assoziationen wie auf Seite 61 beschrieben, ist zu erstellen.

2 Eine *Startklasse* mit der *main*-Methode ist einzurichten und als Daten das Kundenobjekt *kunde1* „Finke Getränke KG“ sowie die folgenden Getränkeobjekte *getraenk1* bis *getraenk5* anzulegen. Zum Anlegen der Getränkeobjekte ist ein Konstruktor zu verwenden, der die Daten des jeweiligen Getränks als Übergabewert fordert.

Bezeichnung	Menge	Verkaufspreis
Cola light, 0,3l	500	1.10
Lemonlimonade, 0,3l	500	0.90
Apfelschorle, 0,5l	1000	1.00
Blauquelle nat., 0,7l	1000	0.45
Himbeersaft, 1,0l	300	1.75

3 Anpassen des Konstruktors der Klasse *Auftrag* und Anlegen eines Auftragsobjekts in der Startklasse.
Wie im Modell auf Seite 61 beschrieben, bezieht sich ein Auftrag auf einen Kunden und er wird nur dann angelegt, wenn er mindestens eine Auftragsposition hat.

3.1 Der Konstruktor der Klasse *Auftrag* ist so anzupassen, dass er neben dem Auftragsdatum

- das bereits existierendes Kundenobjekt *kunde1* der assoziierten Klasse *Kunde* als Übergabewert einfordert und
- ein Objekt *positionsliste* der Klasse *ArrayList* erzeugt wird, in das die Objekte der assoziierten Klasse *Auftragsposition* aufgenommen werden können.

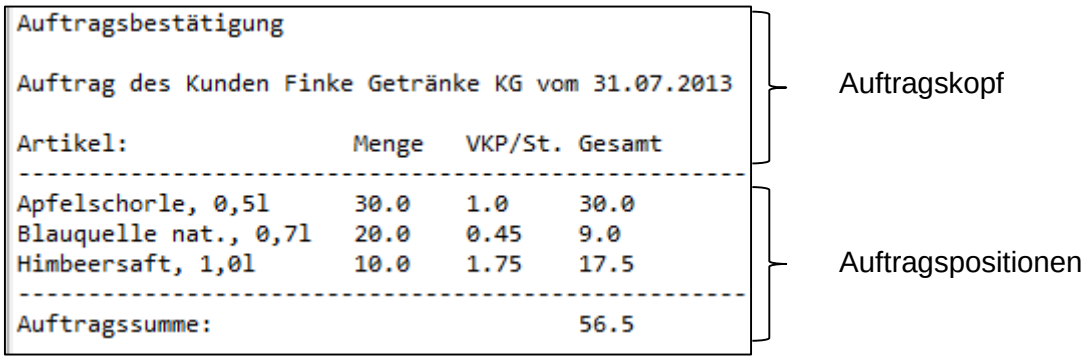
3.2 Anschließend ist der Auftrag des Kunden Finke Getränke KG als Objekt *auftrag1* in der *Startklasse* anzulegen.

4 Die Objekte *pos1*, *pos2* und *pos3* der Klasse *Auftragsposition* sind anzulegen und anschließend in das Attribut *positionsliste* aufzunehmen.
Wie im Modell auf Seite 61 beschrieben ist, muss sich eine Auftragsposition auf ein Getränk beziehen.

4.1 Der Konstruktor der Klasse *Auftragsposition* muss angepasst werden. Er muss neben der verkauften Menge das im Auftrag angegebene Objekt aus der Klasse *Getraenk* verlangen.

4.2 Anschließend werden in der *Startklasse* die Objekte der Klasse *Auftragsposition* erzeugt und in das Attribut *positionsliste* hinzugefügt.

5 Nachdem die Auftragsdaten erfasst sind, ist die nachfolgend dargestellte Auftragsbestätigung im Konsolenfenster anzuzeigen.

