DESIGN PATTERNS HW2

Betülnaz Hayran-28354853660 Github:Naz-bh

Batuhan Kesikbaş-40573251614 Github:BatuhanKesikbas

1-Statement of Work

Playing Moderator-Game, the number of players is 5 by default and after that all the players will get 10 numbers randomly. Then the game starts with moderators generated numbers, moderator will generate 10 numbers randomly. If these generated numbers and the numbers assigned to the players match at least 3 times, the players are going to win. There can be more than one winner but also there can be no winner as well. In the end 'Score Table' is printed and each players' score can be viewed.

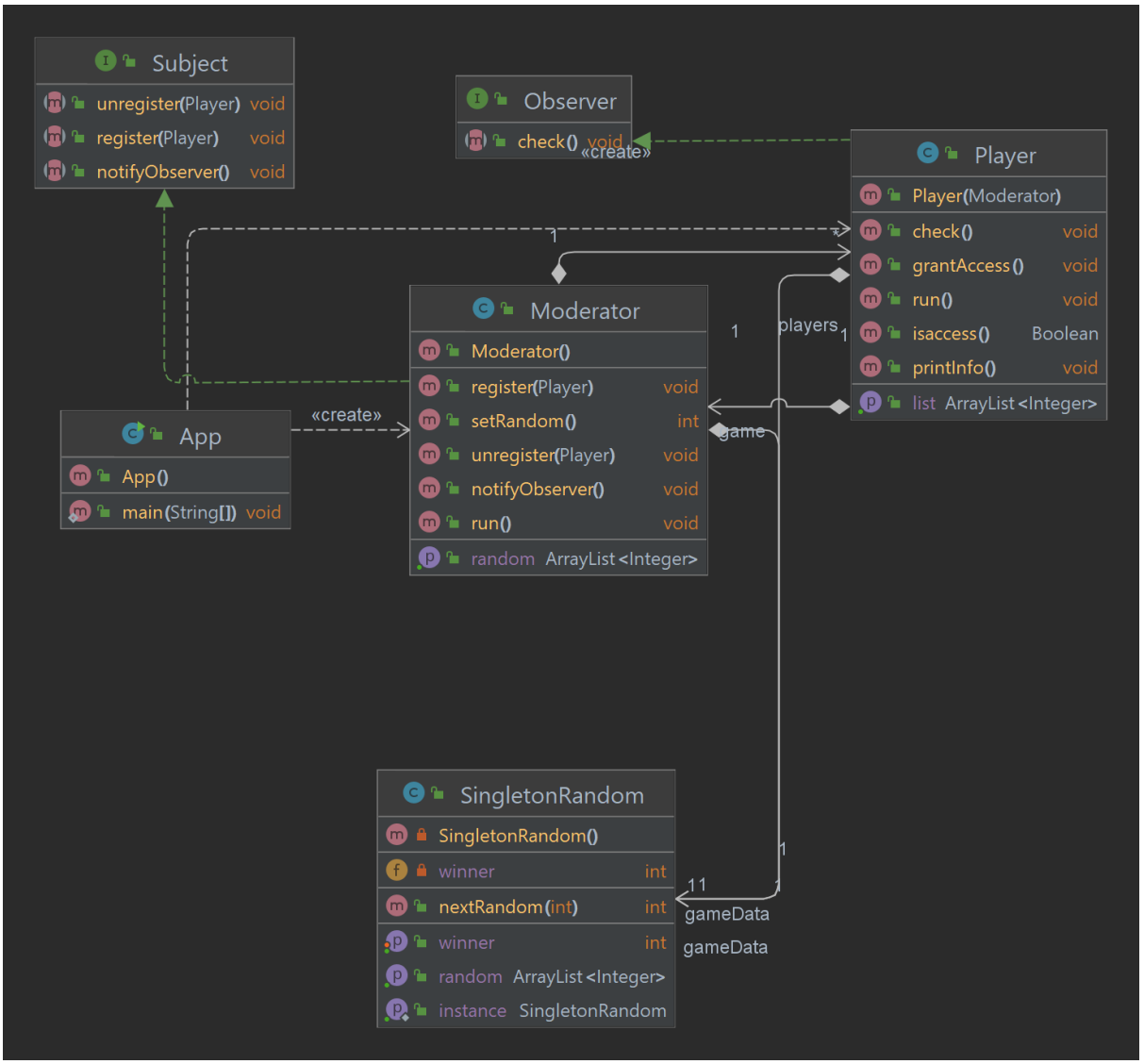
The problem: We need to relate moderator and players also we need to use shared data between players as well as moderator.

