# Final Project – A Monopoly™ game

## 1. Introduction

For the *« Design Pattern & Software Development Process »* course, we carried out this project as a group of 2, Yanis DAHMOUCHE and François COUTAU. The goal of the task is to simulate a simplified version of the famous Monopoly game. You will find our code on github here .

### Rules

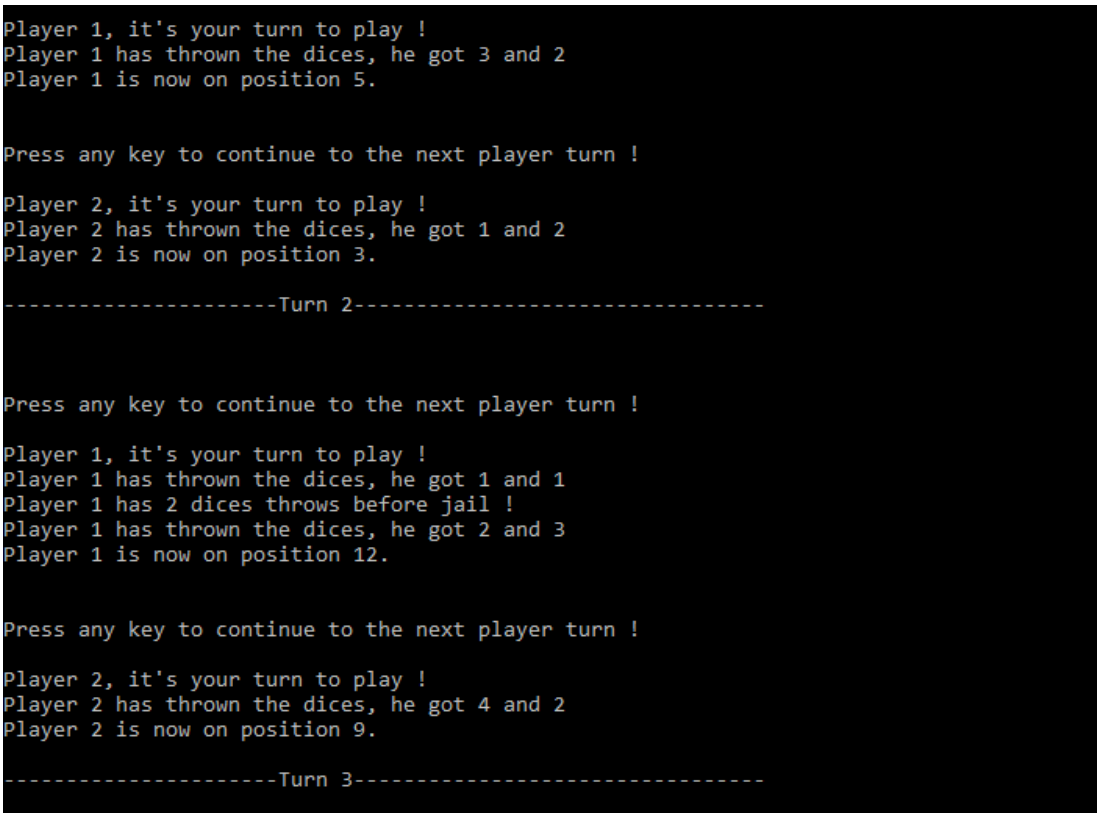
In this game, the players can only move throughout the board, which consists of 40 blocks. Each player has 2 dices in his hands, which he throws at every turn and moves forward of the sum indicated by the 2 dices. Players can go to jail if they throw the dices with the same value 3 times in a row or if they land on box number 30 (the « go to jail » box). They stay in prison for 3 turns unless they manage to throw the dices with the same value.

### Concrete objective

The program needs to be implemented with some design patterns, as best implemented as we can and in a useful way.

## 2. Design Hypotheses

There were no specifications on the interface we had to do. So we decided to make a simple console application and update with a comments every time someone does something.




This allowed us to implement the observer pattern which, as its name indicated, observes every action done throughout the game and lets the user know by writing each action to the console.