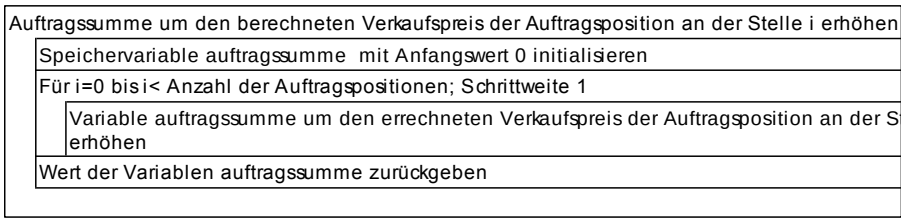


Dazu müssen folgende Aufgabenstellungen bearbeitet werden:

5.1 In der Klasse *Auftragsposition* ist die Methode *public double berechneVerkaufspreis()* zu erstellen. Die Methode liefert als Ergebnis das Produkt aus der verkauften Menge (Attribut *menge*) und dem Verkaufspreis (Attribut *vkpreis* des assoziierten Objekts der Klasse *Getraenk*).

5.2 In der Klasse *Auftrag* ist die Methode *public double berechneAuftragssumme()* zu erstellen. Die Methode liefert als Ergebnis die Summe der errechneten Verkaufspreise aller Positionen des Auftrages. Dazu wird der berechnete Gesamtverkaufspreis jeder Position ermittelt und aufsummiert.

Struktogramm der Methode *public double berechneAuftragssumme()*



5.3 Die Auftragsbestätigung (siehe Abb. oben) ist im Konsolenfenster darzustellen.

Die Konsolenanzeige soll so aufgebaut werden, dass die Daten der Auftragsbestätigung jeweils als Ausgabestring von Methoden aus den Fachklassen geliefert werden.

5.3.1 Die Methode *String konsolenanzeigeAuftragsposition()* der Klasse *Auftragsposition* liefert die Daten einer Auftragsposition.

5.3.2 Die Methode *String konsolenanzeigeAKop()* der Klasse *Auftrag* liefert die Daten des Auftragkopfes.

5.3.3 Die Methode *String konsolenanzeigeAPos()* der Klasse *Auftrag* liefert die Daten aller Auftragspositionen eines Auftrags.

Die beschriebene Vorgehensweise vereinfacht den Inhalt der *Startklasse*, insbesondere, wenn mehrere Aufträge erfasst werden sollen.

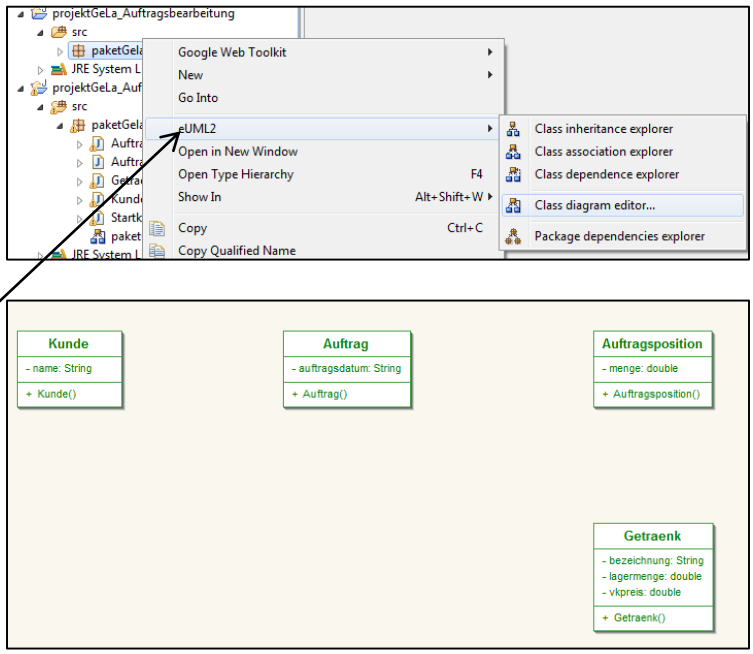
Lösungsvorschlag

Vorgehensweise

Zu 1

- Neues Javaprojekt (Menübefehl: **`File New Javaprojekt`** – *projektGeLa_Auftragsbearbeitung)* und
- im Projekt ein neues Paket (Menübefehl: **`File New Package`**– *paketGeLa_Auftragsbearbeitung)* anlegen.
- Anschließend das Paket anklicken und den Klassendiagrammeditor *eUML2* mithilfe des Kontextmenüs (rechte Maustaste) starten.
- Anschließend werden die Klassen mit den Attributen zunächst ohne die Assoziationen modelliert (siehe Seite 61).

