Schehat Abdel Kader - 1630110

Detijon Lushaj – 1630149

Aufgabe 2.1: Interface versus Abstrakte Klasse

Wie Sie auf den Folien bestimmt gemerkt haben, werden bei den Entwurfsmustern die Oberklassen als Interface oder als abstrakte Klasse modelliert. Wann sollte ein Interface, wann eine abstrakte Klasse verwendet werden?

- In der Regel sollten Interfaces benutzt werden, weil Klassen Interfaces beliebig oft implementieren können. Abstrakte Klassen sollten nur verwendet werden, wenn Code geteilt wird oder eine Instanziierung notwendig ist wie z.B. beim decorator pattern für die decorators

Aufgabe 2.2: Open-Closed Principle (Offen/Geschlossen-Prinzip)

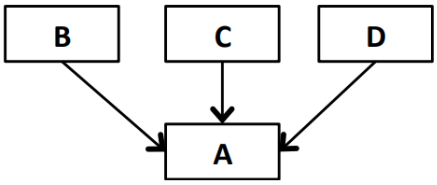
Ein weiteres wichtiges Entwurfsprinzip für objektorientierte Software ist das Open-Closed Principle (OCP):

> "Modules should be both open (for extension) and closed (for modification)."
> [Bertrand Meyer: *Object Oriented Software Construction*. Prentice Hall, 1988]
>
> "Komponenten sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Änderungen) sein."

Eine Komponente ist *verschlossen*, wenn sich deren Schnittstelle mit den angebotenen Operationen nicht verändert und auch deren Implementierung unverändert bleibt. Komponenten sind so zu entwerfen, dass sie sowohl stabil benutzbar (geschlossen) als auch offen für Erweiterungen sind.

a) Zum Beispiel realisiert das Observer-Muster (siehe SE1) das OCP, da neue Beobachter hinzugefügt werden können, ohne den Code des Modells zu ändern. Welche OO-Techniken ermöglichen das OCP?

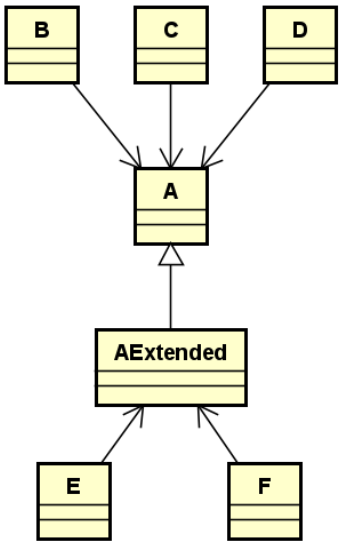
b) Erweitern Sie folgenden Entwurf: Die Klassen B, C und D verwenden die Klasse A. Bei der Weiterentwicklung des Systems kommen zwei neue Klassen E und F hinzu, die ebenfalls die Klasse A benutzen könnten, wenn Klasse A einige zusätzliche Funktionen anbieten würde.



a)

- Abstrakte Klassen und Interfaces, welche als Abstraktion für die genauen Klassen benutzt werden und als Bindeglied dienen
- Vererbung, um die Basisklasse ohne zu verändern zu erweitern
- Delegation, siehe facade pattern

b)



Aufgabe 2.3: Abstract Factory Pattern (Pflichtaufgabe)
In einem System sollen Parks verwaltet werden. Es gibt verschiedene Arten von Parks: Zum einen Klosterparks, in denen Kräuter angepflanzt werden. Zum anderen Stadtparks, in denen Rosen wachsen. Die Umrandung unterscheidet sich: Der Klosterpark ist von einer Steinmauer, der Stadtpark von einer Hecke umgeben. Der Boden ist auch unterschiedlich: Im Klosterpark liegen Steinplatten, im Stadtpark wächst Gras. Kloster- und Stadtpark sind zwei unterschiedliche Produktfamilien. Jede Produktfamilie hat gleiche Merkmale (Pflanzen, Umrandung, Boden), die aber unterschiedlich ausgestaltet sind.
Ein erster Lösungsansatz zum Anlegen, Pflegen und Verwalten (u.a. Mitarbeiter*innen einsetzen, Bewässerung steuern, Besucher*innen zählen etc.) von Parks könnte wie folgt aussehen:

```
public class Park {
    private enum Parktyp { Klosterpark, Stadtpark };
    private Parktyp park;
    private Boden boden;
    private Pflanze pflanze;
    private Umrandung umrandung;

    public void bodenLegen() {
        switch (park) {
            case Klosterpark:
                boden = new Steinplatte();
            break;

            case Stadtpark:
                boden = new Gras();
            break;
        }
    }

    public void pflaenzeSetzen() {
        switch (park) {
            case KlosterPark:
                pflanze = new Kraeuter();
            break;

            case Stadtpark:
                pflanze = new Rose();
            break;
        }
    }

    public void umranden() {
    // ... wie die obigen Methoden

//  die eigentlichen operativen Funktionen im Park
    public void zuweisenAufgabeMitarbeiter() {
    // ...
    }

    public void steuernBewaesserung() {
    // ...
    }

    public void steuernBeleuchtung() {
    // ...
    }

    public void zaehlenBesucher() {
    // ...
    }

    public void verwaltenFinanzen) {
    // ...
    }

    // ...
    // ...
```