We also implemented a View for the Monopoly game, this allows us to print a recap of the game whenever we desire, it looks like that :

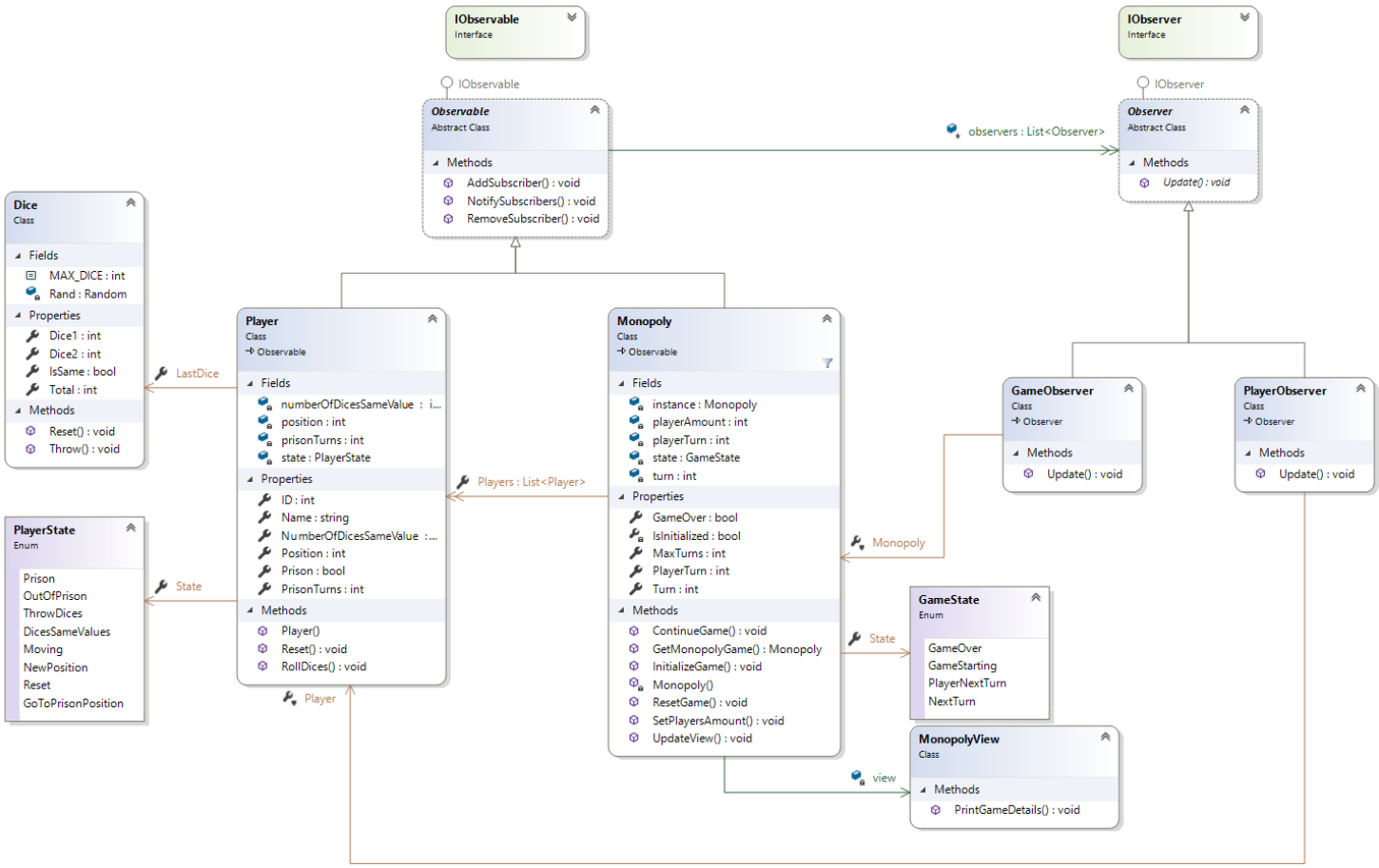
```
****************** GAME UPDATE ******************

Turn : 3/2147483647  ||  Players : 2

 | ID | Name     | Position | Prison? |
 |-----------------------------------|
 | 1  | Player 1 | 20       | False   |
 | 2  | Player 2 | 23       | False   |


****************** END UPDATE ******************
```

The assignment also let us decide which programming languages we could use so we chose *C#*.