Screenshot

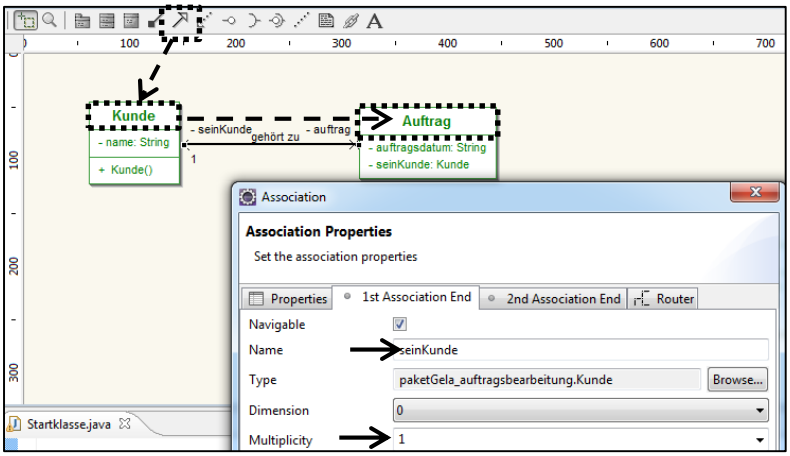


Stattdessen kann auch aus dem Ordner *Objektorientierte Systementwicklung: Assoziationen (Projekte - Quellcodes)* das Projekt *ProjektGela_auftragsabicklung* importiert werden.
(http://www.schule-bw.de/unterricht/faecher/informatik/material/softwareentwicklung/ooa-ood-oop/)

- Modellieren der Assoziationen
  - Klassen *Kunde* und *Auftrag*

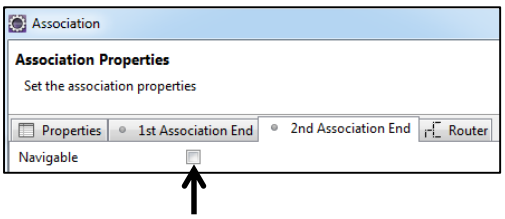
    Unidirektionale Assoziation mit Multiplizität 1. Ein Auftrag wird von einem Kunden erteilt. Die Assoziation wird mit dem Symbol aus der Symbolleiste modelliert. Sie erhält die Beschriftung „gehört zu“. Im ersten Assoziationsende (bei der Klasse *Auftrag*) wird das Referenzattribut *seinKunde* angegeben und die Multiplizität 1 ausgewählt.

erstes Assoziationsende:



Im zweiten Assoziationsende (Klasse *Kunde*) wird die Option *Navigable* deaktiviert.

zweites Assoziationsende:



Javacode des Referenzattributes *seinKunde* mit den dazugehörenden Zugriffsmethoden:

```
private Kunde seinKunde = new Kunde();

public Kunde getSeinKunde() {
      return seinKunde;
}

public void setSeinKunde(Kunde seinKunde) {
      this.seinKunde = seinKunde;
}
```

- Klassen *Auftrag* und *Auftragsposition*

  Unidirektionale Assoziation mit Multiplizität N. Ein Auftrag hat eine oder mehrere Auftragspositionen.

  Die Assoziation wird mit dem Symbol aus der Symbolleiste modelliert. Sie erhält die Beschriftung „*hat*“.

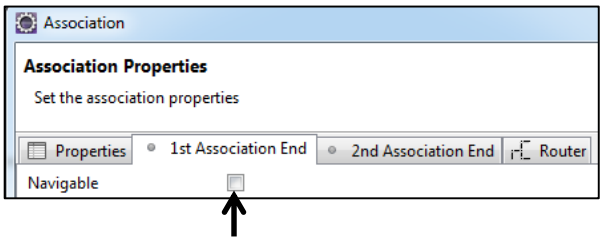
  Im ersten Assoziationsende (Klasse *Auftragsposition* wird die Option *Navigable* deaktiviert.