Welche Nachteile hat obige Implementierung? Verbessern Sie den Entwurf der Klasse `Park` durch Anwendung des **Abstract Factory Pattern**. Geben Sie hierzu das resultierende Klassendiagramm und die Implementierung der Java-Klassen an.
Welche Änderungen sind notwendig, um mit "Japanischer-Park" eine neue Parkart hinzuzufügen?

- Nachteile:
  - Geringe Kohäsion
  - Nicht robust, schlecht erweiterbar und verletzt Offen-Geschlossen-Prinzip
- Neue japanische Parkart hinzufügen:
  - Die Ausprägungen erstellen und von den Boden, Pflanze und Umrandung implementieren lassen
  - Neuen Generator erstellen, der die AbstractParkGenerator Klassen implementiert
  - In der ParkGeneratorFactory die Art hinzufügen bei der Abfrage

```
public interface Boden {

}
```

```java
public interface Pflanze {

}
```

```java
public interface Umrandung {

}
```

```java
public class Steinplatte implements Boden {

}
```

```java
public class Gras implements Boden{

}
```

```java
public class Kraut implements Pflanze {

}
```

```java
public class Rose implements Pflanze{

}
```

```java
public class Steinmauer implements Umrandung {

}
```

```java
public class Hecke implements Umrandung{

}
```

```java
public class ParkGeneratorFactory {
    public AbstractParkGenerator createParkGenerator(String type) throws Exception {
        if (type.equals("Klosterpark")) {
            return new KlosterparkGenerator();
        } else if (type.equals("Stadtpark")) {
            return new StadtparkGenerator();
        }
        throw new Exception("no correct generator type given");
    }
}
```

```java
public interface  AbstractParkGenerator {
    public Boden createBoden();
    public Pflanze createPflanze();
    public Umrandung createUmrandung();
}
```

```java
public class KlosterparkGenerator implements AbstractParkGenerator {
    @Override public Boden createBoden() {
        return new Steinplatte();
    }

    @Override public Pflanze createPflanze() {
        return new Kraut();
    }

    @Override public Umrandung createUmrandung() {
        return new Steinmauer();
    }
}
```

```java
public class StadtparkGenerator implements AbstractParkGenerator {
    @Override public Boden createBoden() {
        return new Gras();
    }

    @Override public Pflanze createPflanze() {
        return new Rose();
    }

    @Override public Umrandung createUmrandung() {
        return new Hecke();
    }
}
```

```java
public class Park {
    private Boden boden;
    private Pflanze pflanze;
    private Umrandung umrandung;

    private ParkGeneratorFactory parkGeneratorFactory;
    private AbstractParkGenerator parkGenerator;

    public Park() {}

    public void init(String type) throws Exception {
        parkGenerator = parkGeneratorFactory.createParkGenerator(type);
        boden = parkGenerator.createBoden();
        pflanze = parkGenerator.createPflanze();
        umrandung = parkGenerator.createUmrandung();
    }
}
```

Aufgabe 2.4: Decorator Pattern (Pflichtaufgabe)

