# CSE 443 – OBJECT ORIENTED ANALYSIS AND DESIGN
# HOMEWORK #02 – REPORT
# GÖKHAN HAS – 161044067

## QUESTION 1:

**1.1)** In Singleton, the user is not allowed to create an object on his own. The user can only request an object. And when the object requests, the user is given the object that exists in memory each time. Thus, the user has to work with only one object. I made experiments by creating a singleton class for this part. Normally, the purpose of using the Singleton design pattern is to perform operations on an object. Cloning is an unfavorable situation. CloneNotSupportedException occurs when the clone () method, which comes directly from the object class, is called and the cloning operation cannot be performed. In this example, the Singleton class has not implemented the Cloneable interface. Clone method does not create a new object, it creates a copy of the existing object.

```
class Singleton {
    private static Singleton singleton = null;
    public Singleton() {
        System.out.println("Singleton Object");
    }

    public static Singleton getSingleton() {
        if(singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

**1.2)** It is possible to block the singleton class by overriding the clone () method from the Object, leaving it blank without doing anything, or throwing an exception or printing a message. As long as the super.clone () method should not be called in this method.

```
@Override
public Object clone() throws CloneNotSupportedException {
    return new CloneNotSupportedException();
}
```

**1.3)** If the Singleton class was derived from the Cloneable interface and super.clone () was called in this overidized clone () method, the cloning process would have been done. However, a new object would not be created. A copy of the existing object would be created.

```
class Singleton implements Cloneable{
    private static Singleton singleton = null;
    public Singleton() {
        System.out.println("Singleton Object");
    }

    public static Singleton getSingleton() {
        if(singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

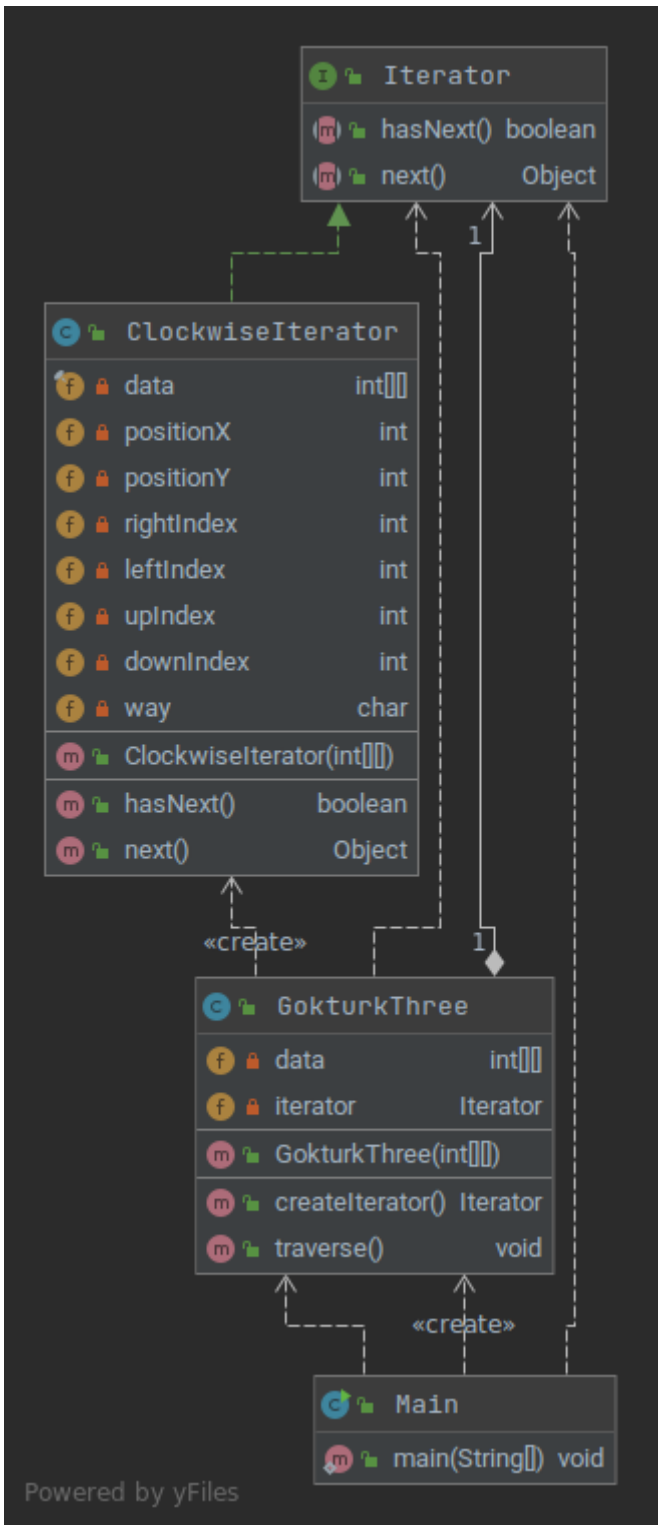
## QUESTION 2:

