Homework 2 Report

Elif Akgün
1801042251

1. Part 1

1.1 Design Explanation

1.1.1 Answer 1

There may be two situation in this question.
1)If Singleton class implements Cloneable interface, then a copy of existing object is created. clone() method does not create a new object. (Actually we can prevent copying the object and it explained in answer 2.)
2)If Singleton class does not implement Cloneable interface, then CloneNotSupportedExeption is thrown when clone() method is invoked.
To justify my answer, I created 3 classes: Singleton, SingletonCloneable, and SingletonThrowsException. Singleton class does not implement Cloneable interface but it overrides clone() method. SingletonCloneable implements Cloneable interface and overrides clone() method. Its clone() method returns super.clone(). SingletonThrowsException class implements Cloneable interface and overrides clone() method. Its clone() method throws CloneNotSupportedException. It is shown in Tests part what happens if we want to clone this classes' objects.

1.1.2 Answer 2

If we want to clone Singleton object, we must implement Cloneable interface. Otherwise CloneNotSupportedException is thrown. To prevent clone Singleton object, we should not implement Cloneable interface or if Cloneable interface is implemented, then we should throw an exception in clone method as SingletonThrowsException class. Another way to prevent object cloning is returning created object in clone() method.

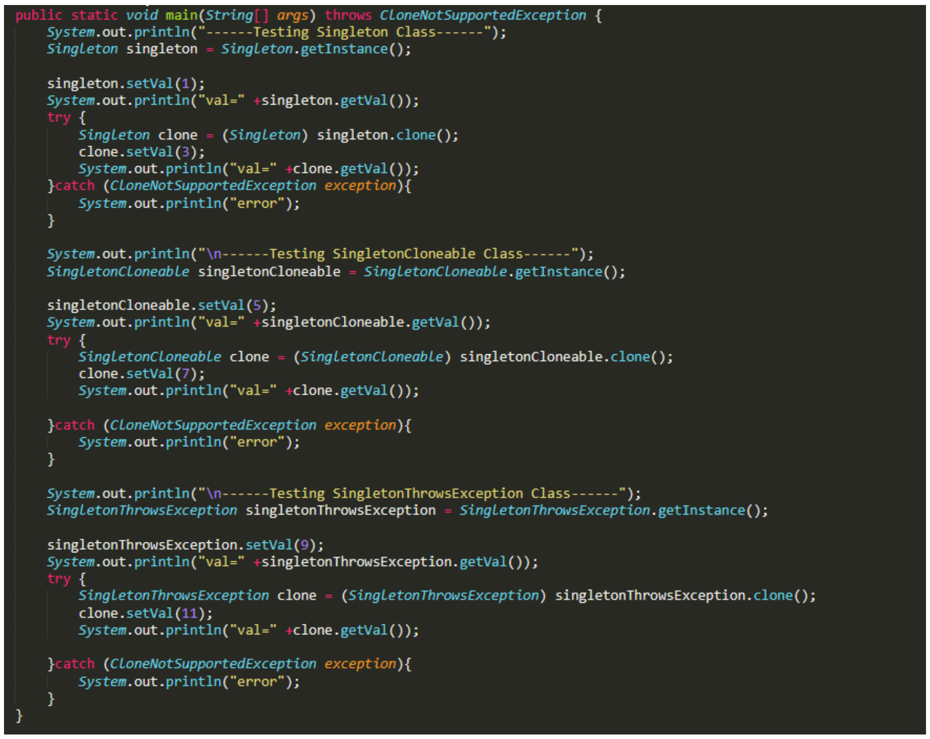
1.1.3 Answer 3

If the class Singleton is a subclass of class Parent, that fully implements the Cloneable interface, then Singleton also become Cloneable.
1) Because Singleton is a Cloneable, the copy of Singleton object's can be created with clone() method like SingletonCloneable class.

2)To prevent clone Singleton object, we should override clone() method and throw CloneNotSupportedException in this method. Also we may return created object in clone() method.

