Softwaretechnik

Hausaufgabenblatt 7

Patrick Gustav Blaneck

Letzte Änderung: 22. November 2021

1. **Veranstaltungsverwaltungssystem**
   Gegeben sei folgende Beschreibung für ein Veranstaltungsverwaltungssystem:

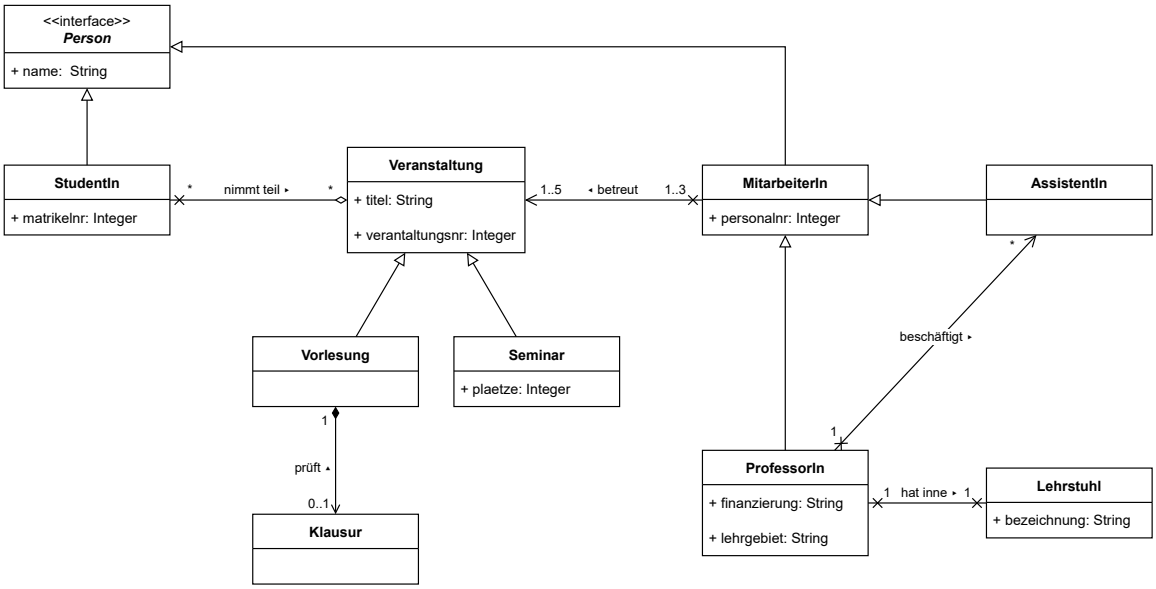
   Personen haben Zeichenketten als Name. Studenten sind Personen, erben also die Eigenschaften von Person, haben aber zusätzlich eine ganzzahlige Matrikelnummer und nehmen an beliebig vielen Veranstaltungen teil. Eine Veranstaltung hat potentiell beliebig viele Teilnehmer, wird aber von einem, zwei oder drei Mitarbeitern betreut. Eine Veranstaltung hat eine Veranstaltungsnummer und einen Titel. Seminare und Vorlesungen sind spezielle Veranstaltungen. Ein Seminar hat eine begrenzte Anzahl an Plätzen, für eine Vorlesung wird eine Klausur angeboten oder nicht. Mitarbeiter sind Personen und betreuen eine bis fünf Veranstaltungen und haben eine Personalnummer. Professoren und Assistenten sind Mitarbeiter. Assistenten sind bei genau einem Professor beschäftigt und haben eine bestimmt Finanzierung (Zeichenkette). Ein Professor hat ein Lehrgebiet (Zeichenkette), beschäftigt beliebig viele Assistenten und ist Inhaber von genau einem Lehrstuhl. Ein Lehrstuhl hat eine Bezeichnung und genau einen Professor als Inhaber.

   Erstellen Sie anhand der obigen Beschreibung ein *Klassendiagramm*. Ihr Diagramm sollte folgende Punkte beinhalten:

   - *Generalisierungsbeziehungen*,
   - *Assoziationen* mit Assoziationsnamen und Leserichtung,
   - *Multiplizitäten* sowie
   - *Attributnamen* und (sinnvolle) *–typen*.

   Finden Sie jeweils ein Beispiel, bei dem eine Aggregations- und eine Kompositionsbeziehung sinnvoll ist. Erläutern Sie kurz den Unterschied zwischen Aggretation und Komposition anhand des Bespiels.