  Im zweiten Assoziationsende (Klasse *Auftrag*) wird das Referenzattribut *positionsliste* vom Typ der Klasse *ArrayList* mithilfe des [*Browse*]-Buttons ausgewählt.

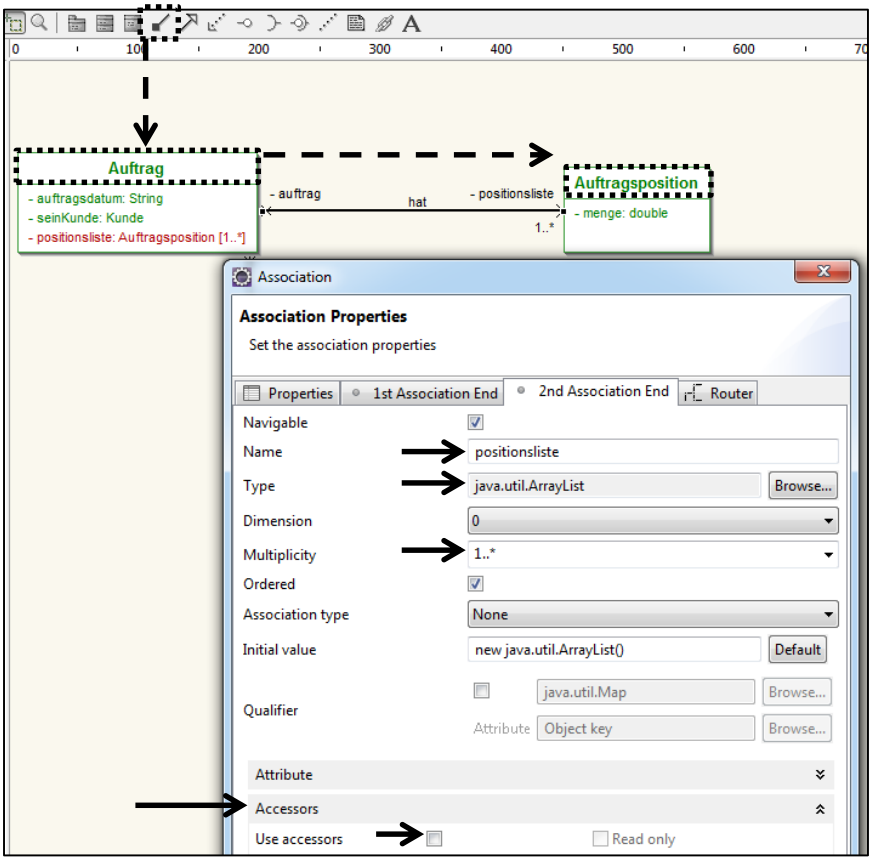
  Die Multiplizität ist 1..*.

  Um für das Attribut *positionsliste* gekapselte Zugriffsmethoden erstellen zu können, wird der Eintrag *Accessors* erweitert und die Option *Use accessors* deaktiviert.

erstes Assoziationsende:



zweites Assoziationsende:



Das Referenzattribut *positionsliste* wurde als Objekt der Klasse *ArrayList* initialisiert. Damit nur Objekte der Klasse *Auftragsposition* in die *positionsliste* aufgenommen werden können, wird die Klasse *ArrayList* typisiert.

Javacode der für das Referenzattribut

```
private ArrayList<Auftragsposition>
            positionsliste =  new java.util.
            ArrayList<Auftragsposition>();
```

Beim Modellieren der N-Assoziation wurde die Option *Use accessors* deaktiviert. Die entsprechenden Zugriffs-methoden müssen noch codiert werden.

```
// Zugriffsmethoden
// Hinzufügen einer Auftragsposition in die
// Positionsliste
public void hinzufuegenPosition
                    Auftragsposition neuePos)
{
    this.positionsliste.add(neuePos);
}
// Auslesen einer Auftragsposition aus einer
// bestimmten Stelle in der Positionsliste
public Auftragsposition
              holeAuftragsposition(int stelle)
{
    return positionsliste.get(stelle);
}
// Zählen der Anzahl der Objekte in der
// Positionsliste
public int anzahlPositionen()
{
    return positionsliste.size();
}
```

- Klassen *Auftragsposition* und *Getraenk*

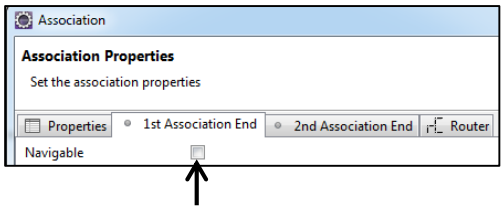
  Unidirektionale Assoziation mit Multiplizität 1. Eine Auftrags-position gehört zu einem Ge-tränk.