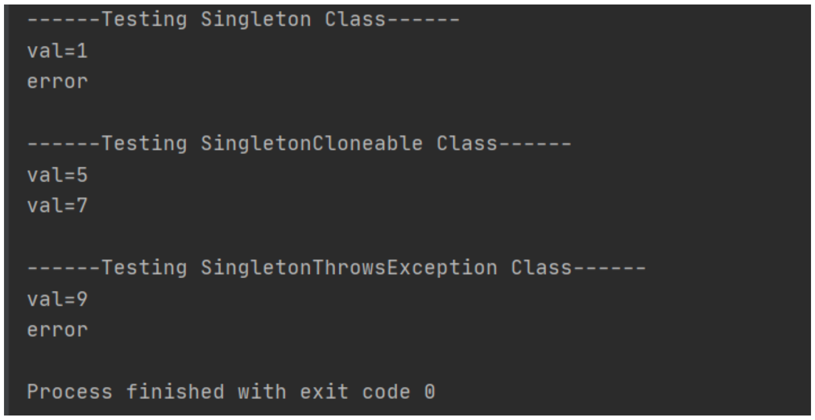
1.2 Test

Figure 1: Main Method



```java
public static void main(String[] args) throws CloneNotSupportedException {
    System.out.println("------Testing Singleton Class------");
    Singleton singleton = Singleton.getInstance();

    singleton.setVal(1);
    System.out.println("val=" +singleton.getVal());
    try {
        Singleton clone = (Singleton) singleton.clone();
        clone.setVal(3);
        System.out.println("val=" +clone.getVal());
    }catch (CloneNotSupportedException exception){
        System.out.println("error");
    }

    System.out.println("\n------Testing SingletonCloneable Class------");
    SingletonCloneable singletonCloneable = SingletonCloneable.getInstance();

    singletonCloneable.setVal(5);
    System.out.println("val=" +singletonCloneable.getVal());
    try {
        SingletonCloneable clone = (SingletonCloneable) singletonCloneable.clone();
        clone.setVal(7);
        System.out.println("val=" +clone.getVal());

    }catch (CloneNotSupportedException exception){
        System.out.println("error");
    }

    System.out.println("\n------Testing SingletonThrowsException Class------");
    SingletonThrowsException singletonThrowsException = SingletonThrowsException.getInstance();

    singletonThrowsException.setVal(9);
    System.out.println("val=" +singletonThrowsException.getVal());
    try {
        SingletonThrowsException clone = (SingletonThrowsException) singletonThrowsException.clone();
        clone.setVal(11);
        System.out.println("val=" +clone.getVal());

    }catch (CloneNotSupportedException exception){
        System.out.println("error");
    }
}
```

Figure 2: Output



```
------Testing Singleton Class------
val=1
error

------Testing SingletonCloneable Class------
val=5
val=7

------Testing SingletonThrowsException Class------
val=9
error

Process finished with exit code 0
```

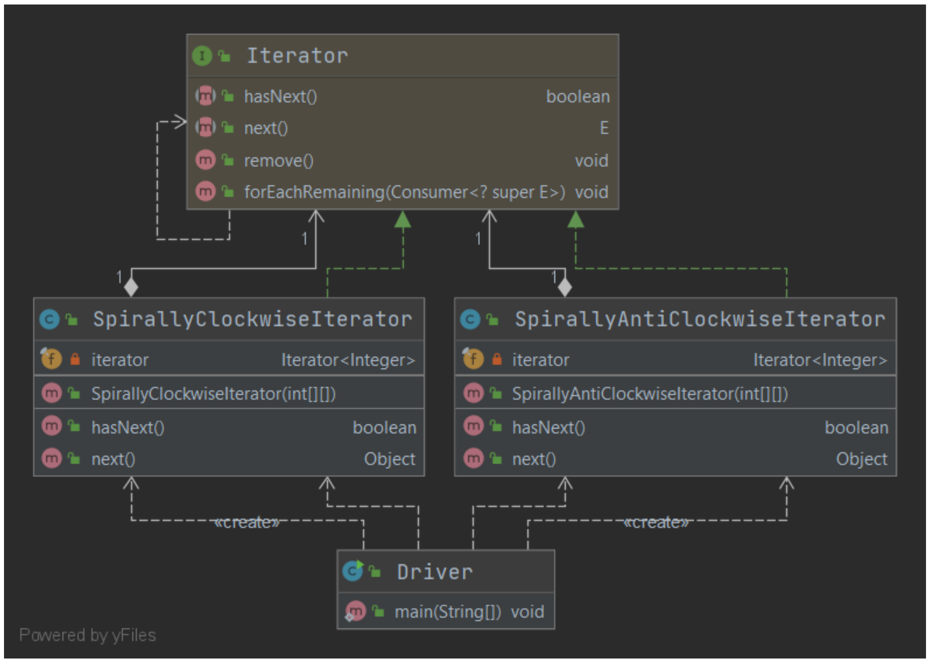
2. Part 2

2.1 Design Explanation

Iterator allows us to traverse the collection. We can access the data element or remove the data of the collection with Iterator interface easily. Iterator has 3 methods: hasNext() method, next() method, and remove() method. The hasNext() method tells us if there are more elements in the aggregate to iterate through. The next() method returns the next object in the aggregate. The remove() method allows us to remove the last item returned by next() method from the aggregate.
In this part, I implemented 2 iterators. SpirallyClockwiseIterator to prints the 2D array spirally clockwise and SpirallyAntiClockwiseIterator to prints the 2D array spirally anti-clockwise. Both iterators implements Java's Iterator interface. In SpirallyClockwiseIterator class, I traversed the 2D array clockwise and I collected array's elements in an arraylist. In SpirallyAntiClockwiseIterator class, I traversed the 2D array anti-clockwise and I collected array's elements in an arraylist. In both class, if there are more element the hasNext() method returns true and the next() method returns the netxt element. I did not implement remove() method. Each iteration, I decrease the size because it traverses spirally. In main method, I printed the elements both clockwise and anti-clockwise.

