CSE443 - Object Oriented Analysis and Design
Homework 2 Report

Ahmed Semih Özmekik
171044039

December 28, 2020

Abstract

Design decisions, explanations and class diagrams.

1 Question 1

An implementation was not requested for Question 1, but we tested with small snippets of code we wrote while answering the questions, and sent the source code for these tests. We will reference it throughout the answer to this first question.

This is the simple singleton class:

```
public class Singleton {
    private static Singleton obj;

    private Singleton() {}

    public static Singleton getInstance() {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

1.1 What happens if someone tries to clone a Singleton object using the clone() method inherited from Object? Does it lead to the creation of a second distinct Singleton object?

Let's simply try to see and interpret.

```
public class Driver {
    public static void main(String[] args) {

        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = (Singleton) singleton1.clone();
    }

}
```

If we try to compile it.

```
$ javac Driver.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Driver.java:6: error: clone() has protected access in Object
        Singleton singleton2 = (Singleton) singleton1.clone();
                                                      ^
1 error
```

As you can see, since the **clone ()** method is defined as protected in the **Object** class, when we make a simple implementation of the design pattern, this method does not endanger us by creating an anti-pattern.

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.

Figure 1: Clone method from Oracle **Object** class documentation

It's a compiler error. Hence, it does not lead to the creation of a second distinct Singleton object whatsoever.

1.2 Cloning Singletons should not be allowed. How can you prevent the cloning of a Singleton object?

We can prevent this in different ways. For example by raising **CloneNotSupportedException**:

```
public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

Or maybe by directly returning the singleton object:

```
public Object clone() {
    return obj;
}
```

1.3 Let's assume the class Singleton is a subclass of class Parent, that fully implements the Cloneable interface. How would you answer questions 1 and 2 in this case?