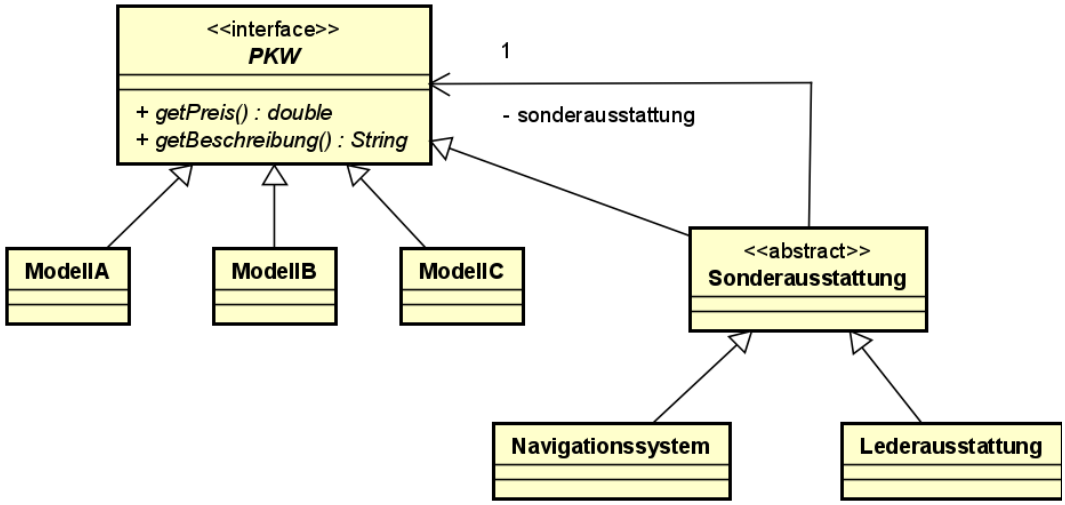
Bei einem PKW-Hersteller gibt es wenige Grundmodelle (z.B. Modell_A, Modell_B, Modell_C) mit einer Fülle von optionalen Sonderausstattungen (z.B. Navigationssystem, Lederausstattung, Klimaanlage). Der Gesamtpreis für ein bestelltes Fahrzeug errechnet sich aus dem festen Grundpreis des Grundmodells plus der Preise der gewählten Sonderausstattung.

(a) Modellieren Sie obigen Sachverhalt mit dem **Decorator Pattern**.

(b) Implementieren Sie das Modell in Java. Neben der Preisberechnung mit der Methode `getPreis()` soll es auch eine Methode `getBeschreibung()` geben, die das bestellte Fahrzeug mit seiner Ausstattung beschreibt, z.B.

```
Bestelltes Fahrzeug: Ein Fahrzeug des Modell_B und eine
Klimaanlage und eine Lederausstattung
```

a)



b)

```java
public abstract class PKW {
    double preis;

    public PKW(double preis) {
        this.preis = preis;
    }

    public double getPreis() {
        return preis;
    }

    public String getBeschreibung() {
        return "Bestellets Fahrzeug: Ein Fahrzeug des Modell " + this;
    }
}
```

```java
public class ModellA extends PKW {
    public ModellA(double preis) {
        super(preis);
    }

    @Override public String toString() {
        return "Modell A";
    }
}
```

```java
public abstract class Sonderausstattung extends PKW {
    private PKW pkw;

    public Sonderausstattung(PKW pkw, double preis) {
        super(preis);
        this.pkw = pkw;
    }

    @Override public double getPreis() {
        return pkw.getPreis() + preis;
    }

    @Override public String getBeschreibung() {
        return pkw.getBeschreibung() + " und eine " + this;
    }
}
```

```java
public class Klimaanlage extends Sonderausstattung {
    public Klimaanlage(PKW pkw, double preis) {
        super(pkw, preis);
    }

    @Override public String toString() {
        return "Klimaanlage";
    }
}
```

```java
public class Lederausstattung extends Sonderausstattung {
    public Lederausstattung(PKW pkw, double preis) {
        super(pkw, preis);
    }

    @Override public String toString() {
        return "Lederausstattung";
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        PKW pkw = new Lederausstattung(new Klimaanlage(new ModellA(10000), 2000), 1000);
        System.out.println(pkw.getPreis());
        System.out.println(pkw.getBeschreibung());
    }
}
```

```
13000.0
Bestelltes Fahrzeug: Ein Fahrzeug des Modell Modell A und eine Klimaanlage und eine Lederausstattung
```