SE 2 Übung 2

Furkan Aydin, M1630039

Falk-Niklas Heinrich, M1630123

Aufgabe 2.1: Interface versus Abstrakte Klasse

Interfaces sollten dann genutzt werden, wenn eine Schnittstelle benötigt wird, die ein Verhalten eines Objekts beschreibt, es aber keine oder nur wenige Gemeinsamkeiten zwischen den tatsächlichen Implementierungen gibt. Eine Klasse kann zudem mehrere Interfaces implementieren, aber nur von einer (abstrakten) Klasse erben.

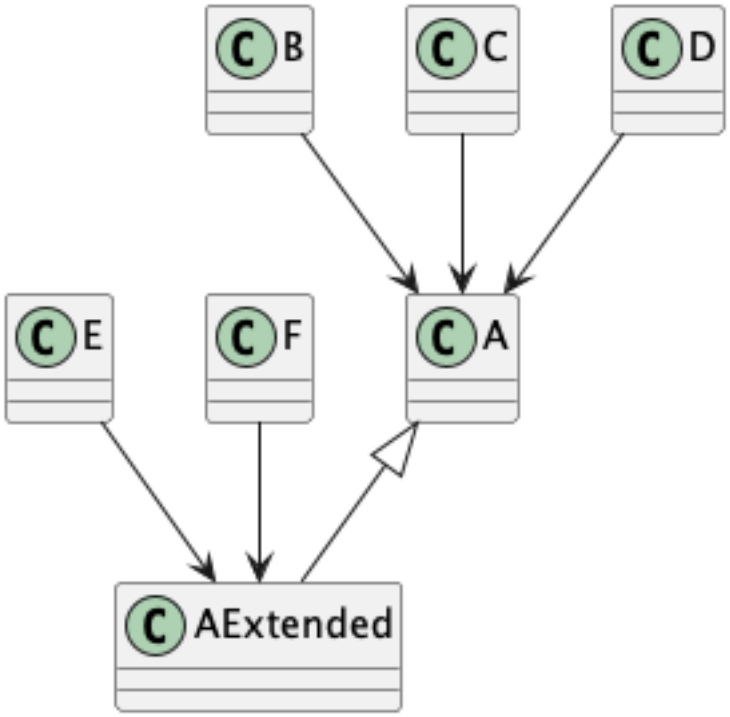
Abstrakte Klassen können hingegen besonders geeignet sein, falls die Elternkomponente gemeinsam genutzte Funktionen für alle Implementierungen enthält. Auch können abstrakte Klasse bereits für alle Kinder bestimmte Interfaces implementieren und so Code-Duplikation vermeiden.

Aufgabe 2.2: Open-Closed Principle

a)

Vererbung bzw. Interfaces (Polymorphie) werden in OO-Sprachen für die Erweiterbarkeit genutzt, private Methoden und Felder und die Klassen an sich sorgen für die Kapselung.

b)

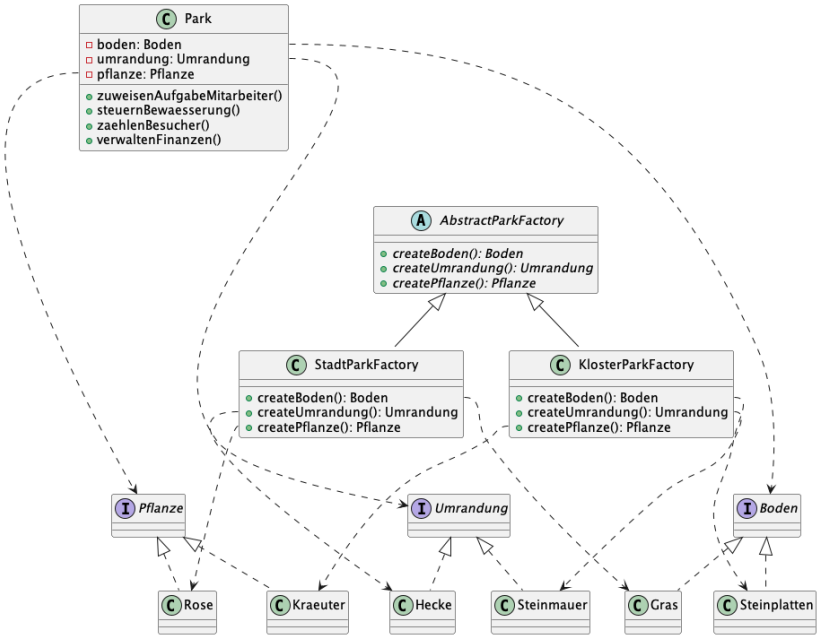


Aufgabe 2.3: Abstract Factory Pattern (Pflichtaufgabe)

Nachteile

- In einer Klasse werden unterschiedliche Funktionen umgesetzt.
- Schlecht wartbar
- Erweiterung komplex
- Fehleranfällige Switch/Case-Statements
- Basiseigenschaften eines Parkobjekts änderbar

Abstract-Factory



UML

Implementierung

```
1
2  public class A2_3 {
3
4      public static void main(String[] args) {
5          KlosterParkFactory kpFactory = new KlosterParkFactory();
6          Park klosterPark = new Park();
7          klosterPark.setBoden(kpFactory.createBoden());
8          klosterPark.setPflanze(kpFactory.createPflanze());
9          klosterPark.setUmrandung(kpFactory.createUmrandung());
10
11         StadtParkFactory spFactory = new StadtParkFactory();
12         Park stadtPark = new Park();
13         stadtPark.setBoden(spFactory.createBoden());
14         stadtPark.setPflanze(spFactory.createPflanze());
15         stadtPark.setUmrandung(spFactory.createUmrandung());
16
17         assert !klosterPark.getBoden().getClass().equals(
18             stadtPark.getBoden().getClass());
19         assert !klosterPark.getPflanze().getClass().equals(
20             stadtPark.getPflanze().getClass());
```