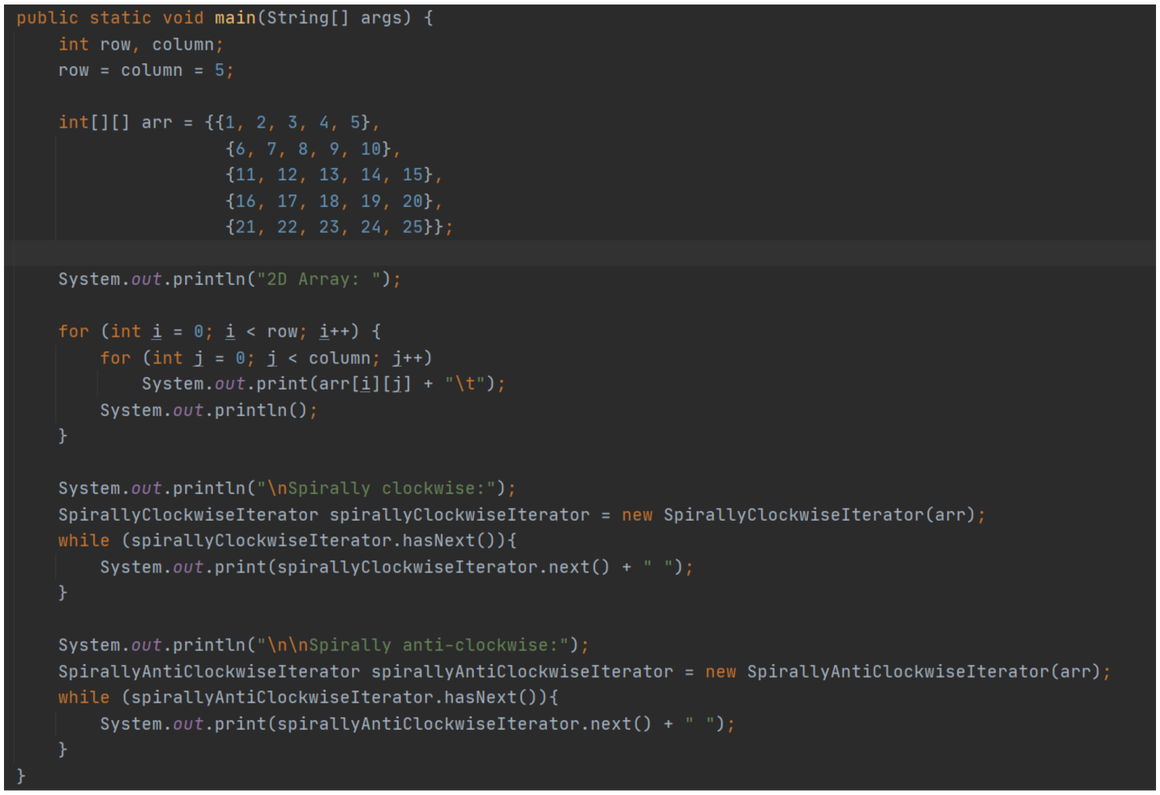
2.2 Class Diagram

Figure 3: Class Diagram of Part 2


Iterator
hasNext() boolean
next() E
remove() void
forEachRemaining(Consumer<? super E>) void
1
1
1
1
SpirallyClockwiseIterator
iterator Iterator<Integer>
SpirallyClockwiseIterator(int[][])
hasNext() boolean
next() Object
SpirallyAntiClockwiseIterator
iterator Iterator<Integer>
SpirallyAntiClockwiseIterator(int[][])
hasNext() boolean
next() Object
«create»
«create»
Driver
main(String[]) void
Powered by yFiles

