LET'S PLAY MONOPOLY

Ugo DEMY & Adrien FIGARD – DIA4

https://github.com/Nibleash/Monopoly

Rules & Introduction

Simulate a simplified version of the Monopoly™ game

A set of **Players** is given, each with *name* and initial *position*. **Dices** are *rolled* and players' positions on the **Game Board** will change. The game is played on a circular game board, composed of 40 positions on the board, indexed from 0 to 39. If a player reaches position 39 and still needs to move forward, he'll continue from position 0. In other words, positions 38, 39, 0, 1, 2 are contiguous.

Each player rolls two dices and *moves* forward by a number of positions equal to the sum of the numbers told by the two dice. A player turn ends after moving. The same position can be occupied by more than one player. If a player gets both dice with the same value, then he rolls the dice and moves again. If this happens three times in a row, the player goes to jail and ends his turn.

Jail can be only visited or be a situation the player is in. The board has a Visit Only / In Jail at position 10 and a **GoToJail** at position 30. If at the end of a basic move, the player lands on GoToJail, then he immediately moves to the position Visit Only / In Jail and is in jail. His turn ends If after moving, the player lands on Visit Only / In Jail, he is visiting only and is not in jail. While the player is in jail, he still rolls the dice on his turn as usual, but does not move until either:

1. He gets both dices with the *same value*.
2. He fails to roll both dice with the same value for three times in a row (i.e., his previous two turns after moving to jail and his current turn).

If either 1. or 2. happens in the player's turn, then he moves forward by the sum of the dice rolled positions and his turn ends. He does not roll the dice again even if he has rolled a both dice with the same value.

Design Hypotheses

Main classes

While reading the instructions, we first wanted to identify **how to construct our Monopoly**. This means we had to look after the main classes and functions we will have to implement. To do so, we put in bold the main classes, italic the important functions and underline the principal conditions as you can see above.

Dices.cs is the class we use to create a pair of dices. Always going by 2 since you can't throw only one dice, we also added an attribute *Sum* computing the sum of the dices. There are **3** main functions in **Dices:**

- *ThrowDices()* which throws the dices one by one randomly with each having values between 1 and 7 and sums it once done.
- *AreEquals()* which checks whether the dices are equals or not, to enable replay if double, and returns a Boolean.
- *toString()* which returns a String gathering the dices' results and the sum to display it. As in **Player**.*Move()*, we use here *System.Sleep* to mimic the real dices throwing.

Player.cs is the class we use to create a player. A player is made of various attributes indicating its *position* on the **Board**, the *distance* covered since the beginning and its jail and profile info (*pseudo*, *rounds_in_jail*, etc.). There are **2** main functions in **Player**:

- *Move()* which moves the player on **Board** by modifying its *position* and its travelled *distance* appropriately based on the total of the **Dices** thrown.
- *toString()* which returns a String gathering the player's name and pseudo for the welcome screen or to announce the winner.

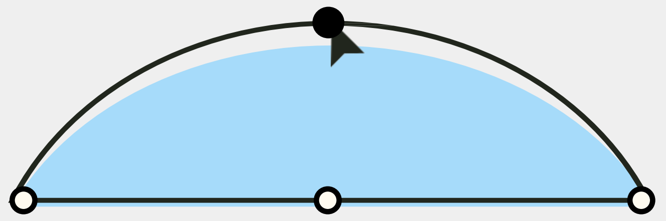
Board.cs is the class we use to create a board. The main specificity of a **Board** instance lies in its **Square** composition. Indeed, a board is made of a 40-squares array itself composed of 6 types of **Squares**: Normal, Start, Luck, Community, GoToJail and Jail. We are precising where each type is in the **Board** constructor. Moreover, as for the **Dices**, our **Board** class has a *Singleton* implemented in it to ensure its unicity.