Other than that, the assignment wasn't pretty detailed on the way to do things so we were pretty free to do things as we wanted to.

# 3. UML diagrams

## 3.1 Class diagram of the solution



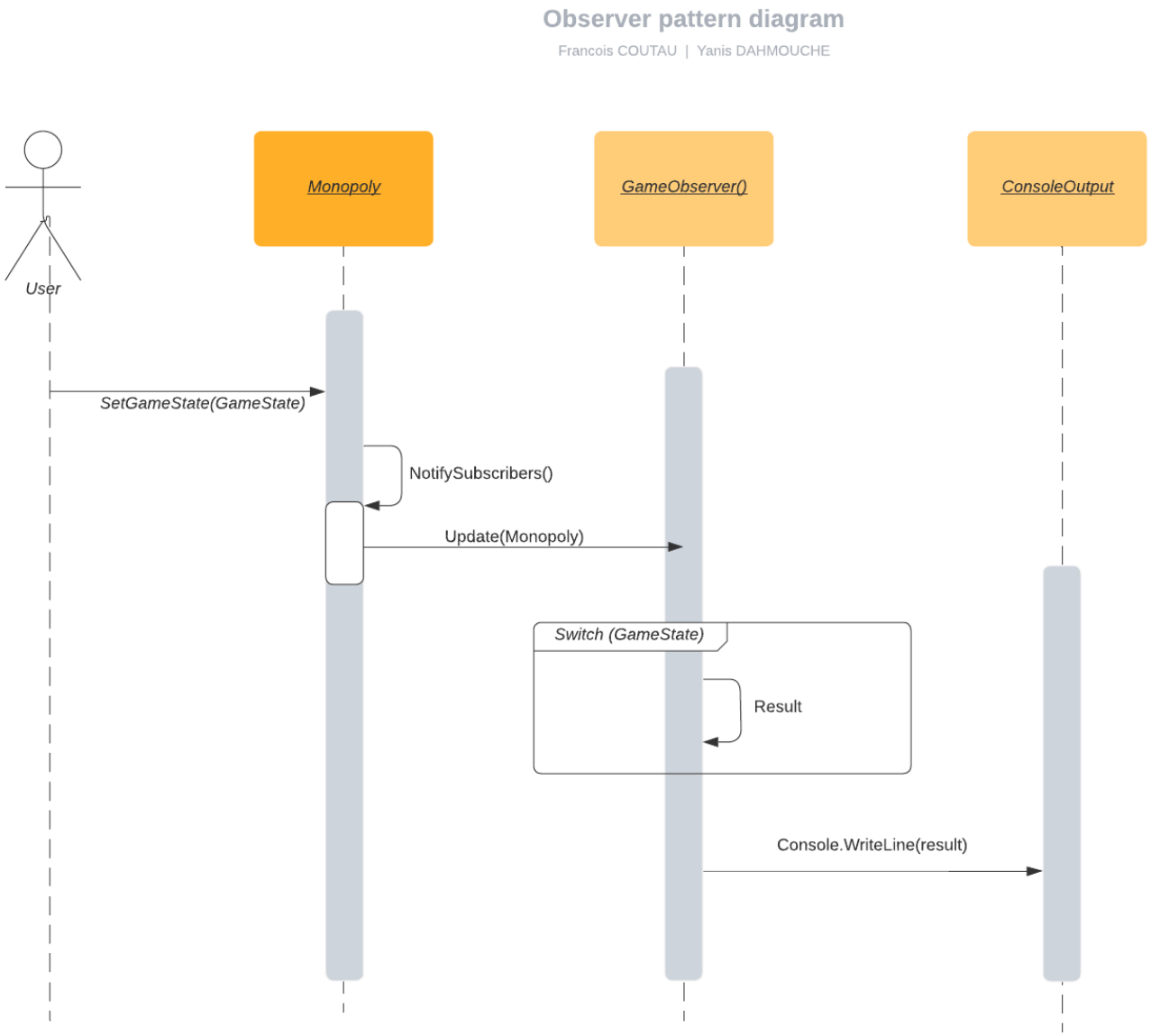

This is our class diagram.