Iterator, design pattern; Allows access to the components of a compound object sequentially without revealing the representation of the object's principal expression. Iterator design pattern, to minimize the link between the structure of a list and its working style with other parts of the application; used to isolate the objects in the list from the application respectively.

For this part, I first made sure that the two-dimensional array coming from the satellite was written in a clockwise spiral form. In order to print this using Iterator, I first created an interface called Iterator. This interface includes the hasNext () and next () methods. The hasNext () method returns true if the next element is present, false otherwise. The next () method returns the next element.

There is the ClockwiseIterator class that implements this Iterator interface. In this class, there is a two-dimensional array of data that simulates data from a two-dimensional satellite. Assignment to this variable is done in Constructor. The hasNext () and next () methods are overridden. The hasNext () method returns true if there is an element according to the direction to go in a spiral, otherwise false. The next () method returns an element if it exists. Elements are simulated as Integers.

Then there is the GokturkThree class. This class holds data from the satellite and holds an iterator with an Iterator reference. It has a method createIterator () that returns this iterator. In this method, the variable with the iterator reference is assigned by creating the ClockwiseIterator object. Traverse method navigates on this iterator and prints the data on the screen.




```
TRAVERSE CLOCK WISE SPIRALLY :  1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
TRAVERSE CLOCK WISE SPIRALLY (Main) :  1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

Process finished with exit code 0
```

**QUESTION 3:**

**3.1) State Design Pattern**

The state design pattern can be used if the behavior of the object changes when the state of the object changes, that is, if the objects behave differently in different situations. Using this design pattern prevents the changing behavior of objects depending on their state from being controlled by complex "if / else" or "switch" statements.