Game.cs is our main class. It gathers instances from the 3 previous classes as attributes in addition to a *round counter* and *limitation*. There are **4** main methods in **Game:**

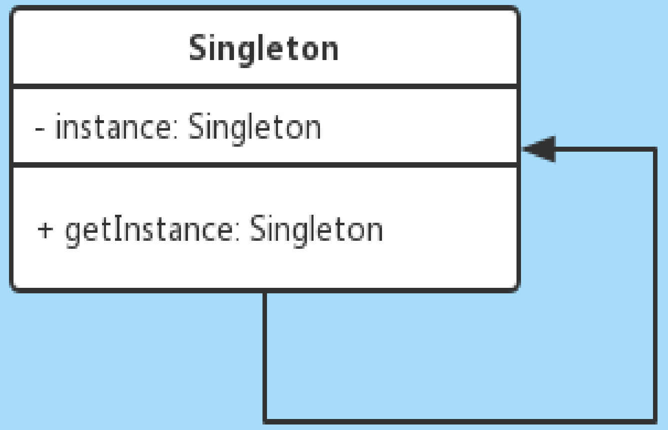
- *Create()* which allows the user to create up to 6 players, adding them to the **PlayerIterator** with a *name* and a *pseudo*.
- *Start()* which is the main core of the game. It iterates over the players and the rounds using a double while loop, treat special cases and uses the **ISquare** to display the **Square** specificities.
- *Jail_Procedure()* which makes the **Player** automatically throw the **Dices** when in jail except if *nbdouble* is 3. If double, the player escapes.
- *EndGame_Procedure()* which displays winner and end-game messages.

Design Hypotheses

Implemented Design Patterns



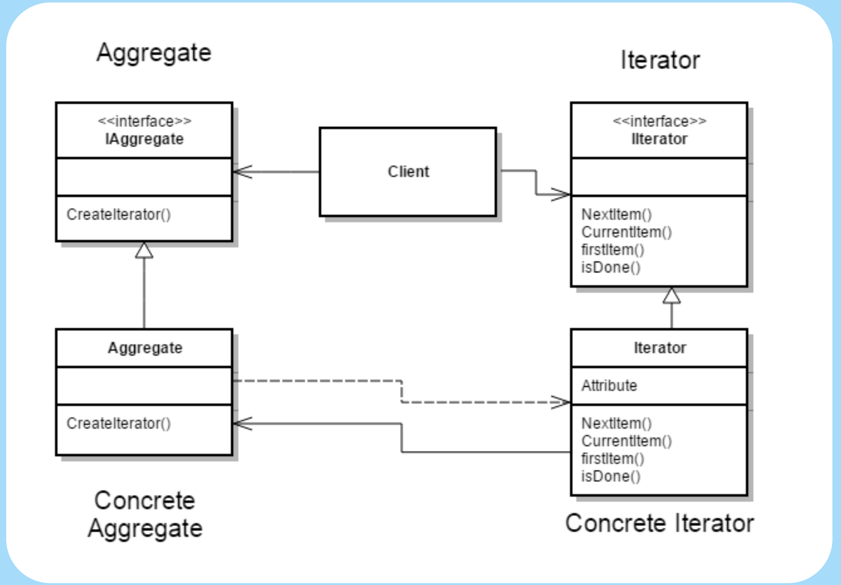
C# Singleton



Another point to consider is that we wanted our program to mimic the real Monopoly. Meaning that, we only have **1 Board**, **1 set of Dices** and **several Players** belonging to **1 unique Game**. This is the reason why we implemented our first design pattern: The *Singleton*.

The *Singleton* design pattern ensures a class has only one instance and provide a global point of access to it. Thus, to meet our specifications, we implemented the *Singleton* design pattern of the **Board** and **Dices** classes.