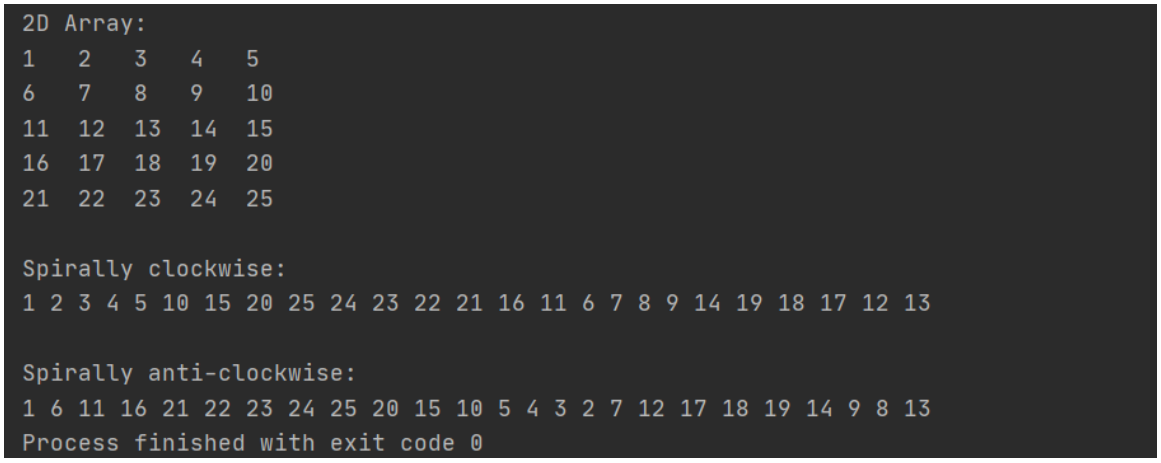
2.3 Test

Figure 4: Main Method



```java
public static void main(String[] args) {
    int row, column;
    row = column = 5;

    int[][] arr = {{1, 2, 3, 4, 5},
                   {6, 7, 8, 9, 10},
                   {11, 12, 13, 14, 15},
                   {16, 17, 18, 19, 20},
                   {21, 22, 23, 24, 25}};

    System.out.println("2D Array: ");

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < column; j++)
            System.out.print(arr[i][j] + "\t");
        System.out.println();
    }

    System.out.println("\nSpirally clockwise:");
    SpirallyClockwiseIterator spirallyClockwiseIterator = new SpirallyClockwiseIterator(arr);
    while (spirallyClockwiseIterator.hasNext()){
        System.out.print(spirallyClockwiseIterator.next() + " ");
    }

    System.out.println("\n\nSpirally anti-clockwise:");
    SpirallyAntiClockwiseIterator spirallyAntiClockwiseIterator = new SpirallyAntiClockwiseIterator(arr);
    while (spirallyAntiClockwiseIterator.hasNext()){
        System.out.print(spirallyAntiClockwiseIterator.next() + " ");
    }
}
```

Figure 5: Output



```
2D Array:
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25

Spirally clockwise:
1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6 7 8 9 14 19 18 17 12 13

Spirally anti-clockwise:
1 6 11 16 21 22 23 24 25 20 15 10 5 4 3 2 7 12 17 18 19 14 9 8 13
Process finished with exit code 0
```

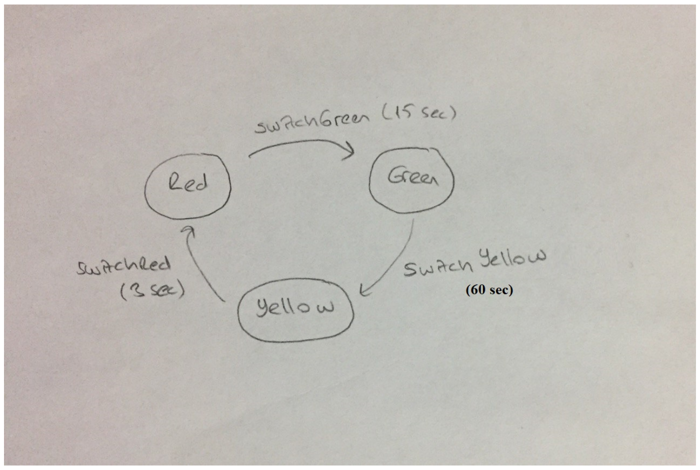
3. Part 3

3.1 Design Explanation

3.1.1 State Pattern

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable. The Context is the class that can have a number of internal states. In my homework, the TrafficLight is the Context. Whenever the request is made on the Context it is delegated to the state to handle. ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.
In this homework, states are RED, GREEN, and YELLOW. I create a State abstract class and it has switchRed(), switchGreen(), and switchYellow() methods. switchGreen() method switches the state from RED to GREEN, switchYellow() method switches the state from GREEN to YELLOW, and switchRed() method switches the state from YELLOW to RED. Each method prints error message in abstract class. Then I created State's subclasses: Red, Green, and Yellow. They override switchGreen(), switchYellow(), and switchRed() methods respectively. Unlike other states, Green has a data field which is *timeout_X* because its waiting time changes according to traffic. Also I created TrafficLight class. It has State objects for each states and current state. It has switchRed(), switchGreen(), and switchYellow() methods and it delegates this methods to related State. And it has a setState method to set cuurent state. There are getter methods for each State type and they return related state object.

