TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Alexander van Renen (renen@in.tum.de)
http://db.in.tum.de/teaching/ss16/ei2/

Lösungen zu Blatt 1

Aufgabe 1: UML

Modellieren Sie das Münchner U-Bahnnetz in UML (Fahrzeuge, Linien, Haltestellen, ...). Überlegen Sie sich dafür sinnvolle Klassen und verbinden Sie diese mit Assoziationen. Anschließend skizzieren Sie ein Objektnetz für das Sie die Klassen Ihres Modells beispielhaft instanziieren.

Lösung 1

(a) Abbildung 1 zeigt ein mögliches Klassendiagramm:

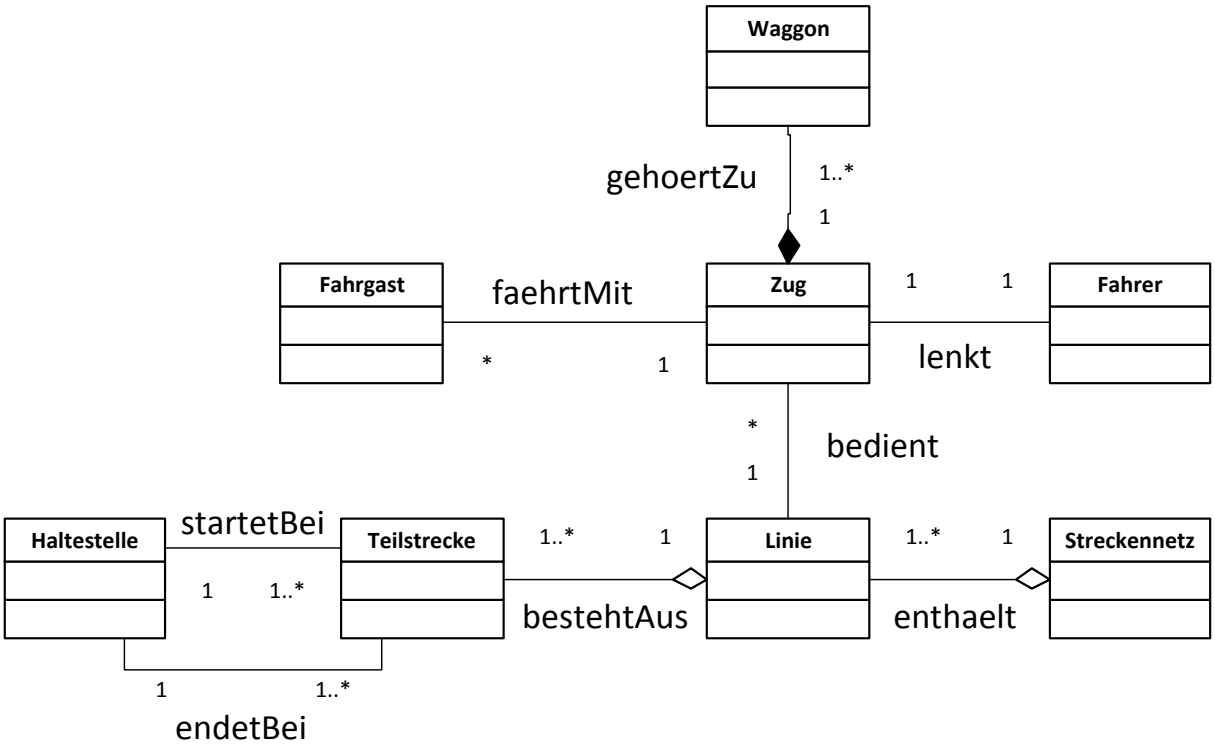

Abbildung 1: Das Münchner U-Bahnnetz modelliert als UML-Klassendiagramm

(b) Abbildung 2 zeigt ein dazu passendes Objektnetz:

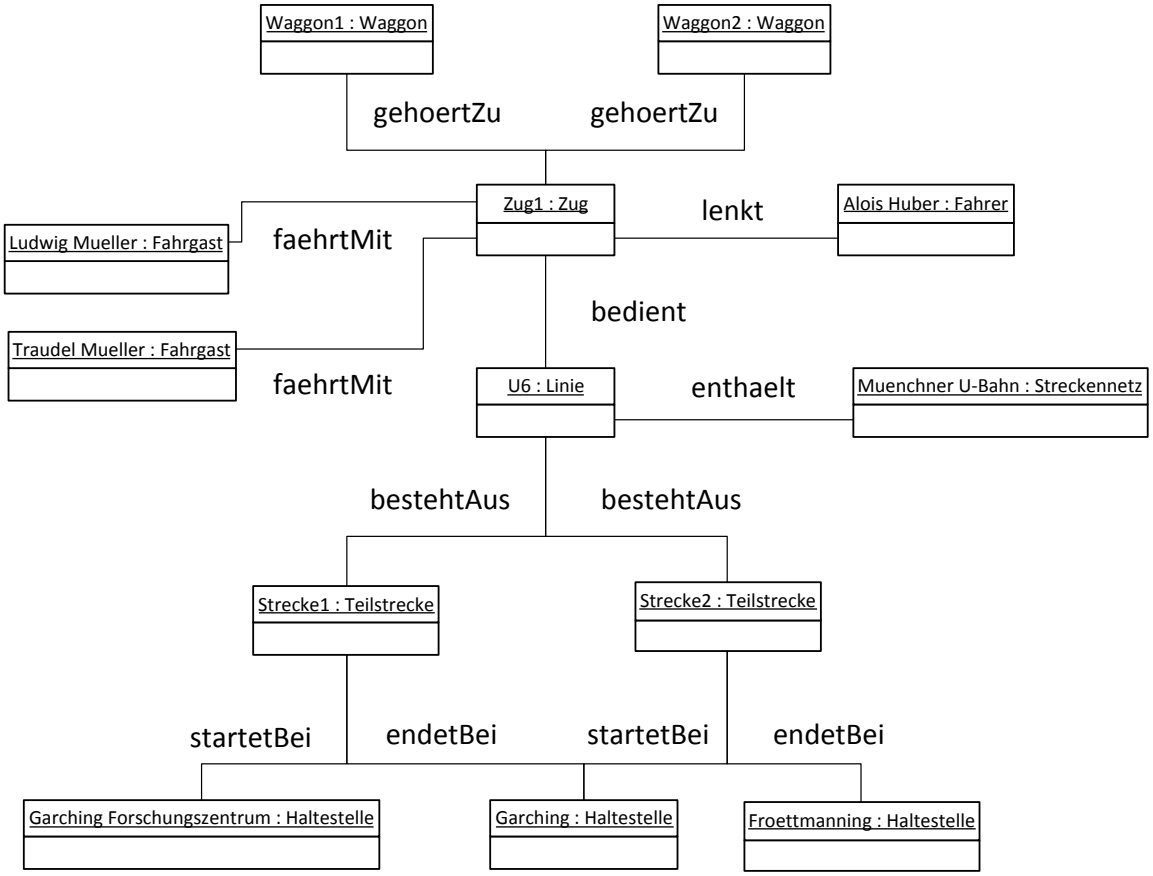

Abbildung 2: Ein Objektnetz als UML-Objektdiagramm, das die Klassen beispielhaft instanziiert

Aufgabe 2: Java

Setzen Sie das Polyeder-Beispiel aus Abbildung 3 in Java um. Bei den Methoden können Sie sich auf `skalieren()`, `verschieben()` und `rotieren()` der Klassen *Polyeder* und *Punkte* beschränken. Überlegen Sie sich dabei sinnvolle Argumente, da diese im UML-Modell fehlen.

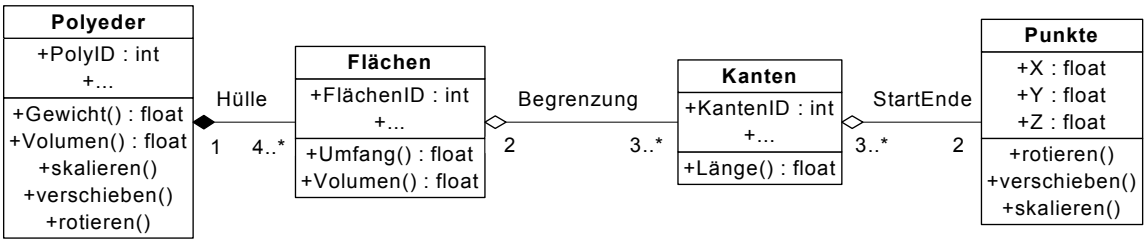

Abbildung 3: Modellierung eines Polyeders in UML

Lösung 2

Diese Aufgabe enthält eine versteckte Schwierigkeit, die aus der Verwendung gemeinsamer Unterobjekte resultiert. Kanten teilen sich im Modell Punkte und analog verwenden zwei angrenzende

Flächen die gleiche Kante. Verschiebt man jetzt einen Polyeder, indem man seine Flächen verschiebt – die wiederum die zugehörigen Kanten verschieben, so dass schlussendlich die Punkte bewegt werden – verschiebt man die Punkte gleich mehrfach. Dies ist der Modellierung geschuldet, die die Punkte zu mehreren Kanten und die Kanten zu je zwei Flächen zuordnet.