As you can see we have implemented two design patterns for this project. The observable pattern which allows us to see and inform the user every time something happens during the game. It is very useful to better control to one place what will be displayed on the screen instead of looking at 'Console.WriteLine's dispatched through the whole code, which can get messy very fast.

We also used the Singleton pattern to make sure only 1 instance of the monopoly game could run at once.

## 3.2 Sequence diagram

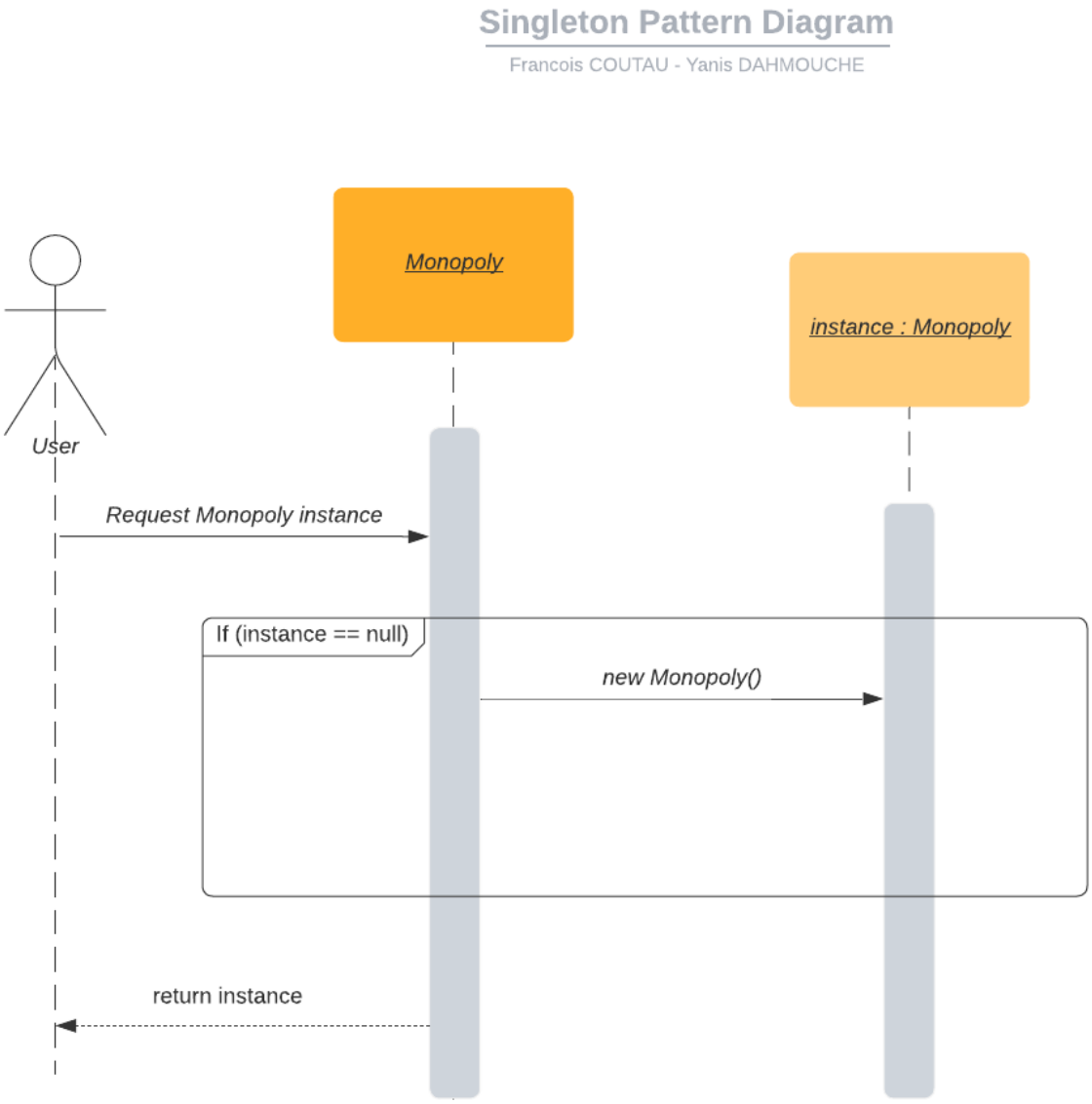
### 3.2.1 Observer Pattern Sequence Diagram