  Die Assoziation wird mit dem Symbol aus der Symbolleiste modelliert. Sie erhält die Beschriftung „gehört zu“.

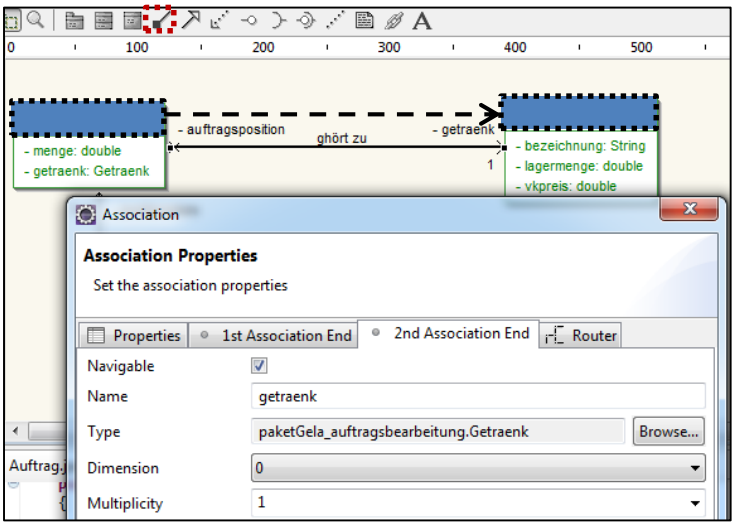
  Im ersten Assoziationsende (Klasse *Getraenk*) wird die Option *Navigable* deaktiviert.

  Im zweiten Assoziationsende (Klasse *Auftragposition*) wird das Referenzattribut *getraenk* vom Typ der Klasse *Getraenk* einge-tragen. Die Multiplizität ist 1.

erstes Assoziationsende:



zweites Assoziationsende:



Javacode des Referenzattri-butes *getraenk* mit den dazuge-hörenden Zugriffsmethoden:

```
private Getraenk getraenk = new
   paketGela_auftragsbearbeitung.Getraenk();

public Getraenk getGetraenk() {
    return getraenk;
}

public void setGetraenk(Getraenk getraenk) {
    this.getraenk = getraenk;
}
```