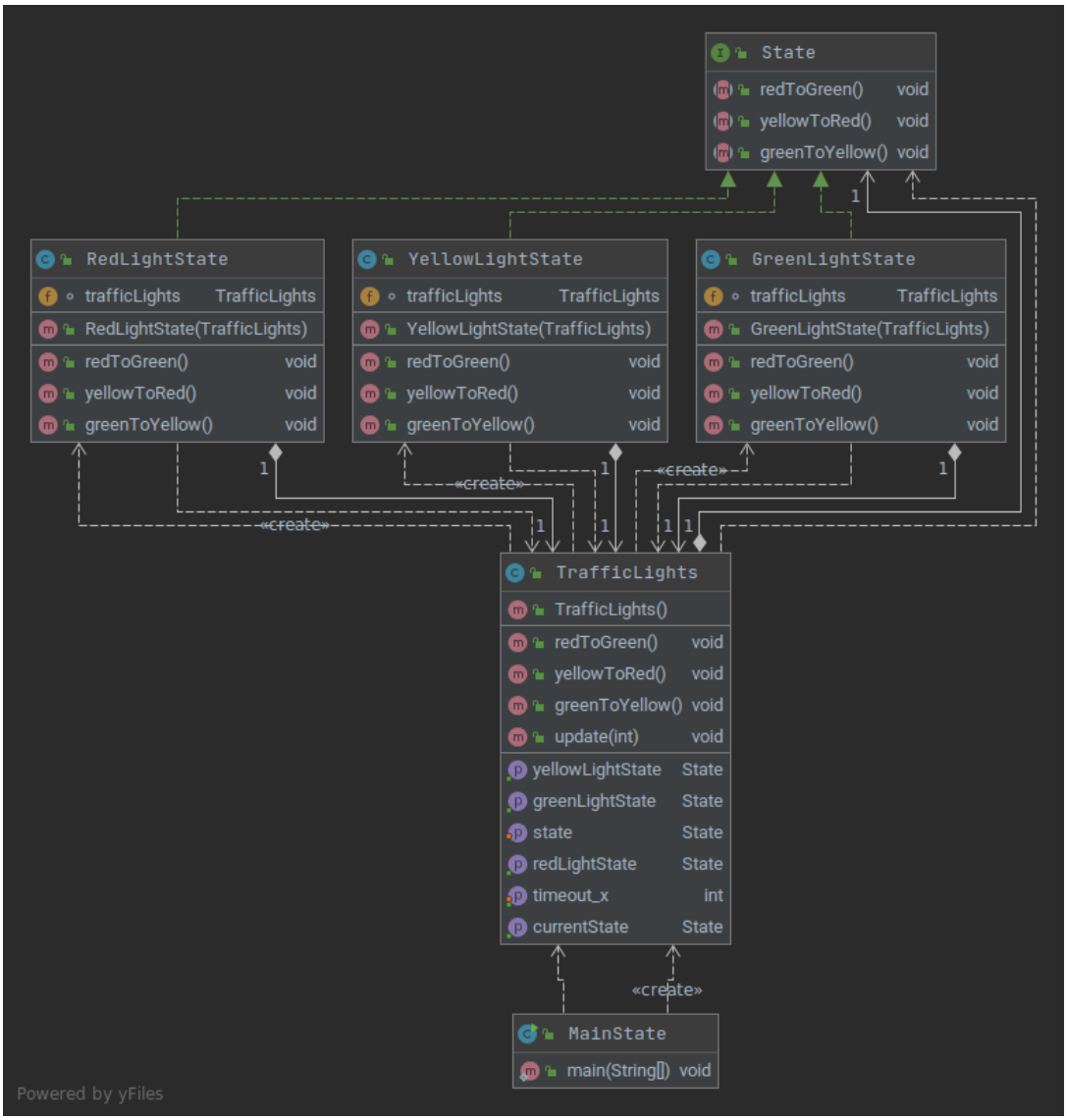
First, there is a state interface. This interface keeps the changed state methods. There are 3 traffic state changing methods in the interface.

