Lab - Design Patterns

This document defines the lab for the "Java Advanced" course @ Software University.

1. Singleton

We are going to create a simple program to demonstrate what Singleton is used for. So, to start, let's create a single interface:

```
import java.util.Map;

public interface SingletonContainer {

    int getPopulation(Map<String, Integer> capitals, String name);
}
```

After that, we have to create a class to implement the SingletonContainer interface. We are going to call it SingletonDataContainer:

```
import java.util.HashMap;
import java.util.Map;

public class SingletonDataContainer implements SingletonContainer {
    private static SingletonDataContainer instance;
    private Map<String,Integer> capitals;


    private SingletonDataContainer() {
        this.capitals = new HashMap<>();
        System.out.println("Initializing singleton object");
    }


    public int getPopulation(Map<String, Integer> capitals, String name) { return capitals.get(name); }
```

So, we have a Map on which we store the capital names and their population. As we can see, in our constructor we are initializing it. And that is all good. Now we are ready to use this class in any consumer by simply instantiating it. But is this really what we need to do, to instantiate the class which uses data that never changes (in this particular project. The Population of the cities is changing daily). Of course not, so using a Singleton pattern would be very useful here. Let's implement it.

After we hid the constructor from the consumer classes by making it private, we will need to create a single instance of our class and expose it like this:

```
public int getPopulation(Map<String, Integer> capitals, String name) { return capitals.get(name); }

public static SingletonDataContainer getInstance() {
    if (instance != null){
        return instance;
    }
    instance = new SingletonDataContainer();
    return instance;
}
```

At this point, we can call the Instance property as many times as we want, but our object is going to be instantiated only once and shared for every other call.

Let's check if our program works:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> capitals = new HashMap<>();

        capitals.put("Sofia", 120000);
        capitals.put("Varna", 90000);

        SingletonDataContainer instance = SingletonDataContainer.getInstance();
        System.out.println(instance.getPopulation(capitals, name: "Sofia"));
        SingletonDataContainer instance1 = SingletonDataContainer.getInstance();
        System.out.println(instance1.getPopulation(capitals, name: "Varna"));
    }
}
```

The expected output should be something like this:

```
Initializing singleton object
120000
90000
```

2. Façade

Now we will take a look at a Façade example implementation.

We will start by creating a class to work with:

```
public class Car {
    private String type;
    private String color;
    private int numberOfDoors;
    private String city;
    private String address;

    public String getType() { return this.type; }

    public void setType(String type) { this.type = type; }

    public String getColor() { return this.color; }

    public void setColor(String color) { this.color = color; }

    public int getNumberOfDoors() { return this.numberOfDoors; }

    public void setNumberOfDoors(int numberOfDoors) { this.numberOfDoors = numberOfDoors; }

    public String getCity() { return this.city; }

    public void setCity(String city) { this.city = city; }

    public String getAddress() { return this.address; }

    public void setAddress(String address) { this.address = address; }

    @Override
    public String toString() {
        return String.format("CarType: %s, " +
                "Color: %s, Number of doors: %d, " +
                "Manufactured in %s, at address: %s", this.type, this.color, this.numberOfDoors, this.getCity(),this.getAddress());
    }
}
```

We have the info part and the address part of our object, so we are going to use two builders to create this whole object.

We need a façade, let's create one:

```
public class CarBuilderFacade {
    protected Car car;

    public CarBuilderFacade() { this.car = new Car(); }

    public Car build() {
        return this.car;
    }
    public CarInfoBuilder info(){
        return new CarInfoBuilder(this.car);
    }
    public CarAddressBuilder built(){
        return new CarAddressBuilder(car);
    }
}
```

We instantiate the **Car** object, which we want to build, and expose it through the Build method.

What we need now is to create concrete builders. So, let's start with the **CarInfoBuilder** which needs to inherit from the facade class:

```
public class CarInfoBuilder extends CarBuilderFacade{

    public CarInfoBuilder(Car car) { this.car = car; }
    public CarInfoBuilder withType(String type){
        car.setType(type);
        return this;
    }
    public CarInfoBuilder withColor(String color){
        car.setColor(color);
        return this;
    }
    public CarInfoBuilder withNumberOfDoors(int number){
        car.setNumberOfDoors(number);
        return this;
    }
}
```

We receive, through the constructor, an object we want to build and use the fluent interface for building purposes.

Let's do the same for the **CarAddresBuilder** class:

```
public class CarAddressBuilder extends CarBuilderFacade{
    public CarAddressBuilder(Car car){
        this.car = car;
    }
    public CarAddressBuilder inCity(String city){
        car.setCity(city);
        return this;
    }
    public CarAddressBuilder atAddress(String address){
        car.setAddress(address);
        return this;
    }
}
```

At this moment we have both builder classes, but we can't start building our object yet because we haven't exposed our builders inside the facade class. Well, let's do that:

```
public class CarBuilderFacade {
    protected Car car;

    public CarBuilderFacade() { this.car = new Car(); }

    public Car build() {
        return this.car;
    }
    public CarInfoBuilder info(){
        return new CarInfoBuilder(this.car);
    }
    public CarAddressBuilder built(){
        return new CarAddressBuilder(car);
    }
}
```

That's it, we can start building our object:

```
public class Main {
    public static void main(String[] args) {
        Car car = new CarBuilderFacade()
                .info() CarInfoBuilder
                    .withType("BMW") CarInfoBuilder
                    .withColor("Black") CarInfoBuilder
                    .withNumberOfDoors(5) CarInfoBuilder
                .built() CarAddressBuilder
                    .inCity("Leipzig") CarAddressBuilder
                    .atAddress("Some address 254") CarAddressBuilder
                .build();
        System.out.println(car);
    }
}
```

And the output should be:

```
CarType: BMW, Color: Black, Number of doors: 5, Manufactured in Leipzig, at address: Some address 254

Process finished with exit code 0
```

3. Façade 2

Here is another example of the Façade Pattern.

https://howtodoinjava.com/design-patterns/structural/facade-design-pattern/

4. Command Pattern

The Command design pattern consists of the Invoker class, Command class/interface, Concrete command classes, and the Receiver class. Having that in mind, in our example, we are going to follow the same design structure.

So, what we are going to do is write a simple app in which we are going to modify the price of the product that will implement the Command design pattern.

That being said, let's start with the **`Product`** receiver class, which should contain the base business logic in our app:

```
public class Product {
    private String name;
    private int price;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getPrice() { return price; }

    public void setPrice(int price) { this.price = price; }

    public Product(String name, int price){
        this.name = name;
        this.price = price;
    }

    public void increasePrice(int amount) { this.setPrice(this.getPrice()+amount); }

    public void decreasePrice(int amount) { this.setPrice(this.getPrice()-amount); }

    @Override
    public String toString() {
        return String.format("Current price for the %s product is %d$.%n",this.name, this.price);
    }
}
```

Now the Client class can instantiate the **Product** class and execute the required actions. But the Command design pattern states that we shouldn't use receiver classes directly. Instead, we should extract all the request details into a special class - Command. Let's do that.

The first thing we are going to do is to add the **Command** interface:

```
public interface Command {

    void executeAction();
}
```

Finally, let's add the **DecreaseProductPriceCommand** and **IncreaseProductPriceCommand** class:

```
public class DecreaseProductPriceCommand implements Command{
    private final Product product;
    private final int amount;

    public DecreaseProductPriceCommand(Product product, int amount){
        this.product = product;
        this.amount = amount;
    }

    public String executeAction() {
        this.product.decreasePrice(this.amount);
        return String.format("The price for the %s has been decreased by %d$.%n",
                this.product.getName(), this.amount);
    }
}
```

```
public class IncreaseProductPriceCommand implements Command {
    private final Product product;
    private final int amount;

    public IncreaseProductPriceCommand(Product product, int amount){
        this.product = product;
        this.amount = amount;
    }

    public String executeAction() {
        this.product.increasePrice(this.amount);
        return String.format("The price for the %s has been increased by %d$.%n",
                this.product.getName(), this.amount);
    }
}
```

To continue, let's add the **`ModifyPrice`** class, which will act as Invoker:

```
import ...

public class ModifyPrice {
    private final List<Command> commands;
    private Command command;
    public ModifyPrice() { this.commands = new ArrayList<Command>(); }

    public void setCommand(Command command) { this.command = command; }

    public void invoke(){
        this.commands.add(this.command);
        this.command.executeAction();
    }
}
```

This class can work with any command that implements the **`Command`** interface and store all the operations as well.

Now, we can start working with the client part:

```
public class Main {
    public static void main(String[] args) {
        ModifyPrice modifyPrice = new ModifyPrice();
        Product product = new Product("Phone",500);

        execute(modifyPrice, new IncreaseProductPriceCommand(product, 100));
        execute(modifyPrice, new IncreaseProductPriceCommand(product, 50));
        execute(modifyPrice, new DecreaseProductPriceCommand(product, 25));

        System.out.println(product);
    }
    private static void execute(ModifyPrice modifyPrice, Command productCommand){
        modifyPrice.setCommand(productCommand);
        modifyPrice.invoke();
    }
}
```

The output should be like this:

```
The price for the Phone has been increased by 100$.
The price for the Phone has been increased by 50$.
The price for the Phone has been decreased by 25$.
Current price for the Phone product is 625$.
```