The problem speaks of such a hierarchy.

```
class Parent implements Cloneable {
    @Override
    protected Object clone() throws CloneNotSupportedException {
```

```
        return super.clone();
    }
}

public class Singleton extends Parent {
    private static Singleton obj;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

And indeed only such a hierarchy can create an anti-pattern for the singleton design pattern.

In such a case, our answer to Question 1.1 changes. In this case, a second distinct object is created and the design we want to do with the singleton design pattern is broken.

The measures in our answer to Question 1.2 should now be implemented in this hierarchy. As we are presenting, we can and should prevent this by either throwing an exception or returning the singleton object, or in other ways.

2 Question 2

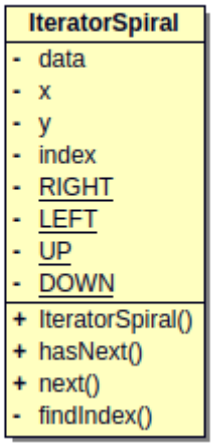
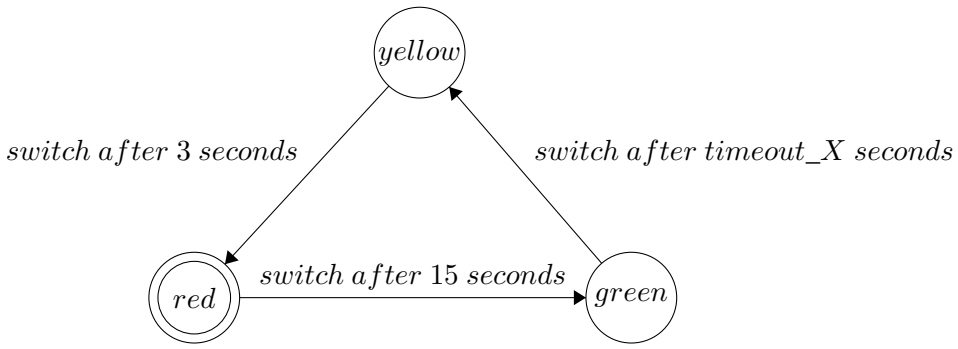


Figure 2: SpiralIterator Class Diagram

Since the Iterator class resides in Java as byte code, we don't need to show it in the class diagram, but it is simply an iterator that implements the Java's Iterator interface.

3 Question 3

- We have drawn the state diagram below according to the specified finite state machine.



Based on this diagram, we implemented our application with state design pattern. Below you can see the class diagram.

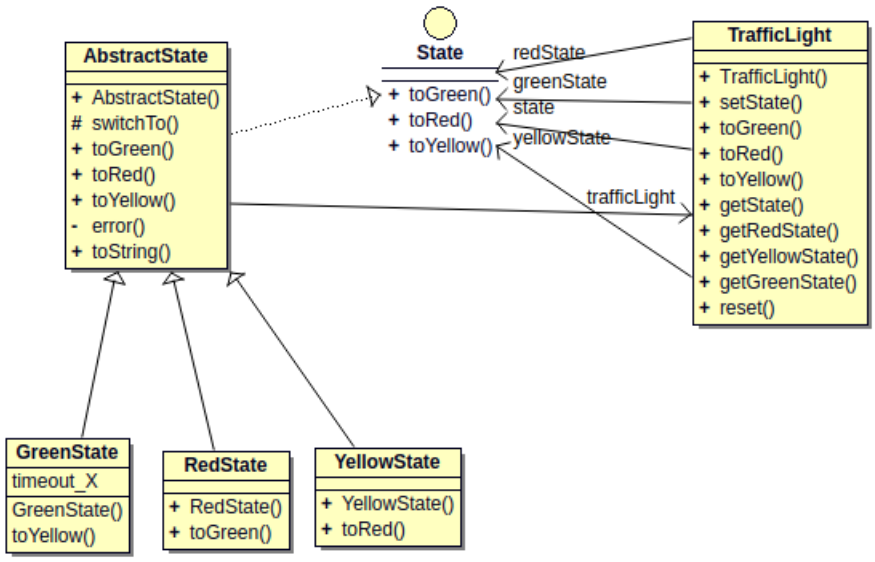


Figure 3: State Design Pattern Implementation

Even if it is not specified, we notice in the demands of the application that we need to make a context state that needs to switch between certain statuses. In this case, it is inevitable that we apply the State Design Pattern.

When realizing, it is logical to draw the state diagram first, because in this case, it is easier to see which transitions are possible between which statuses and thus leave the active and the others inactive when coding.

That's why we add an Abstract state class. While this intermediate class defines the fields and methods required for all states, it also makes all transitions inactive. It does

this with an error message. Then the child class deriving from it defines the appropriate transition method by override.

Then, we developed our application with the functionality expected to be added in the continuation of the question. And we added a simple implementation of the observer design pattern to it. The new class diagram looks like this.

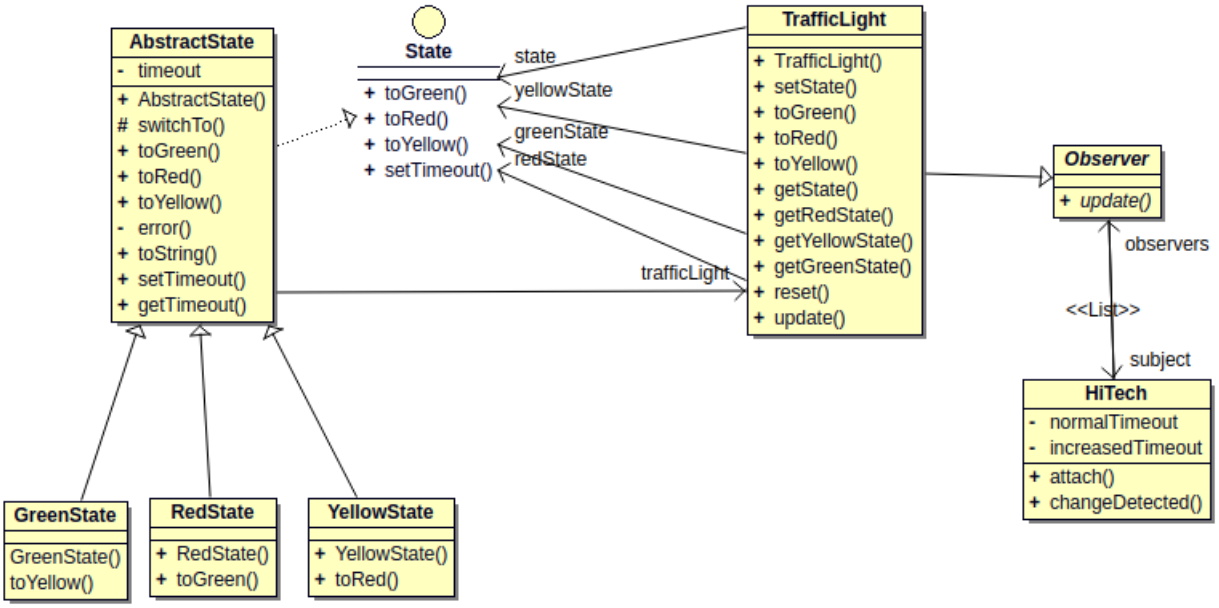


Figure 4: Observer Design Pattern Added Implementation

4 Question 4

In this design, what we had to do was add a function to the table, which is already a data structure, and improve it. But we only had the interface. In this case, we had to make a behavioral adjustment by keeping the loose coupling.

We did this by adding a behavior to it while wrapping this class, as defined in the Proxy Design Pattern. So it takes a new class composition with the table, and also as a table, adding behavior to it using the table underneath. The behavior it adds is a sync feature for both questions (a and b) separately.

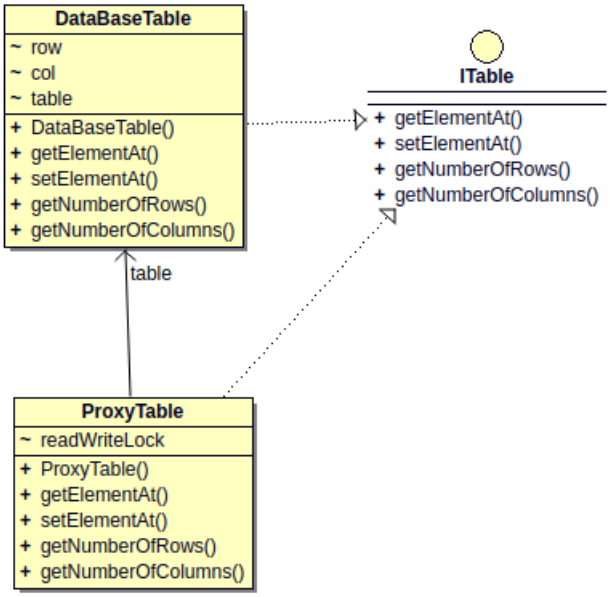


Figure 5: Proxy Design Pattern Implementation