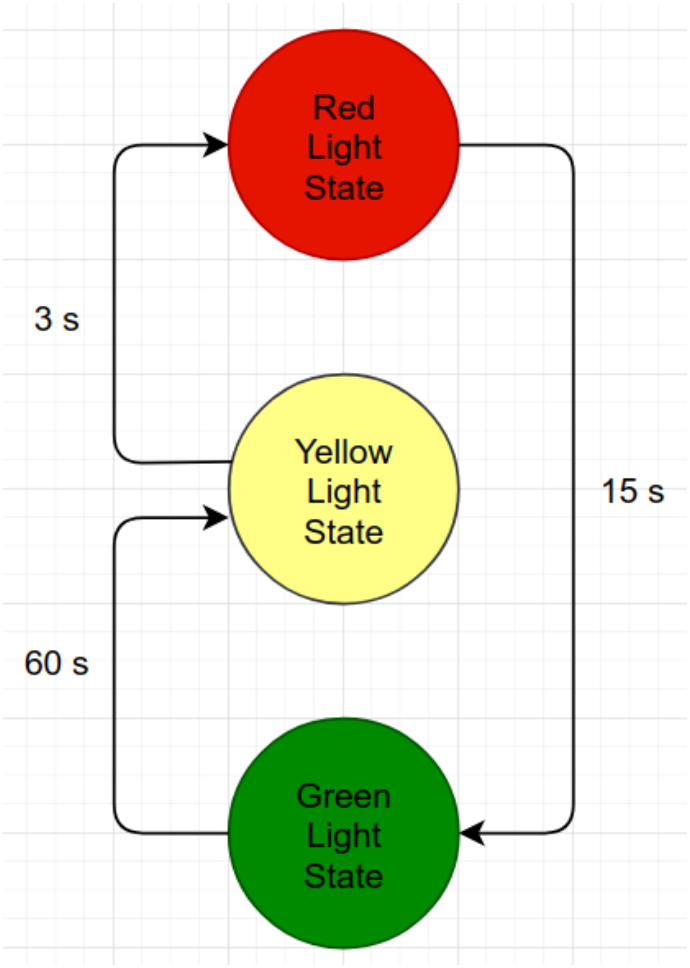
There is later the TrafficLights class. In this class, there are 3 variables with State reference to indicate the states of 3 different traffic lights. These are redLightState, yellowLightState, and greenLightState, respectively. Likewise, there is a variable named currentState to specify the current state. The variable named timeout_x keeps the green light on. This time is 60 seconds by default. In Contructor, the objects of the classes that are related to these 3 states are created with the new keyword. This case classes will be explained later. There are 3 methods that simulate the change of situations. It calls the corresponding method of the currentState () variable in them. The names of these methods are redToGreen, yellowToRed and greenToYellow respectively. There are 3 getter methods that return these states. And there is also a setState () method to change the currentState () value.

The RedLightState, YellowLightState, and GreenLightState classes, which simulate states, implement the State interface. And inside each of them there is a data field that holds the reference of the TrafficLights object. Assignment to this object is done in its Constructors. There are 3 methods of changing the state from the state interface. However, each class operates on its own method. For example, only the redToGreen method in the RedLightState class does the job. The other two methods (yellowToRed and greenToYellow) "ERROR! This is invalid! The light is RED!" prints the error message in the form. The redToGreen () method, on the other hand, prints the information on the screen and switches to the trafficLights.getGreenLightState () state with the setState () method in TrafficLights. In the same way in other lights, they do these operations only in the state transition methods specific to their own state.

```
------- NORMAL TEST -------
15 Seconds Wait ...
Red to Green switch ...

60 Seconds Wait ...
Green to Yellow switch ...

3 Seconds Wait ...
Yellow to Red switch ...
```

```
------- ERROR TEST -------
ERROR ! This is invalid ! The light is RED !
ERROR ! This is invalid ! The light is RED !
15 Seconds Wait ...
Red to Green switch ...

ERROR ! This is invalid ! The light is GREEN !
ERROR ! This is invalid ! The light is GREEN !
60 Seconds Wait ...
Green to Yellow switch ...

ERROR ! This is invalid ! The light is YELLOW !
ERROR ! This is invalid ! The light is YELLOW !
3 Seconds Wait ...
Yellow to Red switch ...


Process finished with exit code 0
```






State diagram starts with Red Light State.

### 3.2) Observer Design Pattern

Observer design pattern is one of the behavioral design patterns. It provides one-to-many relationship between objects. When an object changes its state, all other objects connected to it are alerted and updated automatically.

Here is the Subject interface and the Observer interface. Subject interface has the registerObserver (), removeObserver () and notifyObservers () methods. The registerObserver () method is the method by which observer is recorded. removeObserver () is the method by which observer is removed. notifyObservers () is the method by which observer reports. Observer interface has an update (int timeout_x) method and will be used to inform the relevant subject when the timeout value is updated.