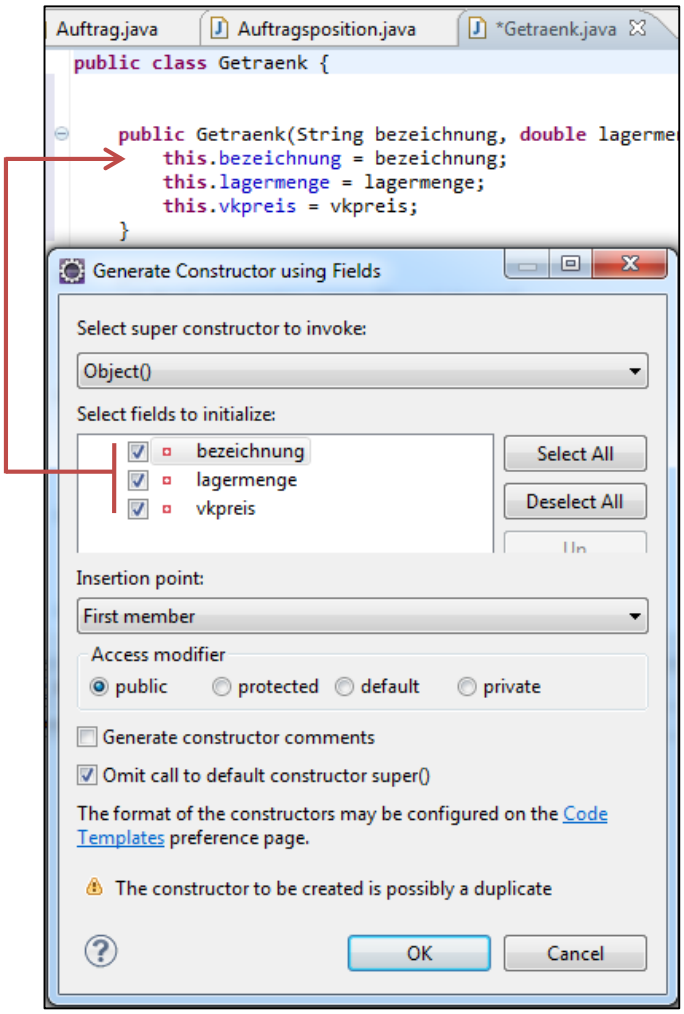
Zu 2

- Anpassen des Konstruktors der Klasse *Getraenk*.

  Der Konstruktor soll beim Erzeugen eines Objekts Werte für die Attribute *bezeichnung*, *lagermenge* und *vkpreis* anfordern.

  Zum Anpassen des Konstruktors mithilfe der Entwicklungsumgebung wird in den Quellcode der Klasse gewechselt.

  Mit dem Menü **`Source Generate Constructor using fields`** können dann die Attribute, die der Konstruktor anfordern soll, ausgewählt werden.



- Erzeugen der Objekte der Klasse *Getraenk* in der Startklasse und setzen der Attributwerte mithilfe des angepassten Konstruktors.

```
/* Objekte  der Klasse Getraenk erzeugen
   und Attributwerte mithilfe des
   Konstruktors setzen */

Getraenk getraenk1 = new Getraenk
       ("Cola light, 0,3l", 500, 1.10);
Getraenk getraenk2 = new Getraenk
       ("Lemonlimonade, 0,3l", 500, 0.90);
Getraenk getraenk3 = new Getraenk
       ("Apfelschorle, 0,5l", 1000, 1.00);
Getraenk getraenk4 = new Getraenk
       ("Blauquelle nat., 0,7l", 1000, 0.45);
Getraenk getraenk5 = new Getraenk
       ("Himbeersaft, 1,0l" , 300, 1.75);
```

Zu 3.1

Der angepasste Konstruktor der Klasse *Auftrag* benötigt zum Erzeugen eines Objekts neben dem Auftragsdatum das Objekt des Kunden, der den Auftrag erteilt hat **(1).**

Das Attribut *positionsliste* als Objekt der Klasse *ArrayList* wird außerhalb des Konstruktors deklariert und innerhalb des Konstruktors initialisiert **(2).**

Dadurch steht das Attribut den Zugriffsmethoden zur Verfügung.

Die früher beim Modellieren generierte Anweisung zum Deklarieren und Initialisieren des Attributes *positionsliste* kann gelöscht bzw. auskommentiert werden **(3).**

```
public class Auftrag {

  private String auftragsdatum;
  private ArrayList<Auftragsposition> positionsliste;

  public Auftrag(String auftragsdatum, Kunde seinKunde) {
     this.auftragsdatum = auftragsdatum;                    (2)
     this.seinKunde = seinKunde;              (1)
     this.positionsliste = new ArrayList<Auftragsposition>();
  }

  public String getAuftragsdatum() {
     return auftragsdatum;
  }

  public void setAuftragsdatum(String auftragsdatum) {
     this.auftragsdatum = auftragsdatum;
  }

  private Kunde seinKunde = new paketGela_auftragsbearbeitung.Kunde();

  public Kunde getSeinKunde() {
     return seinKunde;
  }

  public void setSeinKunde(Kunde seinKunde) {
     this.seinKunde = seinKunde;
  }

  // private ArrayList<Auftragsposition> positionsliste = new
  //                            java.util.ArrayList<Auftragsposition>();

  // Zugriffsmethoden
  // Hinzufügen einer Auftragsposition in die positionsliste
  public void hinzufuegenPosition(Auftragsposition neuePos){
     this.positionsliste.add(neuePos);
  }

  // Auslesen einer Auftragsposition aus einer bestimmten Stelle
  // in  der positionsliste
  public Auftragsposition holeAuftragsposition(int stelle) {
     return positionsliste.get(stelle);
  }

  // Zählen der Anzahl der Objekte in der positionsliste
  public int anzahlPositionen()      {
     return positionsliste.size();
  }
}
```