Lösung:



Reddit=User raja_42 fasst den Unterschied zwischen Aggregation und Komposition gut zusammen:

> Association is a relationship between two entities. Kind of, associated with, in English.
>
> E.g. An Employee class will have a property which is a list of Projects.
>
> Composition is when a container entity has child entities which cannot survive on their own.
>
> E.g. A Shopping Cart class is composed of a list of Cart Items. When a Shopping Cart is deleted, the Cart Items cannot exist conceptually. So they are transient in nature.
>
> Any physical representation of them in database etc. will follow this principle. E.g. There will never be a Cart Item record in the DB without a Shopping Cart record id.
>
> Finally, an aggregation is like composition except the contained entities can exist on their own.
>
> E.g. When you are selecting a group of people in a classroom for a science project, you model few entities or classes.
>
> A Student class for every student in the class. A Professor entity. Etc.
>
> A ProjectGroup class is an aggregation which then has a list of students, a professor etc.
>
> Both are independent concepts. Dissolving a project group doesn't make the corresponding students and professors vanish from the system. They still exist.
>
> In physical form, they represent DB records that don't get deleted when the container record is deleted. This just means that they are independent entities that can live on their own.

2. **Kohäsion und Kopplung**

(a) Beschreiben Sie in eigenen Worten das Prinzip der Kohäsion und Kopplung in der Implementierung von Software-Projekten.

Lösung:

Als Reaktion auf einen mittelmäßigen Artikel über die Standarddefinitionen von Kohäsion und Kopplung, fasste ein mittlerweile gelöschter Reddit-User die beiden Begriffe gut zusammen:

> I feel that the author only succeeds in making it clear „cohesion“ and „coupling“ is the same thing, but with different emotional connotations. You know, sort of like militant groups are „terrorists“ if they go against your interests and „freedom fighters“ if they go in favor of your interests.
>
> This unfortunately happens, because people don't like shades of gray. Shades of gray make you look like you hesitate and you aren't sure what you're talking about.
>
> - If you say „Coupling can be both good and bad, it depends if you choose the right points where to split your codebase in two and introduce minimal interface between them . . . “ Boo! Not cool, I can't follow this advice. It's too philosophical and stuff.
> - But if you say „Coupling: bad; Cohesion: good!“ Yes, now I can go on forums and tell everyone who has coupling in their code that they're idiots, and refer to my own code's coupling as „cohesion“. I like this.
>
> If we need to find a distinction, we can argue that cohesion is actually a force, the „magnetic power“ between the elements in your codebase. The „desire“ for classes to be part of one unit. While coupling is a fact: these parts have now snapped together as a result of these forces, and they're together, they're coupled.
>
> But the „forces“ are still subjective. They're within our mental model of the problem, and not objective, not in the actual code. So we can't really say code is cohesive. Only our idea of it is cohesive or not. While coupling is objective and in the code.
>
> This means cohesive is an opinion, while coupling is a fact. And I'm not sure if we need a separate word to hide behind for our opinions. „I think this should remain coupled because it's cohesive“ means „I think this should remain coupled because I think it should remain coupled“, it's tautological.
>
> If we remove „cohesive“ from our dictionary, we can have a more substantial debate about why code should be coupled or decoupled at certain points.

(b) Warum sind hohe Kohäsion und lose Kopplung der niedrigen Kohäsion und enger Kopplung vorzuziehen?

Lösung:

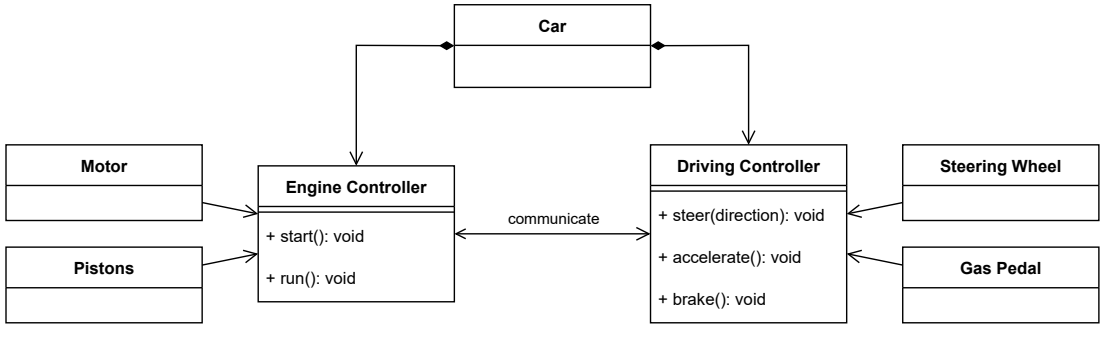
StackOverflow-Nutzer CardCastle Studio fasst es sehr gut zusammen:

> Let's take an example - We want to design a self-driving car.
>
> 1. We need the motor to run properly.
> 2. We need the car to drive on its own.
>
> All of the classes and functions in 1. starting the motor and making it run work great together, but do not help the car steer. So we place those classes behind an Engine Controller.
>
> All of the classes and functions in 2. work great to make the car steer, accelerate and brake. They do not help the car start or send gasoline to the pistons. So we place these classes behind its own Driving Controller.
>
> These controllers are used to communicate with all of the classes and functions that are available. The controllers then communicate only with each other. This means I can't call a function in the piston class from the gas pedal class to make the car go faster.
>
> The pedal class has to ask the Driving Controller to talk to the Engine Controller which then tells the piston class to go faster.
>
> This allows us programmers to be able to find issues and allows us to combine large programs without worrying. This is because the code was all working behind the controller.

(c) Geben Sie ein *eigenes* Beispiel in Form eines Klassendiagrammes für hohe Kohäsion und lose Kopplung an.

Lösung:

Wir greifen das Beispiel aus Teilaufgabe (b) auf:



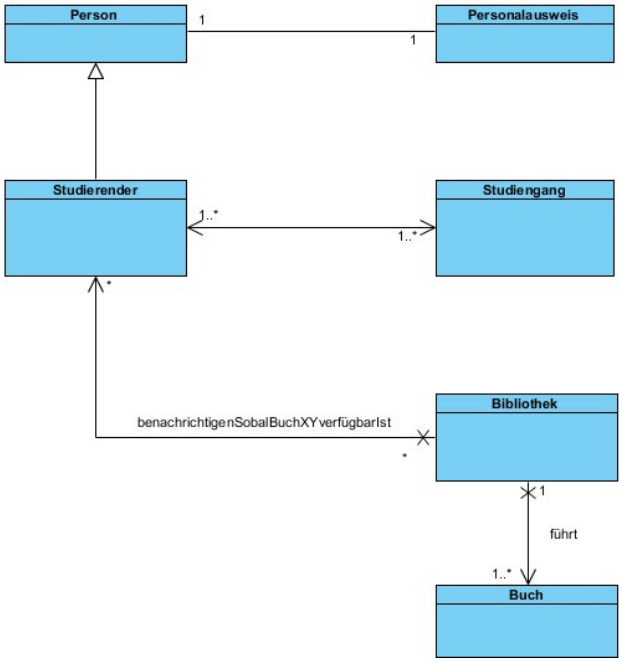
3. **Klassendiagramm Implizierungen**

Gegeben sei folgendes UML-Klassendiagramm mit ähnlichem Kontext wie aus der ersten Aufgabe.

Zusätzliche Annahmen:

- Jede Person hat nur einen Personalausweis. Wir vernachlässigen den Fakt, dass man diesen verlegen oder verlieren kann.
- Studierende können sich bei der Bibliothek auf Wartelisten für Bücher setzen lassen. Die Bibliothek benachrichtigt die Studierenden nach dem FIFO-Prinzip, wenn das Buch verfügbar ist. Das soll sich in der verwendeten Datenstruktur für die Assoziation zwischen Bibliothek und Studierender in der Bibliothek widerspiegeln.

Hinweis: Achten Sie darauf, dass die Multiplizitäten vom Code auch abgebildet werden! Zum Beispiel, dass eine Person keine Null-Referenz auf einen Personalausweis haben darf!



Tragen Sie in die vorgegebenen Java-Klassen, die Umsetzung der Assoziationen mit korrekter Multiplizität ein. An welchen Stellen ergibt es Sinn, die Assoziationen weiter über (unique, ordered) einzuschränken?