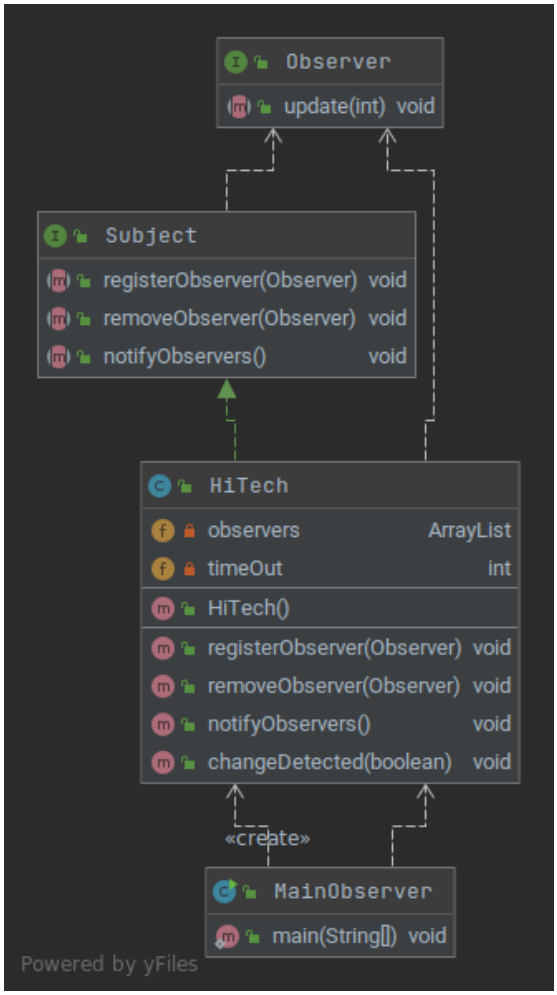
Implements HiTech class from Subject. And it contains the observer ArrayList. I have set it to be in Object reference type. In Consturctor, the timeout variable is assigned 60. The registerObserver () method adds the object to the list. The removeObserver () method removes the object from the list. The notifyObservers () method browses the list and calls each object's update method, and does this by passing the timeOut variable as a parameter. The changeDetected () method calls the notifyObservers () method as in the picture below.

```java
/**
 * It is the method by which the traffic situation is changed.
 * @param flag is true then traffic is anormal otherwise normal.
 */
public void changeDetected(boolean flag) {
    if(flag == true) {
        System.out.println("----------------------------");
        System.out.println("!!! Traffic is increased !!!");
        System.out.println("----------------------------");
        timeOut = 90;
    }
    else {
        System.out.println("--------------------------");
        System.out.println("!!! Traffic is normal !!!");
        System.out.println("--------------------------");
        timeOut = 60;
    }
    notifyObservers();
}
```

Likewise, the TrafficLights class implements Observer and the update method in it works as follows.

```java
@Override
public void update(int timeout_X) {
    this.timeout_x = timeout_X;
}
```

```
15 Seconds Wait ...
Red to Green switch ...

60 Seconds Wait ...
Green to Yellow switch ...

3 Seconds Wait ...
Yellow to Red switch ...

----------------------------
!!! Traffic is increased !!!
----------------------------
15 Seconds Wait ...
Red to Green switch ...

90 Seconds Wait ...
Green to Yellow switch ...

3 Seconds Wait ...
Yellow to Red switch ...

--------------------------
!!! Traffic is normal !!!
--------------------------
15 Seconds Wait ...
Red to Green switch ...

60 Seconds Wait ...
Green to Yellow switch ...

3 Seconds Wait ...
Yellow to Red switch ...
```



