Software Engineering

Exercise sheet 06: Design pattern

Software Engineering (winter semester 24/25)

Submission: **Fri. 20.12.2024, 10:00 a.m.**
Meeting: Fri. 13.12.2024

Please note the instructions for submission in Moodle (see "Submission sheet 06")

Task 1: Application of a design pattern (20 points)

Consider the following code. A *collection* has already been developed (see `CollectionImpl` in Listing 1) that allows you to iterate over its elements. A simple user interface (UI) has also been developed (see `CollectionUI` in Listing 2) that displays the elements of a collection, but this UI uses the `CollectionIterator` interface (see Listing 3) to iterate over the elements of a collection.

The aim now is for the `CollectionUI` to also be able to process instances of `CollectionImpl` without having to change the `CollectionUI` and `CollectionImpl` classes and the `CollectionIterator` interface (see Listing 4).

Listing 1: `CollectionImpl`

```
1  package en. hu. se. task 1 ;
2
3  public class Collection Impl {
4
5      private String [] elements ;
6      private int index = 0;
7
8      public Collection Impl ( String [] elements ) {
9          this . elements = elements ;
10     }
11
12     public boolean has More Elements () {
13       return this . index < this . elements . length ;
14     }
15
16     public String next Element () {
17        String elem = this . elements [ this . index ];
18          this . index ++;
19          return elem ;
20     }
21
22 }
```

Listing 2: `CollectionUI`

```
1  package de. hu. se. task 1 ;
2
3  public class Collection UI {
4
5      public void show Collection ( Collection Iterator collection          {
       Iterator )
6          while ( collection Iterator . has Next ()) {
7            System . out . println ( collection Iterator . next ());
8          }
9      }
10
11 }
```

Listing 3: `CollectionIterator`

```
1  package de. hu. se. task 1 ;
2
3  public interface Collection Iterator {
4
5      public boolean has Next ();
6
7      public String next ();
8
9  }
```

Listing 4: `Client`

```
1  package en. hu. se. task 1 ;
2
3  public class Client {
4
5      public static void main ( String [] args ) {
6          Collection Impl c = new Collection Impl ( new String []{ " elem1 ", " elem2 " });
7
8          // We can directly use the class Collection Impl to list all elements
9          // but we want to use the UI!
10
11         // UI is expecting a Collection Iterator .
12         // TODO How can we use a Collection Impl object in the UI? Collection
13         Iterator collection Iterator = null ;
14
15         Collection UI ui = new Collection UI ();
16         ui. show Collection ( collection
17         Iterator );
18     }
}
```

(a) *(8 points)* This task is to be completed in Moodle exercise06.

(b) *(12 points)* Extend the code with this design pattern so that the `CollectionUI` can display the elements of `CollectionImpl` instances. The predefined classes (`CollectionImpl` and `CollectionUI`) and the interface (`CollectionIterator`) must **not** be changed here. Complete or provide a class with a Main method that executes your solution as an example.

Task 2: Application of a design pattern (40 points)

The cash register system of a café is to be further developed. So far, the classes for beverages have already been developed, in particular for coffee (see `Coffee` in Listing 7) and decaffeinated coffee (see `Decaf` in Listing 8), both of which inherit from the abstract class `Beverage` (see Listing 9). In particular, the `costs()` method, which calculates the price of a beverage, is abstract and must be defined and implemented for each type of beverage.

For each of the two types of beverages (`Coffee` and `Decaf`), a guest can order any number and combination of the following additions, each of which costs extra:

Milk (Price: 0.15)
Soy milk (0.20)
Sugar (0.10)
Cream (0.30)

The aim is to extend the existing code using a design pattern, in particular the properties of a beverage tank are to be *dynamically and flexibly* supplemented by the selected additions to the beverage tank. Accordingly, the behavior for price calculation and the description of a tank should be supplemented by the selected additions. It is not an option to list the different possibilities and combinations of beverages and additives based on their number (e.g. `CoffeeWithMilk`, `CoffeeWithTwoMilk`, etc.).

Listing 7: `Coffee`

```
1  package de. hu. se. on Issue 2 ;
2
3  public class Coffee  extends Beverage {
4
5      public Coffee () {
6          description  = " House Blend Coffee ";
7      }
```

```
8
9       @ Override
10      public double costs () {
11          return 1. 10;
12      }
13  }
```

Listing 8: Decaf

```
1   package de. hu. se. task 2 ;
2
3   public class Decaf extends Beverage {
4
5       public Decaf () {
6           description = " Decaf Coffee ";
7       }
8
9       @ Override
10      public double costs () {
11          return 1. 20;
12      }
13  }
```

Listing 9: Beverage

```
1   package de. hu. se. task 2 ;
2
3   public abstract class Beverage {
4       String description = " Unknown Beverage ";
5
6       public String get Description () {
7           return description ;
8       }
9
10      public abstract double costs ();
11  }
```

(a) *(16 points)* This task is to be completed in Moodle exercise 06.

(b) *(24 points)* Extend the code with this design pattern so that the predefined classes `Coffee` and `Decaf` can be dynamically and flexibly supplemented with properties that any selection of attributes in account. Add or provide a class with a main method that executes your solution as an example.

Task 3: Application of a design pattern (40 points)

Consider the following code that describes a shopping cart for groceries (see `ShoppingCart` in Listing 16). Groceries (see `ShoppingItem` in Listing 17) are either basic groceries (see `Food` in Listing 18) or luxury groceries (see `LuxuryFood` in Listing 19).

These classes can be used to create an object structure that describes a filled shopping cart. The aim is to add behavior to this object structure without adding new attributes and references to these classes. Specifically, the VAT is to be calculated for a filled shopping cart. As the software is used in supermarkets in Germany (19% on luxury foodstuffs and 7% on basic foodstuffs) and Italy (22% on luxury foodstuffs and 10% on basic foodstuffs), the VAT calculation should be supported for both countries. In the future, it should also be easy to support other countries. Therefore, the implementation of the behavior for calculating the VAT should take place outside the specified classes or object structure as far as possible.

Listing 16: ShoppingCart

```
1   package de. hu. se. task 3 ;
2   import java . util . Array List ;
3   import java . util . Collection ;
4   import java . util . Collections ;
5
6   public class Shopping Cart {
7
8       private Collection < Shopping Item > items = new Array List < Shopping Item >();
9
```

```
10      public void add Item ( Shopping Item item ) {
11          items . add ( item );
12      }
13
14      public Collection < Shopping Item > get Items (){
15          return Collections . unmodifiable Collection (
            items );
16      }
17  }
```

Listing 17: ShoppingItem

```
1   package de. hu. se. task 3 ;
2
3   public abstract class Shopping Item {
4
5       private double price ;
6
7       public Shopping Item ( double price ) {
8           this . price = price ;
9       }
10
11      public double get Price () {
12          return price ;
13      }
14
15      public void set Price ( double price ) {
16          this . price = price ;
17      }
18  }
```

Listing 18: Food

```
1   package de. hu. se. task 3 ;
2
3   public class Food extends Shopping Item {
4
5       public Food ( double price ) {
6           super ( price );
7       }
8   }
```

Listing 19: LuxuryFood

```
1   package de. hu. se. task 3 ;
2
3   public class Luxury Food extends Shopping Item {
4
5       public Luxury Food ( double price ) {
6           super ( price );
7       }
8   }
```

(a) *(16 points)* This task is to be completed in Moodle exercise 06.

(b) *(24 points)* Extend the code to include this design pattern without adding new attributes and references to these classes, while the calculation of the VAT should take place outside the specified classes or object structure as far as possible. Complete or provide a class with a main method that executes your solution as an example.