Lösung:

Siehe Anhang.

```java
import java.util.ArrayList;
import java.util.Collections;

public class Bibliothek {
    private String bezeichnung; // unique
    private ArrayList<Buch> buecher = new ArrayList<Buch>();

    public Bibliothek(String bezeichnung) {
        this.bezeichnung = bezeichnung;
    }

    public void add(Buch buch) {
        buecher.add(buch);
        Collections.sort(buecher);
    }

    public void ausleihen(Buch buch, Studierender studi) {
        buch.ausleihen(studi);
    }

    public void zurueckgeben(Buch buch) {
        buch.zurueckgeben();
        buch.benachrichtige();
    }

    public void addWartender(Buch buch, Studierender studi) {
        buch.addWartender(studi);
    }
}
```

Listing 1: `Bibliothek.java`

```java
import java.util.ArrayDeque;

public class Buch implements Comparable<Buch> {
    private String bezeichnung;
    private ArrayDeque<Studierender> warteliste = new ArrayDeque<>();
    private boolean blocked = false;

    public Buch(String bezeichnung) {
        this.bezeichnung = bezeichnung;
    }

    @Override
    public int compareTo(Buch buch) {
        return this.bezeichnung.compareTo(buch.bezeichnung);
    }

    public String getBezeichnung() {
        return bezeichnung;
    }

    public void ausleihen(Studierender ausleihender) {
        if (blocked)
            throw new ArithmeticException(
                "Buch gesperrt");
        if (warteliste.isEmpty()) {
            blocked = true;
            return;
        }
        if (warteliste.getFirst() == ausleihender)
            warteliste.pop();
        else
            throw new ArithmeticException(
                    "Buch vorgemerkt");
        blocked = true;
    }

    public void zurueckgeben() {
        blocked = false;
    }

    public void addWartender(Studierender wartender) {
        warteliste.offer(wartender);
    }

    public void benachrichtige() {
        if (warteliste.isEmpty())
            return;
        Studierender wartender = warteliste.getFirst();
        warteliste.pop();
        wartender.benachrichtige(this);
    }
}
```

Listing 2: Buch.java

```java
public abstract class Person {
    protected String name;
    protected Personalausweis personalausweis = new Personalausweis(); // unique

    public Person(String name, Personalausweis personalausweis) {
        this.name = name;
        this.personalausweis = personalausweis;
        personalausweis.setPerson(this);
    }

    public String getName() {
        return name;
    }
}
```

Listing 3: `Person.java`

```java
public class Personalausweis {
    private static int counter = 1;

    private int id; // unique
    private Person person;

    public Personalausweis() {
        // I solemnly swear that I'll add a Person
        this.id = getNewID();
    }

    private static int getNewID() {
        return counter++;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    public int getID() {
        return id;
    }
}
```

Listing 4: `Personalausweis.java`

```java
import java.util.ArrayList;
import java.util.Collections;

public class Studiengang {
    private String bezeichnung; // unique
    private ArrayList<Studierender> teilnehmende = new ArrayList<>(); // ordered

    public Studiengang(String bezeichnung) {
        this.bezeichnung = bezeichnung;
    }

    public void addTeilnehmender(Studierender teilnehmender) {
        teilnehmende.add(teilnehmender);
        // sort teilnehmende
        Collections.sort(teilnehmende);
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Studiengang: " + bezeichnung + "\n");
        for (Studierender studierender : teilnehmende) {
            sb.append("\t" + studierender + "\n");
        }
        return sb.toString();
    }
}
```

Listing 5: Studiengang.java

```java
public class Studierender extends Person implements Comparable<Studierender> {

    private String studienfach;
    private int semester;

    public Studierender(String name, String studienfach, int semester) {
        super(name, new Personalausweis());
        this.studienfach = studienfach;
        this.semester = semester;
    }

    @Override
    public int compareTo(Studierender studierender) {
        assert studierender != null;

        return this.studienfach.compareTo(studierender.studienfach);
    }

    public String getStudienfach() {
        return studienfach;
    }

    public void setStudienfach(String studienfach) {
        this.studienfach = studienfach;
    }

    public int getSemester() {
        return semester;
    }

    public void setSemester(int semester) {
        this.semester = semester;
    }

    @Override
    public String toString() {
        return "Studierender{" +
                "name='" + getName() + '\'' +
                ", studienfach='" + studienfach + '\'' +
                ", semester=" + semester +
                '}';
    }

    public void benachrichtige(Buch buch) {
        // whatever
    }
}
```

Listing 6: Studierender.java