**QUESTION 4:**

**part A:**

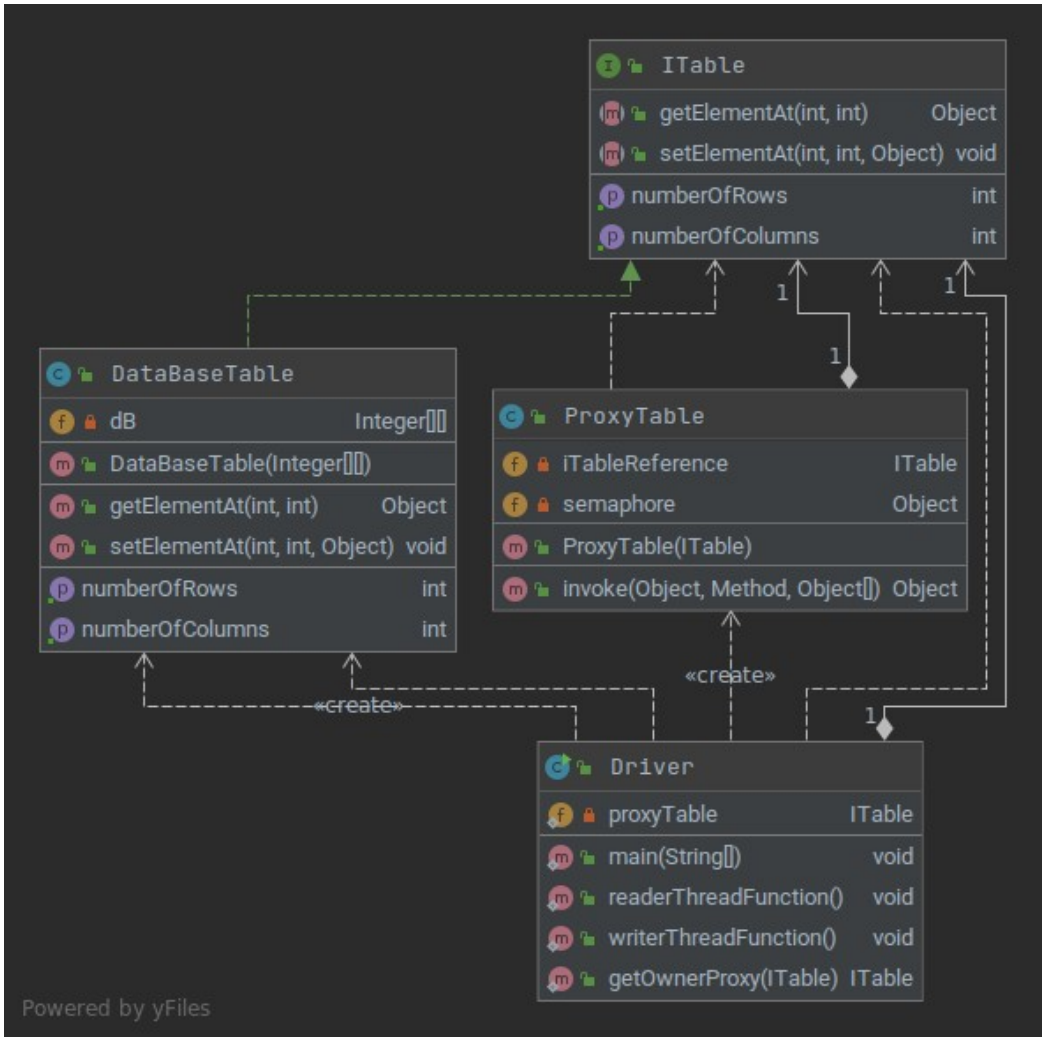
The proxy design pattern allows us to represent another class through a Proxy class we created. In this way, the Proxy class can perform all the operations or transactions it wants without interfering with the main class.

I designed the proxy design pattern using java proxy for this part. Here is the ITable interface in the question.

There are classes of DataBaseTable that implements it and ProxyTable classes that use it. One of these classes, DataBaseTable is the class whose source codes are unknown and simulated. ProxyTable is the class that allows the use of proxy design patterns. I would like to continue by explaining the properties of the Driver and ProxyTable class, since there is nothing in the other classes. The ProxyTable class implements the InvocationHandler class based on the Java Proxy instance in the processed book. The invoke () method is overridden. It should be noted which function of the object called in this method is used. That's why the getElementAt () and setElementAt () methods have been checked. Because these methods should be considered in the first part. Among these methods, synchronization is done with the synchronized keyword. Because in the first part, "Make sure no reader thread calls getElementAt while a writer thread is executing setElementAt." The sentence is carried out as desired. This is done with a semaphore variable with an Object reference. return method.invoke (this.iTableReference, args); By using the proxy class, the desired method is executed. Only these two methods are focused. Other methods in Interface run as they are.