Figure 6: State diagram of traffic light

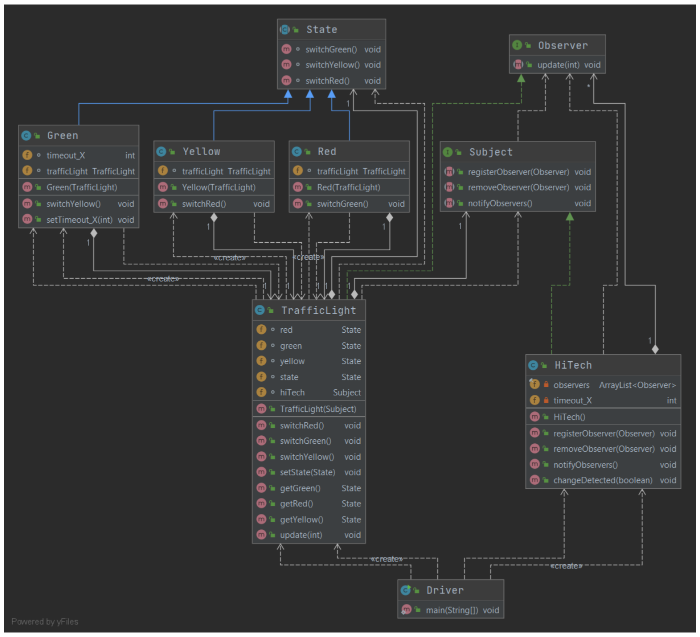


3.1.2 Observer Pattern

In Observer Pattern, there is one subject and there are some observers. With the observer pattern, the subject is the object that contains the state and controls it. So, there is one subject with state. The observers, use the state, even if they don't own it. There are many observers and they rely on the subject to tell them when its state changes. So there is a relationship between the one subject to the many observers.
For observer part, I added the Observer and Subject interfaces, and HiTech class. The Observer interface has update method to updates *timeout_X* value. The Subject interface has registerObserver, removeObserver, and notifyObservers methods. registerObserver method registers an observer and adds new observer to observers list. removeObserver method removes the observer from the observers list. notifyObservers method called when new update is available. HiTech class impements Subject interface and overrides its methods. In addition, it has changeDetected method. Whenever the camera detects a change of traffic, this method called. If flag is true, then *timeout_X* is increased from 60 seconds to 90 seconds.
For Observer Pattern, I did some changes in TrafficLight class. It implements Observer class because it subscribes to the camera's software which is HiTech. So, it overrides update method and it updates *timeout_X* value according to traffic. When it detects a lot of traffic *timeout_X* is increased from 60 seconds to 90 seconds.

3.2 Class Diagram

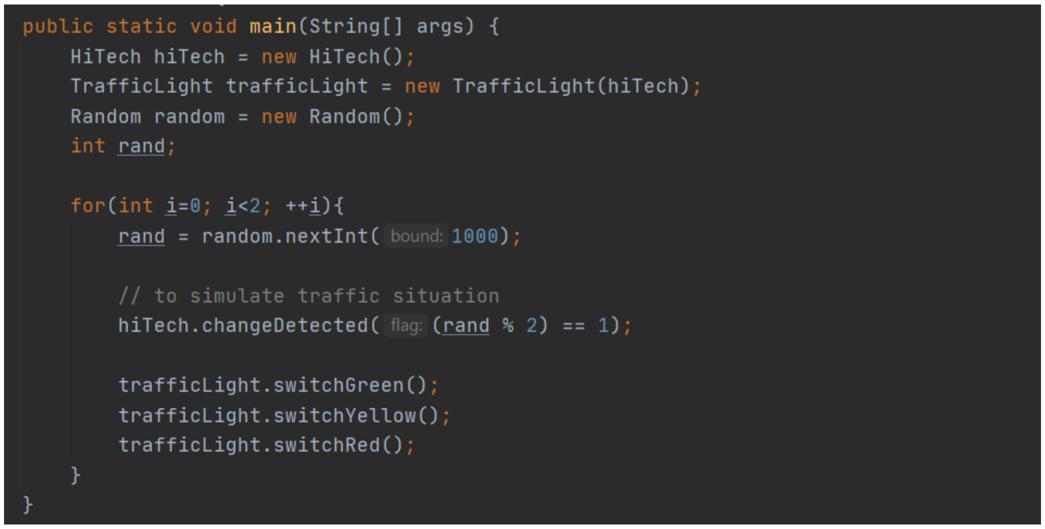
Figure 7: Class Diagram of Part 3



3.3 Test

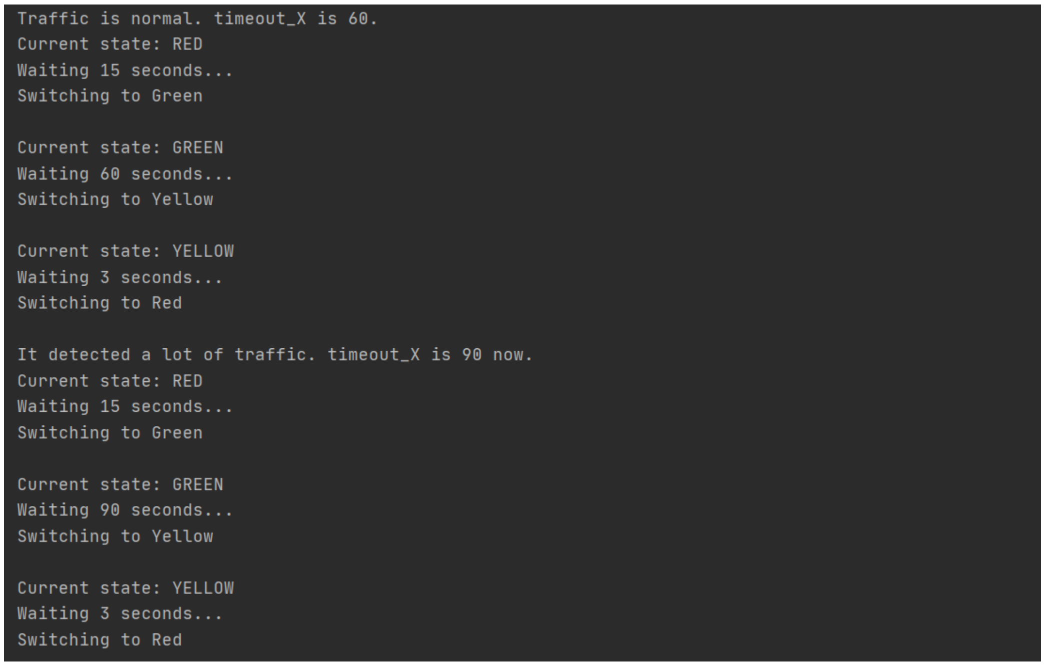
In main method, I used a random number generator to simulate traffic situation. If the mode of random number with 2 is 1, then *timeout_X* updated as 90. Otherwise *timeout_X* is 60. To illustrate transition between states, I used a for loop that iterates 2 times.