(3) →

Zu 3.2

Das Objekt *auftrag1* der Klasse *Auftrag* wird erzeugt. Der Konstruktor benötigt die Angabe des assoziierten Objekts *kunde1.*

```
// Objekt auftrag1 der Klasse Auftrag erzeugen
   Auftrag auftrag1 = new Auftrag("31.07.2013",kunde1);
```

Zu 4.1

Der angepasste Konstruktor der Klasse *Auftragsposition* benötigt zum Erzeugen eines Objekts neben der Auftragsmenge noch das Objekt des Getränks, das zu der Auftragsposition gehört.

```
// angepasster Konstruktor
   public Auftragsposition(double menge,
                                        Getraenk getraenk) {
      this.menge = menge;
      this.getraenk = getraenk;
   }
```

Zu 4.2

Die Objekte *pos1, pos2 und pos3* der Klasse *Auftragsposition* werden mithilfe des angepassten Konstruktors erzeugt. Zu jeder Auftragsposition muss ein Objekt der Klasse *Getraenk* angegeben werden (Mussassoziation).

```
// Objekte der Klasse Auftragsposition mit den dazuge-
// hörenden Objekten der Klasse Getraenk erzeugen
Auftragsposition pos1 = new Auftragsposition(30, getraenk3);
Auftragsposition pos2 = new Auftragsposition(20, getraenk4);
Auftragsposition pos3 = new Auftragsposition(10, getraenk5);
```

Dem Attribut *positionsliste* der Klasse *Auftrag* werden die erfassten Auftragspositionen (Objekte *pos1, pos2* und *pos3*) der Klasse *Auftragsposition*) hinzugefügt.

```
// Objekte der Klasse Auftragsposition dem Attibut
// positionsliste in der Klasse Auftrag hinzufügen
auftrag1.hinzufuegenPosition(pos1);
auftrag1.hinzufuegenPosition(pos2);
auftrag1.hinzufuegenPosition(pos3);
```

Zu 5.1

Die Methode *public double berechnePreis()* in der Klasse *Auftragsposition* liefert den Gesamtpreis der verkauften Menge (Attribut *menge* des aktuellen Objekts) und dem Verkaufspreis des Getränks (Attribut *vkpreis* des assoziierten Objekts der Klasse *Getraenk*).

```
// Methode zum berechnen des Gesamtpreises
public double berechnePreis()
{
   return this.menge * this.getGetraenk().getVkpreis();
}
```

Zu 5.2

In der Wiederholungsstruktur wird für jede Auftragsposition aus dem Attribut *positionsliste* der Verkaufspreis berechnet und zu der Variablen *auftragssumme* hinzuaddiert.

```
// Methode zum berechnen der Auftragssumme
public double berechneAuftragssumme() {
double auftragssumme=0;
for (int i = 0; i < this.anzahlPositionen(); i++)
   {
   auftragssumme = auftragssumme+
       this.holeAuftragsposition(i).berechnePreis();
   }
return auftragssumme;
}
```

Zu 5.3

Die Anzeige der Auftragsbestätigung im Konsolenfenster wird bei der nachfolgenden Vorgehensweise nicht in der Startklasse aufgebaut, sondern es werden in den Fachklassen für die auszugebenden Daten Methoden erstellt, welche die auszugebenden Daten jeweils als String liefern. In der *Startklasse* werden die auszugebenden Daten durch den Aufruf der Methode im Konsolenfenster angezeigt.
Diese Vorgehensweise ist bei umfangreicheren Ausgaben übersichtlicher und flexibler.