In the Driver, 4 threads were created and it was specified to run the reader, writer, writer and reader functions respectively. Since the Java proxy is used, the static getOwnerProxy () method has also been added. Proxy operation is done with this method.

```
Thread-1  : Set Method Starts...
Thread-1  : Set Method is finished
Thread-2  : Set Method Starts...
Thread-2  : Set Method is finished
Thread-0  : Get Method Starts...
Thread-0  : Get Method Result : -10
Thread-3  : Get Method Starts...
Thread-3  : Get Method Result : -10

Process finished with exit code 0
```

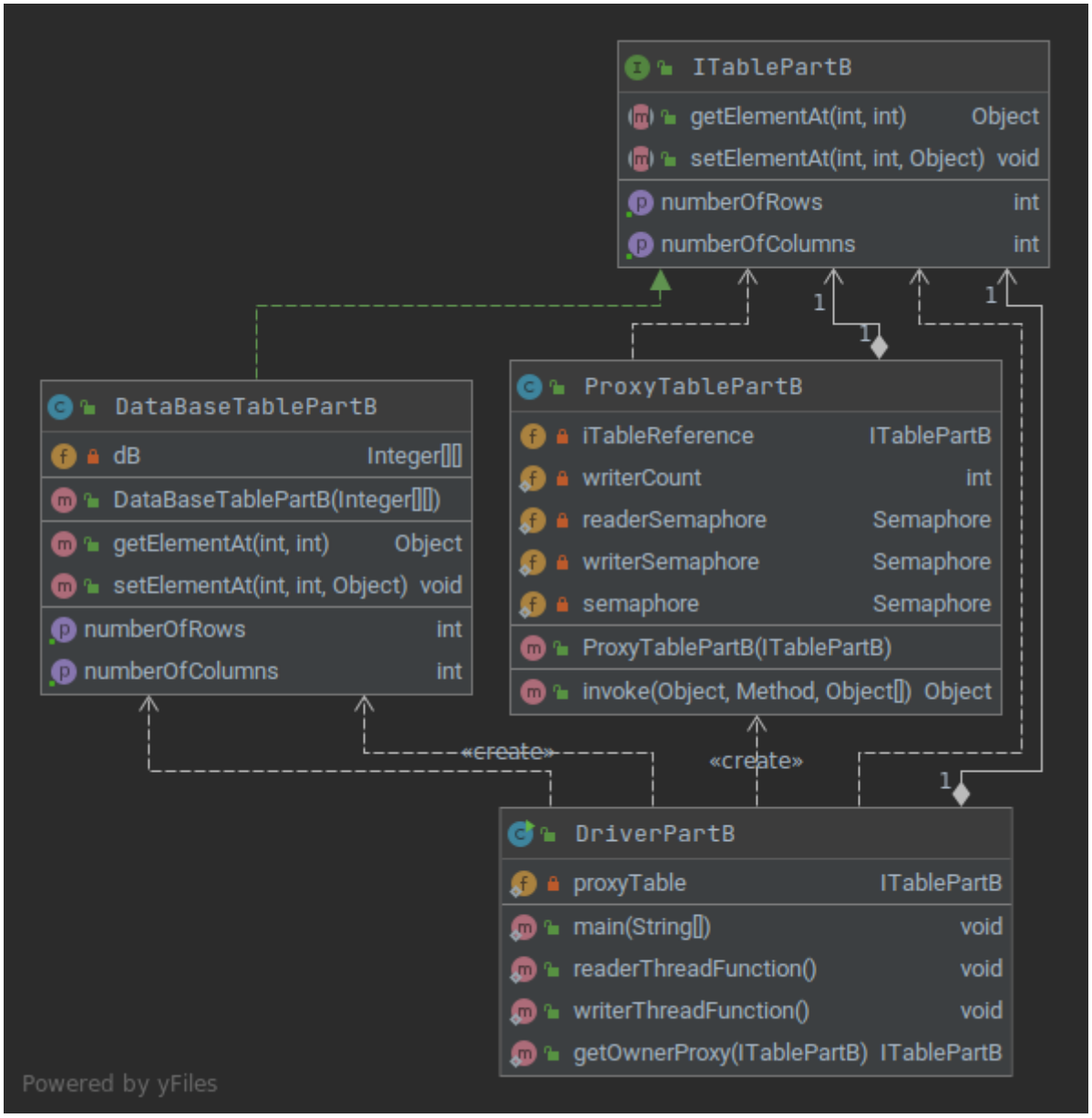



**part B:**

The priority order for Part B and this is the subject covered in the System Programming course last semester. The source code is almost the same, only the ProxyTablePartB class and Driver are different. Likewise, Java proxy is used as in part A. Issues that differ from those described above will be mentioned. Here the invoke () method is different. Because now the threads running the writer function are given priority. The slides used in this system programming course were used. A variable holding the number of 3 semaphore and writers is used. Since the Java proxy feature return is done, the corresponding semaphore object references are freed using the finally block. Thus, if writer threads are available, reader threads will not work. Sometimes, in the output results, it can be seen that the reader threads are running before the writer thread runs. This is normal. Many threads have been tried in Driver.

*Write-preferring RW locks* avoid the problem of writer starvation by preventing any *new* readers from acquiring the lock if there is a writer queued and waiting for the lock; the writer will acquire the lock as soon as all readers which were already holding the lock have completed. The downside is that write-preferring locks allows for less concurrency in the presence of writer threads, compared to read-preferring RW locks. Also the lock is less performant because each operation, taking or releasing the lock for either read or write, is more complex, internally requiring taking and releasing two mutexes instead of one. This variation is sometimes also known as "write-biased" readers–writer lock.