Figure 8: Main Method



```java
public static void main(String[] args) {
    HiTech hiTech = new HiTech();
    TrafficLight trafficLight = new TrafficLight(hiTech);
    Random random = new Random();
    int rand;

    for(int i=0; i<2; ++i){
        rand = random.nextInt( bound: 1000);

        // to simulate traffic situation
        hiTech.changeDetected( flag: (rand % 2) == 1);

        trafficLight.switchGreen();
        trafficLight.switchYellow();
        trafficLight.switchRed();
    }
}
```

Figure 9: Output



```
Traffic is normal. timeout_X is 60.
Current state: RED
Waiting 15 seconds...
Switching to Green

Current state: GREEN
Waiting 60 seconds...
Switching to Yellow

Current state: YELLOW
Waiting 3 seconds...
Switching to Red

It detected a lot of traffic. timeout_X is 90 now.
Current state: RED
Waiting 15 seconds...
Switching to Green

Current state: GREEN
Waiting 90 seconds...
Switching to Yellow

Current state: YELLOW
Waiting 3 seconds...
Switching to Red
```

4. Part 4

4.1 Design Explanation

In this part, there is a class DataBaseTable which implements ITable. We do not have the source code for this class library, but we have the complete documentation and know about the interface ITable.
I created DataBaseTable class. It overrides ITable's methods. Proxy classes delegates methods to DataBaseTable class's method with using DataBaseTable object.

4.1.1 *a*

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. A proxy object can act as the intermediary between the client and the target object.
In this part, I used Synchronization Proxy because DataBaseTable does not provide the capability to allow clients to lock individual table rows. Thus, two clients might end up modifying the same row simultaneously, and we don't want that happening. Synchronization Proxy provides safe access to a subject from multiple threads.
I created ProxyDataBaseTableA which implements ITable. I used a mutex and conditional variables to solve this problem. In getElementAt method, each threads take mutex and check active writer number. If a reader thread call getElementAt method and a writer thread sets the table, reader thread wait for writer thread. If a reader thread call getElementAt method and another reader thread calls the getElementAt; it allows both because no thread set the table. In setElementAt method, each threads take mutex and check both active writer and active reader number. If a writer thread call setElementAt method and if a reader thread reads the table, or a writer thread sets the table, writer thread wait for this thread.

4.1.2 *b*

There are too many getElementAt calls that keep "locking" the Table's rows, and this way the clients that need to modify them using setElementAt wait too long to acquire the table lock. This problem like as part a but there are a difference. Writer threads should have priority.
To solve this problem, I used Synchronization Proxy and I also created ProxyDataBaseTableB implements ITable. In getElementAt method, reader threads wait for both waiting writers and active writers. In setElementAt method, writer threads wait for only active writers and active readers. I implemented the following pseudocodes used in System Programming course for this part. Reader threads are threads that call getElementAt, and writer threads are threads that call setElementAt.
number of active readers AR = 0
number of active writers AW = 0
number of waiting readers WR = 0
number of waiting writers WW = 0