Here is the observer pattern for our Monopoly class. Usually the Monopoly instance would change its status itself, but here we showed the user changing it for better clarity. When the *State* property of Monopoly instance is changed, it calls the *NotifySubscribers()* function which then Updates the gameobserver with itself. The gameobserver then choses the appropriate result according to the *State* of the game. Then itw rites the result to the console.

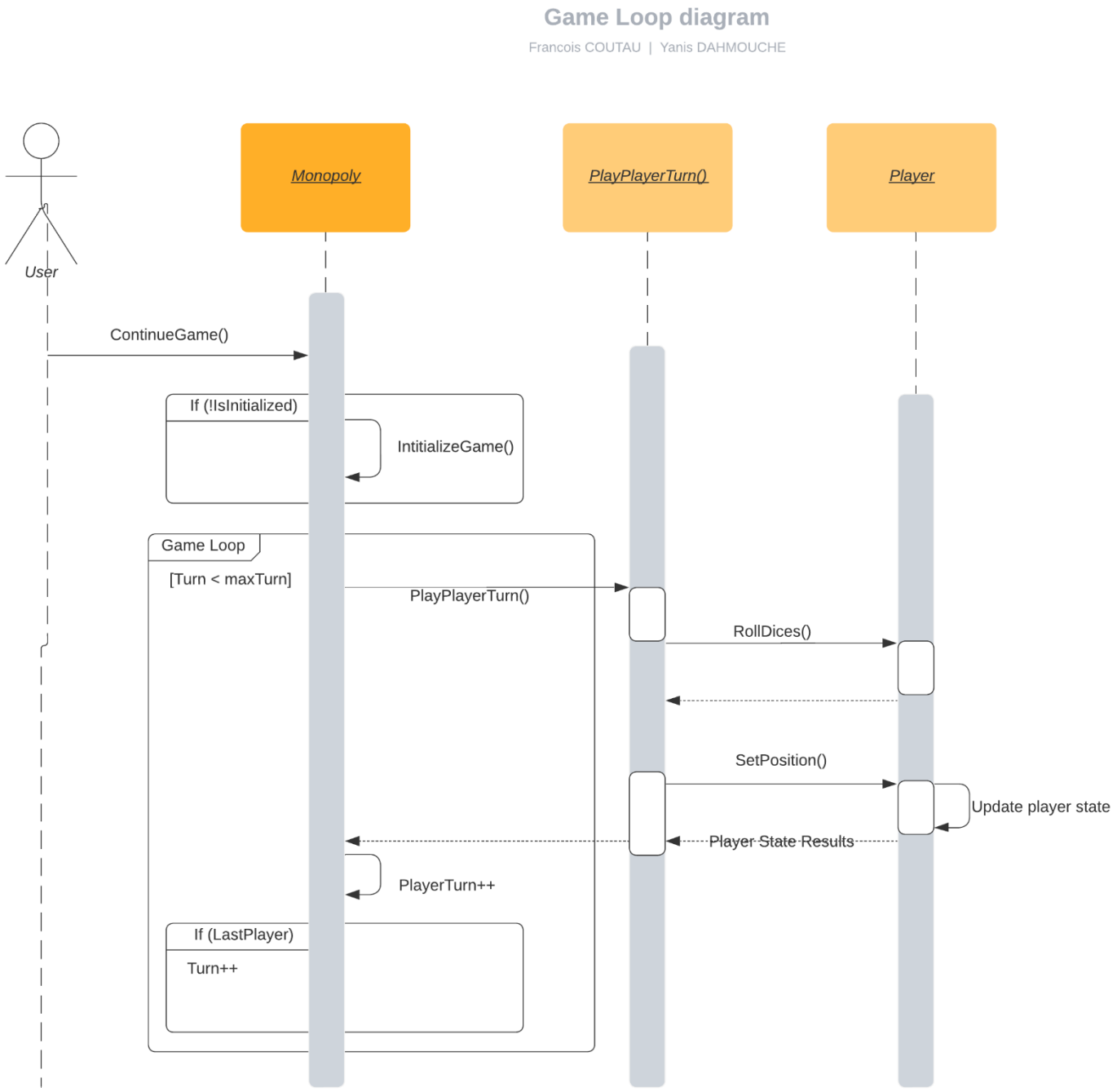
The diagram is also valid for Player and PlayerObserver which work the same way.