Eine mögliche Lösung für das Problem ist, dass man die Modellierung so abändert, dass sich Kanten keine Punkte mehr teilen und Flächen keine Kanten. Ein Punkt würde dann zu genau einer Kante und eine Kante zu genau einer Fläche gehören. Dies löst das ursprüngliche Problem, führt aber andererseits zu Redundanz, da nun mehrere Objekte für die gleichen räumlichen Punkte und Kanten erzeugt werden. Diese müssen nun synchron gehalten werden, da es sonst zu Inkonsistenzen kommt.

Die andere mögliche Lösung entspricht der Modellierung in der Angabe und vermeidet Redundanz. Statt in der `verschieben()`-Methode des Polyeders die `verschieben()`-Methode für jede einzelne Fläche aufzurufen, werden zuerst alle Punkte des Polyeders über die Methode `findePunkte()` gesammelt. Da die Punkte in einem `Set` gesammelt werden, erhält man dadurch jeden Punkt nur genau einmal (eine Menge kann kein Element mehrfach enthalten). Ruft man jetzt auf allen gesammelten Punkten `skalieren()`, `verschieben()` oder `rotieren()` auf, erhält man das erwartete Ergebnis.

```
import java.util.Set;
import java.util.HashSet;

class Polyeder {
  Flaeche[] huelle;

  public Polyeder(Flaeche[] huelle) {
    this.huelle = huelle;
  }

  public Set<Punkt> findePunkte() {
    HashSet<Punkt> ergebnis = new HashSet<Punkt>();
    for (Flaeche flaeche : huelle) {
      ergebnis.addAll(flaeche.findePunkte());
    }
    return ergebnis;
  }

  public void skalieren(double faktor) {
    Set<Punkt> punkte = findePunkte();
    for (Punkt punkt : punkte) {
      punkt.skalieren(faktor);
    }
  }

  public void verschieben(double deltaX, double deltaY, double deltaZ) {
    Set<Punkt> punkte = findePunkte();
    for (Punkt punkt : punkte) {
      punkt.verschieben(deltaX, deltaY, deltaZ);
    }
  }
```

```
  public void rotieren(double alpha, double n1, double n2, double n3) {
    Set<Punkt> punkte = findePunkte();
    for (Punkt punkt : punkte) {
      punkt.rotieren(alpha, n1, n2, n3);
    }
  }
}

class Flaeche {
  Kante[] begrenzung;

  public Flaeche(Kante[] begrenzung) {
    this.begrenzung = begrenzung;
  }

  public Set<Punkt> findePunkte() {
    HashSet<Punkt> ergebnis = new HashSet<Punkt>();
    for (Kante kante : begrenzung) {
      ergebnis.addAll(kante.findePunkte());
    }
    return ergebnis;
  }
}

class Kante {
  Punkt start;
  Punkt ende;

  public Kante(Punkt start, Punkt ende) {
    this.start = start;
    this.ende = ende;
  }

  public Set<Punkt> findePunkte() {
    HashSet<Punkt> ergebnis = new HashSet<Punkt>();
    ergebnis.add(start);
    ergebnis.add(ende);
    return ergebnis;
  }
}

class Punkt {
  double x;
  double y;
  double z;

  public Punkt(double x, double y, double z) {
```

```
    this.x = x;
    this.y = y;
    this.z = z;
  }

  public void skalieren(double faktor) {
    x *= faktor;
    y *= faktor;
    z *= faktor;
  }

  public void verschieben(double deltaX, double deltaY, double deltaZ) {
    x += deltaX;
    y += deltaY;
    z += deltaZ;
  }

  public void rotieren(double alpha, double n1, double n2, double n3) {
    x = x*(n1*n1*(1 - Math.cos(alpha)) + Math.cos(alpha))
      + y*(n1*n2*(1 - Math.cos(alpha)) - n3*Math.sin(alpha))
      + z*(n1*n3*(1 - Math.cos(alpha)) + n2*Math.sin(alpha));
    y = x*(n2*n1*(1 - Math.cos(alpha)) + n3*Math.sin(alpha))
      + y*(n2*n2*(1 - Math.cos(alpha)) + Math.cos(alpha))
      + z*(n2*n3*(1 - Math.cos(alpha)) - n1*Math.sin(alpha));
    z = x*(n3*n1*(1 - Math.cos(alpha)) - n2*Math.sin(alpha))
      + y*(n3*n2*(1 - Math.cos(alpha)) + n1*Math.sin(alpha))
      + z*(n3*n3*(1 - Math.cos(alpha)) + Math.cos(alpha));
  }
}
```