Condition variable okToRead
Condition variable okToWrite
Lock lock

Figure 10: Pseudocode of reader thread

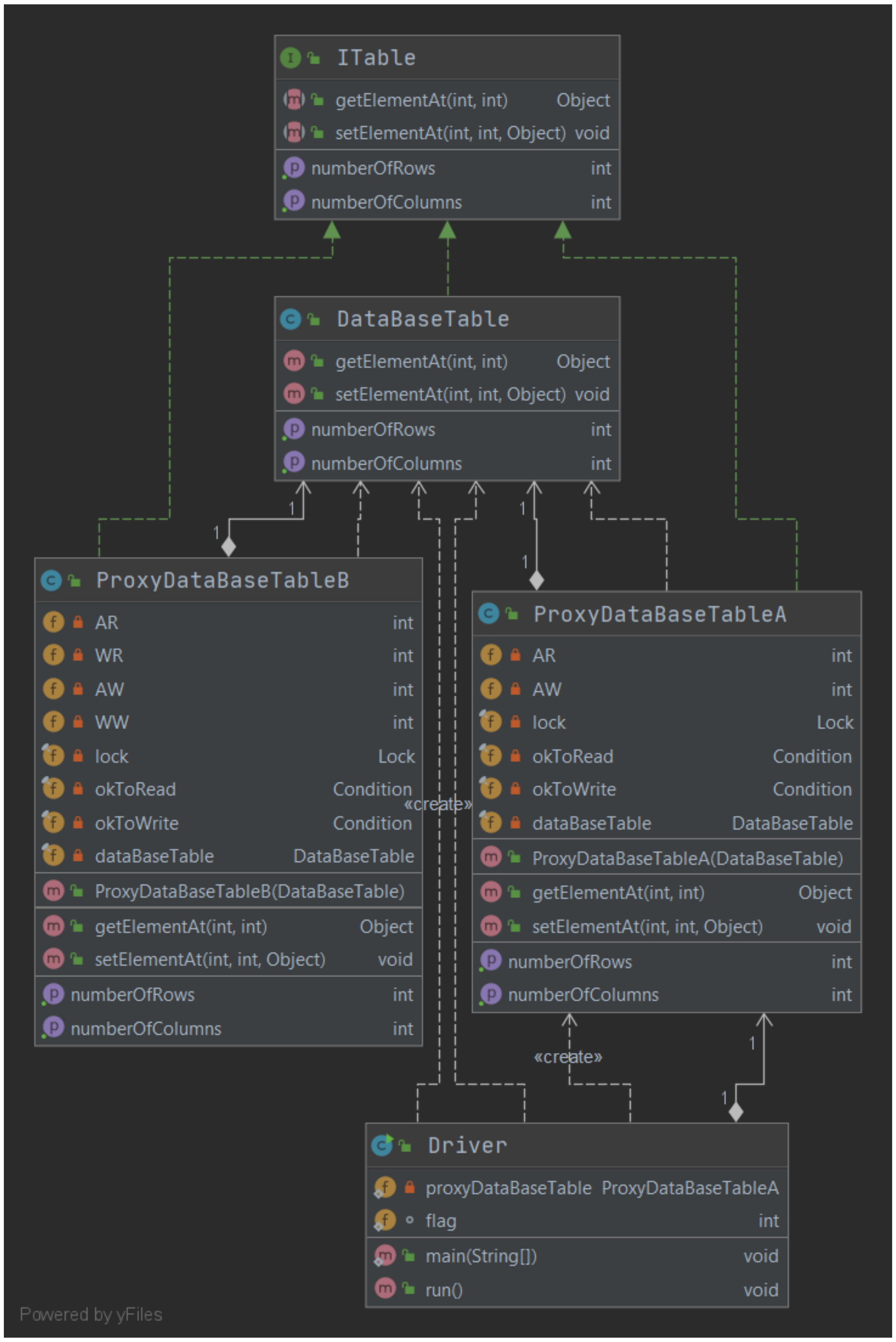
```
Reader()
    lock(m);
    while ((AW + WW) > 0) {   // if any writers, wait
       WR++;      // waiting reader
       cwait(okToRead,m);
       WR--;
    }
    AR++;         // active reader
    unlock(m);
    Access DB
    lock(m);
    AR--;
    if (AR == 0 && WW > 0)
       signal(okToWrite, m);
    unlock(m);
```

Figure 11: Pseudocode of writer thread

```
Writer()
    lock(m);
    while ((AW + AR) > 0) { // if any readers or writers, wait
       WW++;                         // waiting writer
       cwait(okToWrite, m);
       WW--;
    }
    AW++;                         // active writer
    unlock(m);
    Access DB
    lock(m);
    AW--;
    if (WW > 0)                   // give priority to other writers
       signal(okToWrite, m);
    else if (WR > 0)
       broadcast(okToRead, m);
    unlock(m);
```

4.2 Class Diagram

Figure 12: Class Diagram of Part 4


ITable
getElementAt(int, int) Object
setElementAt(int, int, Object) void
numberOfRows int
numberOfColumns int
DataBaseTable
getElementAt(int, int) Object
setElementAt(int, int, Object) void
numberOfRows int
numberOfColumns int
ProxyDataBaseTableB
AR int
WR int
AW int
WW int
lock Lock
okToRead Condition
okToWrite Condition
dataBaseTable DataBaseTable
ProxyDataBaseTableB(DataBaseTable)
getElementAt(int, int) Object
setElementAt(int, int, Object) void
numberOfRows int
numberOfColumns int
ProxyDataBaseTableA
AR int
AW int
lock Lock
okToRead Condition
okToWrite Condition
dataBaseTable DataBaseTable
ProxyDataBaseTableA(DataBaseTable)
getElementAt(int, int) Object
setElementAt(int, int, Object) void
numberOfRows int
numberOfColumns int
«create»
«create»
1
1
1
1
1
1
Driver
proxyDataBaseTable ProxyDataBaseTableA
flag int
main(String[]) void
run() void
Powered by yFiles

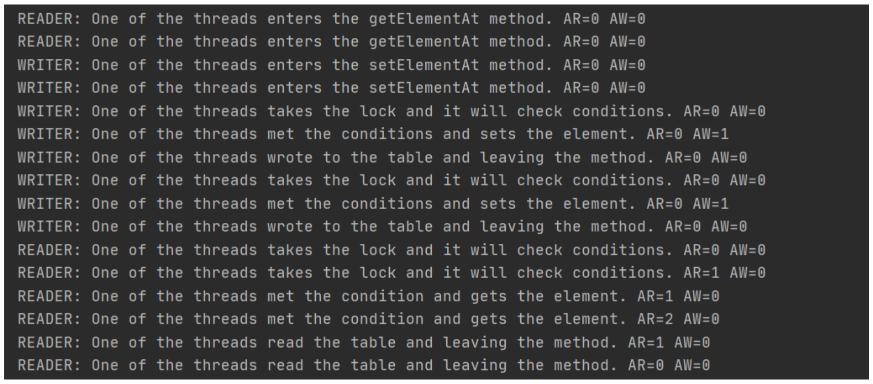
4.3 Test

I wrote some messages in getElementAt and setElementAt methods to test program.