- Ausgabestring für die Positionen eines Auftrages in der Klasse *Auftragsposition*.
  Eine Auftrasposition enthält die Bezeichnung aus dem assoziierten Objekt der Klasse *Getraenk*, die verkaufte Menge und den Gesamtverkaufspreis dieser Postion. Dieser ergibt sich aus der Multiplikation von Menge und dem VK-Preis aus dem assoziierten Objekt der Klasse *Getraenk.*

```
// Auftragspositionsdaten als Ausgabestring
public String konsolenanzeigeAuftragsposition(){
  String tab = "\t";            // Tabulator
  return this.getGetraenk().getBezeichnung() +
              tab + this.getMenge()+
              tab + this.getGetraenk().getVkpreis() +
              tab + this.berechnePreis();
}
```

  Hinweis:

  Damit die Positionen spaltengerecht aufgelistet werden, werden einfache Tabulatoren verwendet. Mit der Anweisung **`String tab = „\t"`** wird in der Variablen *tab* ein „Steuerstring" abgelegt. Damit wird erreicht, dass die Ausgabe in der nächsten (vordefinierten) Tabulatorzone fortgesetzt wird.

- Ausgabestrings für die Anzeige der Auftragsbestätigung in der Klasse *Auftrag*.
  Für die Anzeige der Auftragsbestätigung werden zwei Methoden zur Rückgabe von Ausgabestrings erstellt (siehe Seite 63).

  - Ausgabestring *aKopf* für die Daten des Auftragskopfes.

    Der Ausgabestring *aKopf* wird schrittweise aufgebaut. Neben dem Auftragsdatum aus dem aktuellen Objekt wird der Name des Kunden aus dem assoziierten Objekt der Klasse *Kunde* in den Ausgabestring übernommen.

```
// Ausgabestring zur Anzeige des Auftrgskopfes
public String konsolenanzeigeAKopf()
{
  String aKopf = "";
  String n = "\n";
  aKopf=aKopf + "Auftragsbestätigung" + n;
  aKopf= aKopf + "Auftrag des Kunden "
                  + this.seinKunde.getName() + n;
  aKopf = aKopf + "Auftrag vom "
                  + this.getAuftragsdatum() + n;
  return aKopf;
}
```

Hinweis:

Mit der Anweisung `String n = "\n";` wird in der Variablen *n* ein „Steuerstring" abgelegt, der einen Zeilenumbruch auslöst.

- Ausgabestring *aPos* für die Auftragspositionen.

  Der Ausgabestring *aPos* wird schrittweise aufgebaut.

  Die Anzeige der Überschriftenzeile ist vor der Schleife positioniert **(1)**.

  In der Schleife wird für jede Auftragsposition die Methode `konsolenanzeigeAuftragsposition()` des assoziierten Objekts ausgeführt **(2)**.

  Vor und nach der Positionsliste wird jeweils ein Strich ausgegeben **(3)**.

  Die Auftragssumme (Methode **`berechneAuftragssumme()`**) wird für das aktuelle Objekt nach der Wiederholungsstruktur ausgeführt und in den Ausgabestring übernommen **(4)**.

  Mithilfe der Steuervariablen *tab* und *n* wird eine vereinfachte Formatierung erreicht.

```
// Ausgabestring zur Anzeige der Auftragspositionen
public String konsolenanzeigeAPos()     {
  String aPos = "";
  String tab = "\t";
  String n = "\n";
  String strich = "-----------------------------" + n;
  aPos = aPos + "Artikel:" + tab + tab
                  + "Menge" + tab
                  + "VKP/St." + tab
                  + "Gesamt" + n + strich; //( 1),(3)
  for (int i = 0; i < this.anzahlPositionen(); i++)
  {
     aPos = aPos + this.holeAuftragsposition(i).
          konsolenanzeigeAuftragsposition() + n; // (2)
  }
  aPos = aPos+strich;                              // (3)
  aPos = aPos + "Auftragssumme"
               + tab + tab + tab +    tab
               + this.berechneAuftragssumme();  // (4)
  return aPos;
}
```

- Ergänzung der Startklasse.

  Anzeigen der Auftragsbestätigung im Konsolenfenster.

```
// Anzeigen der Auftragsbestätitgung
// im Konsolenfenster
System.out.println(auftrag1.konsolenanzeigeAKopf());
System.out.println(auftrag1.konsolenanzeigeAPos());
```

-