```
Thread-0  : Set Method Starts ...
Thread-0  : Set Method Execute ...
Thread-0  : Set Method Finished ...
Thread-1  : Set Method Starts ...
Thread-1  : Set Method Execute ...
Thread-1  : Set Method Finished ...
Thread-2  : Set Method Starts ...
Thread-3  : Set Method Starts ...
Thread-2  : Set Method Execute ...
Thread-2  : Set Method Finished ...
Thread-4  : Set Method Starts ...
Thread-5  : Get Method Starts ...
Thread-4  : Set Method Execute ...
Thread-4  : Set Method Finished ...
Thread-6  : Get Method Starts ...
Thread-6  : Get Method Execute ...
Thread-6  : Get Method Finished ...
Thread-6  : Get Method Result : -10
Thread-3  : Set Method Execute ...
Thread-3  : Set Method Finished ...
Thread-5  : Get Method Execute ...
Thread-5  : Get Method Finished ...
Thread-5  : Get Method Result : -10
Thread-7  : Get Method Starts ...
Thread-7  : Get Method Execute ...
Thread-7  : Get Method Finished ...
Thread-7  : Get Method Result : -10
Thread-8  : Get Method Starts ...
Thread-8  : Get Method Execute ...
Thread-8  : Get Method Finished ...
Thread-8  : Get Method Result : -10

Process finished with exit code 0
```


ITablePartB
getElementAt(int, int) Object
setElementAt(int, int, Object) void
numberOfRows int
numberOfColumns int
DataBaseTablePartB
dB Integer[][]
DataBaseTablePartB(Integer[][])
getElementAt(int, int) Object
setElementAt(int, int, Object) void
numberOfRows int
numberOfColumns int
ProxyTablePartB
iTableReference ITablePartB
writerCount int
readerSemaphore Semaphore
writerSemaphore Semaphore
semaphore Semaphore
ProxyTablePartB(ITablePartB)
invoke(Object, Method, Object[]) Object
DriverPartB
proxyTable ITablePartB
main(String[]) void
readerThreadFunction() void
writerThreadFunction() void
getOwnerProxy(ITablePartB) ITablePartB
«create»
«create»
1
1
1
1
Powered by yFiles