2- Explanation on Utilized Design Patterns

The Observer Pattern is used when there is one-to-many relationship between objects.
In our scenario, we implemented Moderator-player relation using observer pattern.

The Singleton Pattern is a design pattern that restricts the instantiation of a class to one object. In our case, we used shared data between players as well as moderator. For this we create a singleton class and use the only instance of it.

3- UML Class Diagram



4- Research

4.1-The Observer Pattern

```
import java.util.ArrayList;
import java.util.List;

public class Subject {

  private List<Observer> observers = new ArrayList<Observer>();
  private int state;

  public int getState() {
```

```
    return state;
  }

  public void setState(int state) {
    this.state = state;
    notifyAllObservers();
  }

  public void attach(Observer observer){
    observers.add(observer);
  }

  public void notifyAllObservers(){
    for (Observer observer : observers) {
      observer.update();
    }
  }
}
   public abstract class Observer {
  protected Subject subject;
  public abstract void update();
}
public class BinaryObserver extends Observer{

  public BinaryObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }

  @Override
  public void update() {
    System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
  }
}
public class OctalObserver extends Observer{

  public OctalObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
```

```
  }

  @Override
  public void update() {
    System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
  }
}
public class HexaObserver extends Observer{

  public HexaObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }

  @Override
  public void update() {
    System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );
  }
}
public class ObserverPatternDemo {
  public static void main(String[] args) {
    Subject subject = new Subject();

    new HexaObserver(subject);
    new OctalObserver(subject);
    new BinaryObserver(subject);

    System.out.println("First state change: 15");
    subject.setState(15);
    System.out.println("Second state change: 10");
    subject.setState(10);
  }
}
//Output:
//First state change: 15
//Hex String: F
//Octal String: 17
//Binary String: 1111
//Second state change: 10
```

```
//Hex String: A
//Octal String: 12
//Binary String: 1010
```

How to Implement

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class Observer and a concrete class Subject that is extending class Observer.

ObserverPatternDemo, our demo class, will use Subject and concrete class object to show observer pattern in action

Steps:

1. Create Subject class.
2. Create Observer class.
3. Create concrete observer classes (BinaryObserver, OctalObserver, HexaObserver.java).
4. Use Subject and concrete observer objects (ObserverPatternDemo.java).

Source: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

4.2-The Singleton Pattern

```
public class SingleObject {
  private static SingleObject instance = new SingleObject();
  private SingleObject(){}

  public static SingleObject getInstance(){
    return instance;
  }
```

```
   public void showMessage(){
      System.out.println("Hello World!");
   }
}
public class SingletonPatternDemo {
   public static void main(String[] args) {

      SingleObject object = SingleObject.getInstance();
      object.showMessage();
   }
}
//Output: Hello World!
```

How to Implement

We're going to create a SingleObject class. SingleObject class have its constructor as private and have a static instance of itself.

SingleObject class provides a static method to get its static instance to outside world. SingletonPatternDemo, our demo class will use SingleObject class to get SingleObject object.

Steps:

1. Create a Singleton Class.
2. Get the only object from the singleton class.(SingletonPatternDemo.java)

Source: https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

5-Implementation codes