CSE 443 OOAD Homework 2 Report

Yusuf Patoglu - 151044027 (ypatoglu@gtu.edu.tr)

December 30, 2020

Question 1.

In this question we have to answer questions about singleton pattern and cloning. The answers will be given straightforwardly. The source code also will be provided in order to justify my answers. my answers with simulating the cases. The questions are answered like this for a Singleton class which doesn't implement the *Cloneable* interface:

1. We can't clone a Singleton object using the *clone()* method inherited from Object because since we do not implement *Cloneable* interface in our class then we do not need to prevent cloning cause Java will not allow us to do. So no, it won't lead us to create second distinct Singleton object.
2. Since it's explained in first question we don't have to do anything explicitly. We can prevent the cloning of a Singleton object by doing nothing. Unless the *Cloneable* interface is not implemented for that object of course. *(See the source files in 1.1 directory.)*

Now the question says *"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?"* Answering questions again:

1. Since we have implemented the *Cloneable* interface explicitly now we can create clone of the Singleton object which will break the rules of Singleton Design pattern. *(See the source files in 2.1 directory.)*
2. We have our Singleton class with *Cloneable*. We can avoid the cloning of Singleton object with two ways.
   - Override the clone() method and invoke the *CloneNotSupportedException* exception from the clone method. *(See the source files in 2.2.a directory.)*
   - Return the original instance inside the overrided *clone* method.*(See the source files in 2.2.b directory.)*

Question 2.

In this question we have to traversal a matrix. There are two options like clockwise or non-clock wise directions. We'll implement only one of the methods so I chose spirally clockwise one. Since we can have multiple directions to iterate which means multiple traversal of objects then we need to provide a generic way to do this. So we need to use this case's iterator pattern. We will need to access our objects without exposing the internal representation, so the use of Iterator Pattern is a smart one.

In order to implement the Iterator design pattern first we will define an Iterator interface which will have essential methods like *hasNext()* and *next()*. Then in our Satellite class which does the traversal we will implement our interface.

Also including the main method the class diagram will look like this:

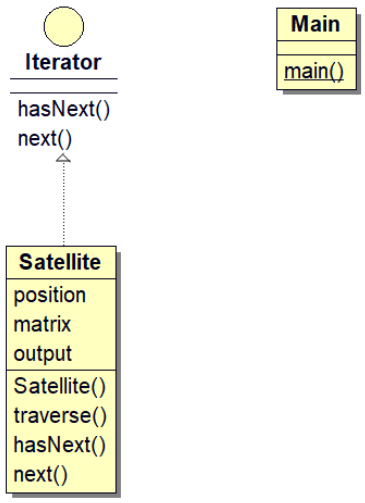


Figure 1: Iterator Pattern

Question 3.

In Question 3 we will simulate a traffic light in campus. The traffic light has three states. It will start with red and the rules will be as follows:

1. RED: switches to GREEN after 15 seconds.
2. YELLOW: switches to RED after 3 seconds.
3. GREEN: switches to YELLOW after 60 seconds (timeout_X).

Since the traffic light has states we will use state design pattern. In the advanced parts of the problem we will combine state design pattern with observer design pattern. So while drawing the class diagrams and state diagrams we will seperate the questions as: *"Q3 without observer pattern"* and *"Q3 with observer pattern"*. Also the source code will be seperated.

Without an observer pattern simply we will ignore the timeout condition. The FSA will look like:

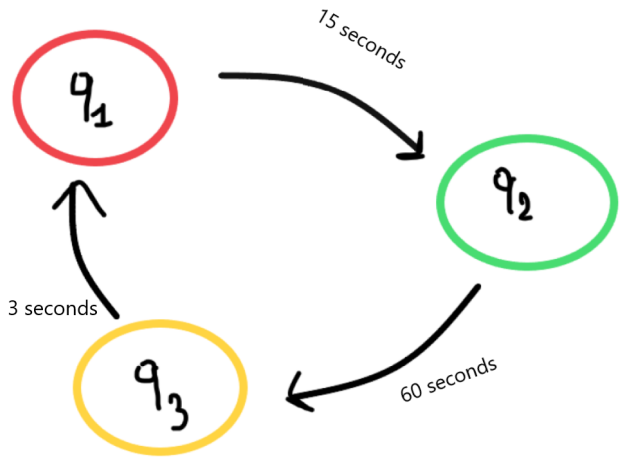
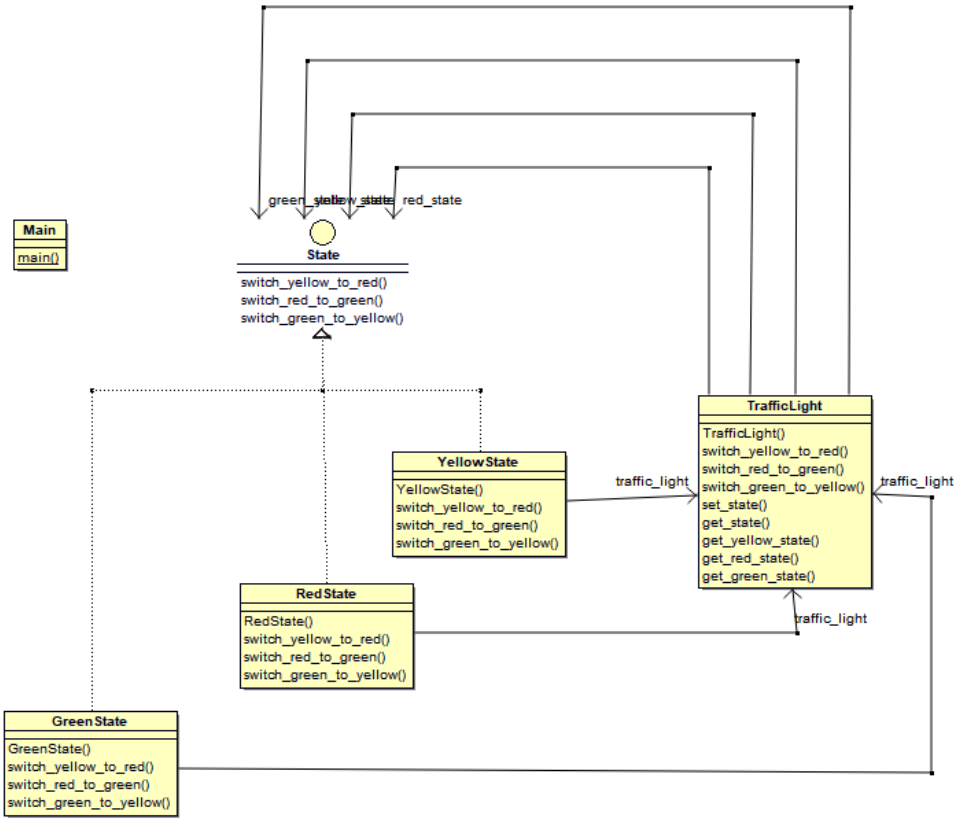


Figure 2: *q1: Red State, q2: Green State,q3: Yellow State*

The class diagram without observer pattern will be like:



In order to add optional state to our project we'll combine state design pattern with observer design pattern. The Publisher issues events of interest to other objects.

These events occur when the publisher changes its state or executes some behaviors.Subscriber Class: The Subscriber interface declares the notification interface.It has single update method. So the updated state diagram and class diagrams will be like:

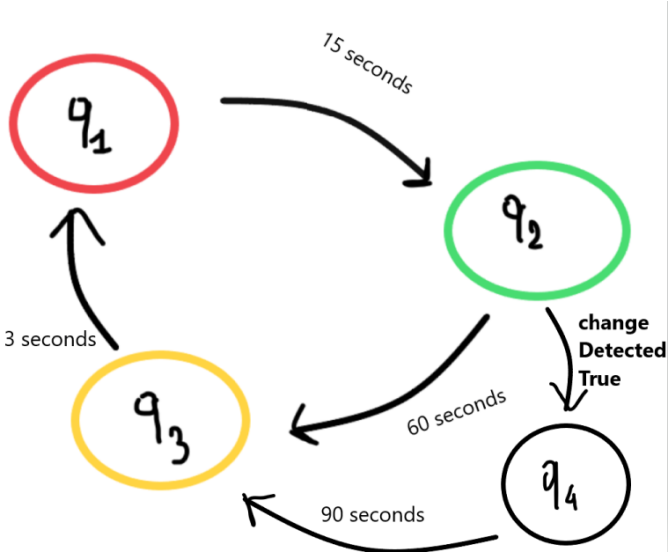


Figure 3: *q1: Red State, q2: Green State,q3: Yellow State, q4: Temporary State*

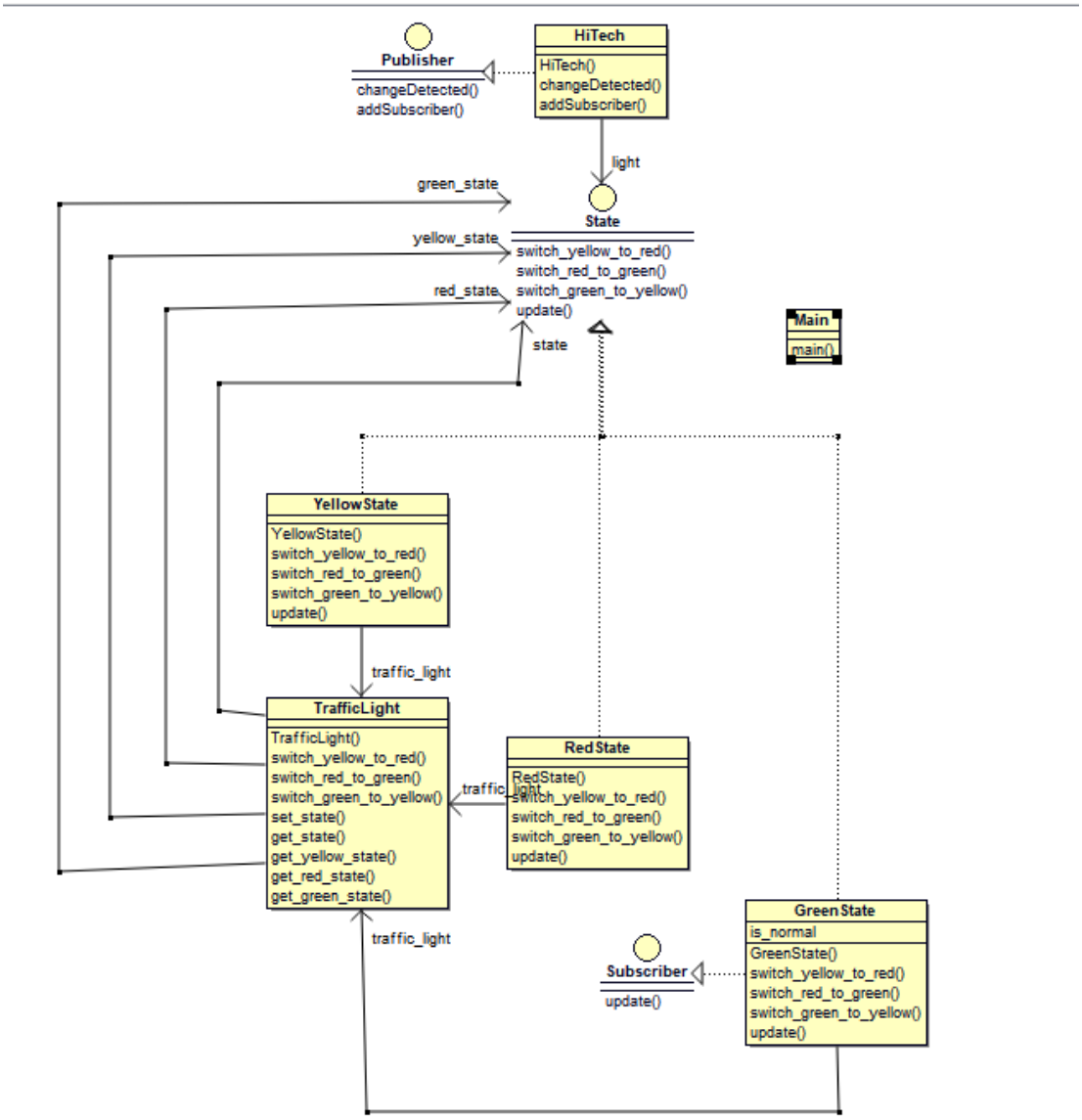


Figure 4: State Pattern Combined with Observer Pattern