4.3.1 *a*

As shown in below, there can be more than 1 active reader thread but only 1 active writer thread at the same time. Also if there are active writer, reader thread waits for this thread and vice versa.

Figure 13: Sample output for part a.

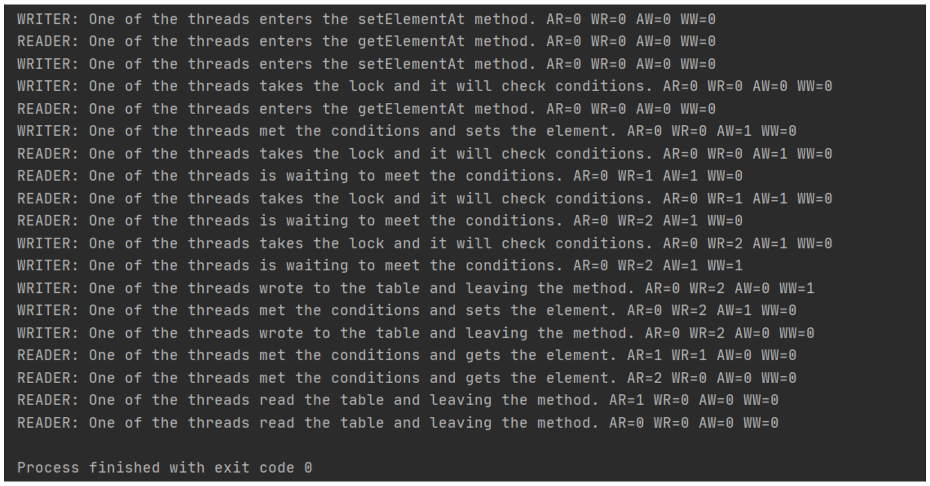


```
READER: One of the threads enters the getElementAt method. AR=0 AW=0
READER: One of the threads enters the getElementAt method. AR=0 AW=0
WRITER: One of the threads enters the setElementAt method. AR=0 AW=0
WRITER: One of the threads enters the setElementAt method. AR=0 AW=0
WRITER: One of the threads takes the lock and it will check conditions. AR=0 AW=0
WRITER: One of the threads met the conditions and sets the element. AR=0 AW=1
WRITER: One of the threads wrote to the table and leaving the method. AR=0 AW=0
WRITER: One of the threads takes the lock and it will check conditions. AR=0 AW=0
WRITER: One of the threads met the conditions and sets the element. AR=0 AW=1
WRITER: One of the threads wrote to the table and leaving the method. AR=0 AW=0
READER: One of the threads takes the lock and it will check conditions. AR=0 AW=0
READER: One of the threads takes the lock and it will check conditions. AR=1 AW=0
READER: One of the threads met the condition and gets the element. AR=1 AW=0
READER: One of the threads met the condition and gets the element. AR=2 AW=0
READER: One of the threads read the table and leaving the method. AR=1 AW=0
READER: One of the threads read the table and leaving the method. AR=0 AW=0
```

4.3.2 *b*

As shown in below, if there are both waiting reader and waiting writer, waiting writer will run because it has priority. And also there can be more than 1 active reader thread but only 1 active writer thread at the same time.

Figure 14: Sample output for part b.



```
WRITER: One of the threads enters the setElementAt method. AR=0 WR=0 AW=0 WW=0
READER: One of the threads enters the getElementAt method. AR=0 WR=0 AW=0 WW=0
WRITER: One of the threads enters the setElementAt method. AR=0 WR=0 AW=0 WW=0
WRITER: One of the threads takes the lock and it will check conditions. AR=0 WR=0 AW=0 WW=0
READER: One of the threads enters the getElementAt method. AR=0 WR=0 AW=0 WW=0
WRITER: One of the threads met the conditions and sets the element. AR=0 WR=0 AW=1 WW=0
READER: One of the threads takes the lock and it will check conditions. AR=0 WR=0 AW=1 WW=0
READER: One of the threads is waiting to meet the conditions. AR=0 WR=1 AW=1 WW=0
READER: One of the threads takes the lock and it will check conditions. AR=0 WR=1 AW=1 WW=0
READER: One of the threads is waiting to meet the conditions. AR=0 WR=2 AW=1 WW=0
WRITER: One of the threads takes the lock and it will check conditions. AR=0 WR=2 AW=1 WW=0
WRITER: One of the threads is waiting to meet the conditions. AR=0 WR=2 AW=1 WW=1
WRITER: One of the threads wrote to the table and leaving the method. AR=0 WR=2 AW=0 WW=1
WRITER: One of the threads met the conditions and sets the element. AR=0 WR=2 AW=1 WW=0
WRITER: One of the threads wrote to the table and leaving the method. AR=0 WR=2 AW=0 WW=0
READER: One of the threads met the conditions and gets the element. AR=1 WR=1 AW=0 WW=0
READER: One of the threads met the conditions and gets the element. AR=2 WR=0 AW=0 WW=0
READER: One of the threads read the table and leaving the method. AR=1 WR=0 AW=0 WW=0
READER: One of the threads read the table and leaving the method. AR=0 WR=0 AW=0 WW=0

Process finished with exit code 0
```