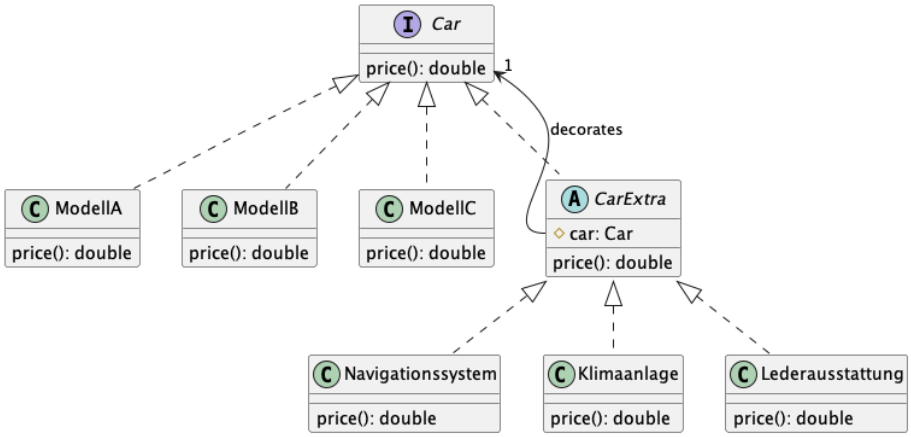
```
21          assert !klosterPark.getUmrandung().getClass().equals(
22              stadtPark.getUmrandung().getClass());
23      }
24  }
25
26  interface Boden {
27  }
28
29  class Steinplatten implements Boden {}
30  class Gras implements Boden {}
31
32  interface Umrandung {
33  }
34
35  class Steinmauer implements Umrandung {}
36  class Hecke implements Umrandung {}
37
38  interface Pflanze {
39  }
40
41  class Kraeuter implements Pflanze {}
42  class Rose implements Pflanze {}
43
44  class Park {
45      private Boden boden;
46      private Umrandung umrandung;
47      private Pflanze pflanze;
48
49      Park() {
50      }
51
52      void zuweisenAufgabeMitarbeiter() {
53      }
54
55      void steuernBewaesserung() {
56      }
57
58      void zaehlenBesucher() {
59      }
60
61      void verwaltenFinanzen() {
62      }
63
64      Boden getBoden() {
65          return boden;
66      }
```

```
67
68      void setBoden(Boden boden) {
69          this.boden = boden;
70      }
71
72      Umrandung getUmrandung() {
73          return umrandung;
74      }
75
76      void setUmrandung(Umrandung umrandung) {
77          this.umrandung = umrandung;
78      }
79
80      Pflanze getPflanze() {
81          return pflanze;
82      }
83
84      void setPflanze(Pflanze pflanze) {
85          this.pflanze = pflanze;
86      }
87  }
88
89  abstract class AbstractParkFactory {
90      abstract Boden createBoden();
91      abstract Umrandung createUmrandung();
92      abstract Pflanze createPflanze();
93  }
94
95  class KlosterParkFactory extends AbstractParkFactory {
96
97      @Override
98      Boden createBoden() {
99          return new Steinplatten();
100     }
101
102     @Override
103     Umrandung createUmrandung() {
104         return new Steinmauer();
105     }
106
107     @Override
108     Pflanze createPflanze() {
109         return new Kraeuter();
110     }
111
112 }
```

```java

class StadtParkFactory extends AbstractParkFactory {

    @Override
    Boden createBoden() {
        return new Gras();
    }

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

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

}
```

Aufgabe 2.4: Decorator Pattern (Pflichtaufgabe)

UML



Implementierung

```java
public class A2_4 {

    public static void main(String[] args) {
        Car modellA = new ModellA();
        assert modellA.price() == 10_000;
```

```
6           modellA = new Navigationssystem(modellA);
7           assert modellA.price() == 11_000;
8           modellA = new Klimaanlage(modellA);
9           assert modellA.price() == 11_500;
10
11          Car modellB = new Lederausstattung(new ModellB());
12          assert modellB.price() == 20_800;
13
14          Car modellC = new Navigationssystem(new Klimaanlage(new
    ↪  Lederausstattung(new ModellC())));
15          assert modellC.price() == 30_000 + 1_000 + 800 + 500;
16      }
17
18  }
19
20  interface Car {
21      double price();
22  }
23
24  class ModellA implements Car {
25
26      @Override
27      public double price() {
28          return 10_000;
29      }
30
31  }
32
33  class ModellB implements Car {
34
35      @Override
36      public double price() {
37          return 20_000;
38      }
39
40  }
41
42  class ModellC implements Car {
43
44      @Override
45      public double price() {
46          return 30_000;
47      }
48
49  }
50
```

```java
abstract class CarExtra implements Car {
    protected final Car car;

    protected CarExtra(Car car) {
        this.car = car;
    }

    public Car getCar() {
        return car;
    }
}

class Navigationssystem extends CarExtra {

    protected Navigationssystem(Car car) {
        super(car);
    }

    @Override
    public double price() {
        return this.getCar().price() + 1_000;
    }

}

class Klimaanlage extends CarExtra {

    protected Klimaanlage(Car car) {
        super(car);
    }

    @Override
    public double price() {
        return this.getCar().price() + 500;
    }

}

class Lederausstattung extends CarExtra {

    protected Lederausstattung(Car car) {
        super(car);
    }

    @Override
    public double price() {
```

```
97              return this.getCar().price() + 800;
98          }
99
100     }
```