### 3.2.2 Singleton Pattern Sequence Diagram




This is the Singleton pattern of our project shown as a sequence diagram. Very simple, it works as intended. We look if the instance is already created and sen dit back to the user, if not, we create it then we sen dit back.

### 3.2.3 Game Loop Sequence Diagram




This is the sequence diagram for our main game loop. The loop goes on for the remaining turns, and each time we call the *PlayPlayerTurn()* function which deals with everything related to the player's turn, as expected. The Player's State is automatically updated when the position property is changed. It triggers all kind of events related to the prison or to the observer pattern. When the loop is over, the function does not return anything to the User, except all the properties Monopoly already contains.

# 4. Test cases

We used the built-in unit tester in Visual Studio to make the test-cases. We tested 4 functions and built-in functionnalities and design-patterns.

## 4.1 Monopoly singleton test

```
[TestMethod]
0 references
public void Monopoly_Only1_Instance()
{
    Monopoly mon = Monopoly.GetMonopolyGame();

    Monopoly mon2 = Monopoly.GetMonopolyGame();

    Assert.IsTrue(mon == mon2);
}
```

Test to check if there is only 1 instance of the monopoly class. We check the reference of the 2 objects.

## 4.2 Monopoly observer test

```
[TestMethod]
0 references
public void Game_Observer_Test_Console_Message()
{
    var consoleOut = new StringWriter();
    Monopoly mon = Monopoly.GetMonopolyGame();
    GameObserver obs = new GameObserver();
    mon.AddSubscriber(obs);

    Console.SetOut(consoleOut);
    mon.InitializeGame();

    Assert.AreEqual($"MonopolyQuickConsoleGame.PlayerObserver has just subscribed to a Player instanceMonopolyQuickConsoleGame.PlayerObs
        $"has just subscribed to a Player instance" +
        $"************************************ MONOPOLY *********************************" +
        "********************** The game is going to start ... Get ready ! ************************",
        Regex.Replace(consoleOut.ToString(), @"\r\n?|\n", ""));

}
```

Test to check whether the game observer writes in the console what is expected when the game is initialized. We intercept the console output by changing the stream output to our StringWriter() object. Then we just have to change the *State* of the game which will then activate the Observer pattern and display, if everything is fine, what is expected to the console.

### 4.3 Player observer test

```
[TestMethod]
0 references
public void Player_Observer_Test_Console_Message()
{
    var consoleOut = new StringWriter();
    Player pl = new Player("Player", 1);
    PlayerObserver obs = new PlayerObserver();
    pl.AddSubscriber(obs);

    Console.SetOut(consoleOut);
    pl.Position = 10;

    Assert.AreEqual($"{pl.Name} is now on position {pl.Position}.",
        Regex.Replace(consoleOut.ToString(), @"\r\n?|\n", ""));

}
```

Test to check whether the player observer writes in the console what is expected when the player's position is changed. Same as the last test, we intercept the console output after changing the Player's *State*, then we compare with what is expected.

### 4.4 Dice test

```
[TestMethod]
0 references
public void Dice_Total()
{
    Dice dice = new Dice();

    dice.Throw();

    Assert.AreEqual(dice.Dice2 + dice.Dice1, dice.Total);
}
```

Test to check whether the 'Total' property of the Dice class returns the expected value

## 5. Conclusion

This project was very instructive as it forced us to make the cleanest code we could do and the easiest to understand for anyone looking at it. We only used 2 defined design-patterns (we did use a view pattern for our main Monopoly class but it's not very interesting), the Observer pattern and the Singleton pattern. We don't feel like we could have used any more patterns, indeed we did have some ideas and we could have implemented some others but that would not have been very useful and would have only added unnecessary complexity to the project.