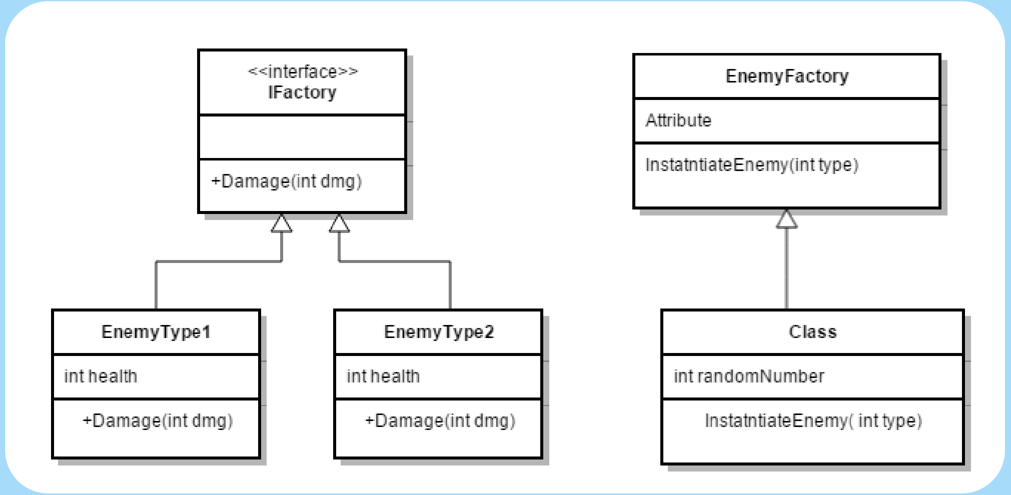
C# Iterator



Concerning the **Player** class, what we aimed at was the **Game** class being able to access one by one to its players' properties without directly accessing a **Player** instance. Thus, we implemented an Iterator design pattern over the **Player** class, the client here being the **Game** class.

The *Iterator* design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

C# Factory Method

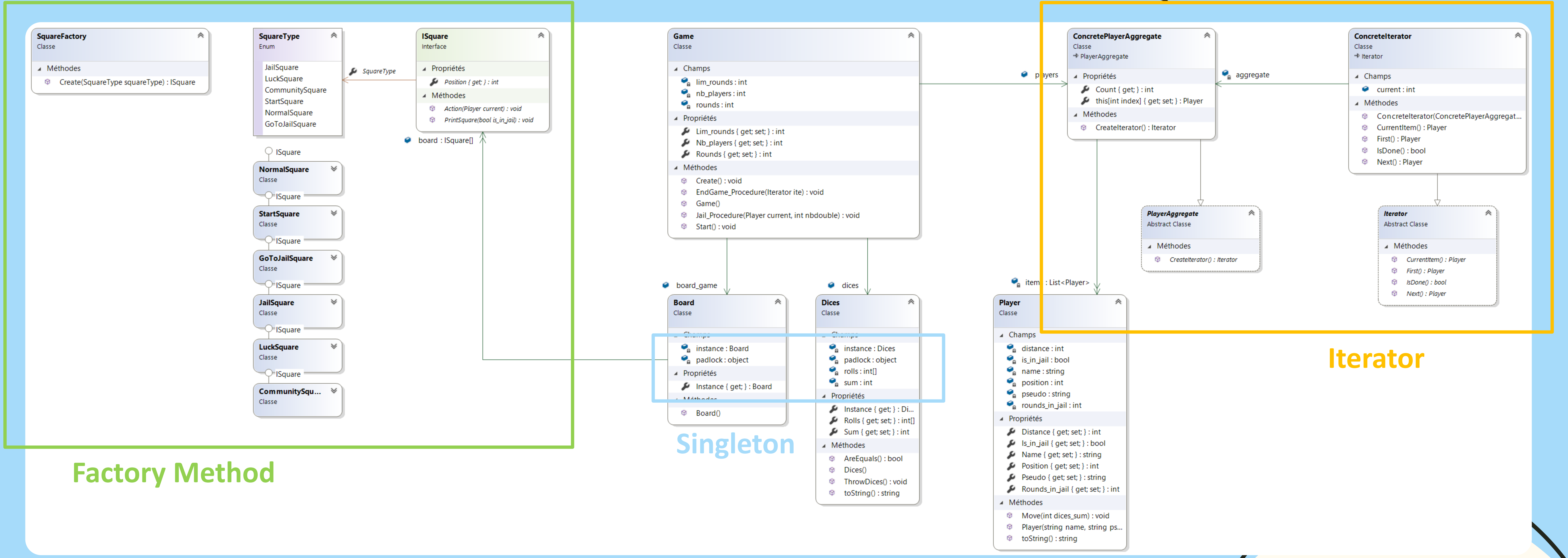


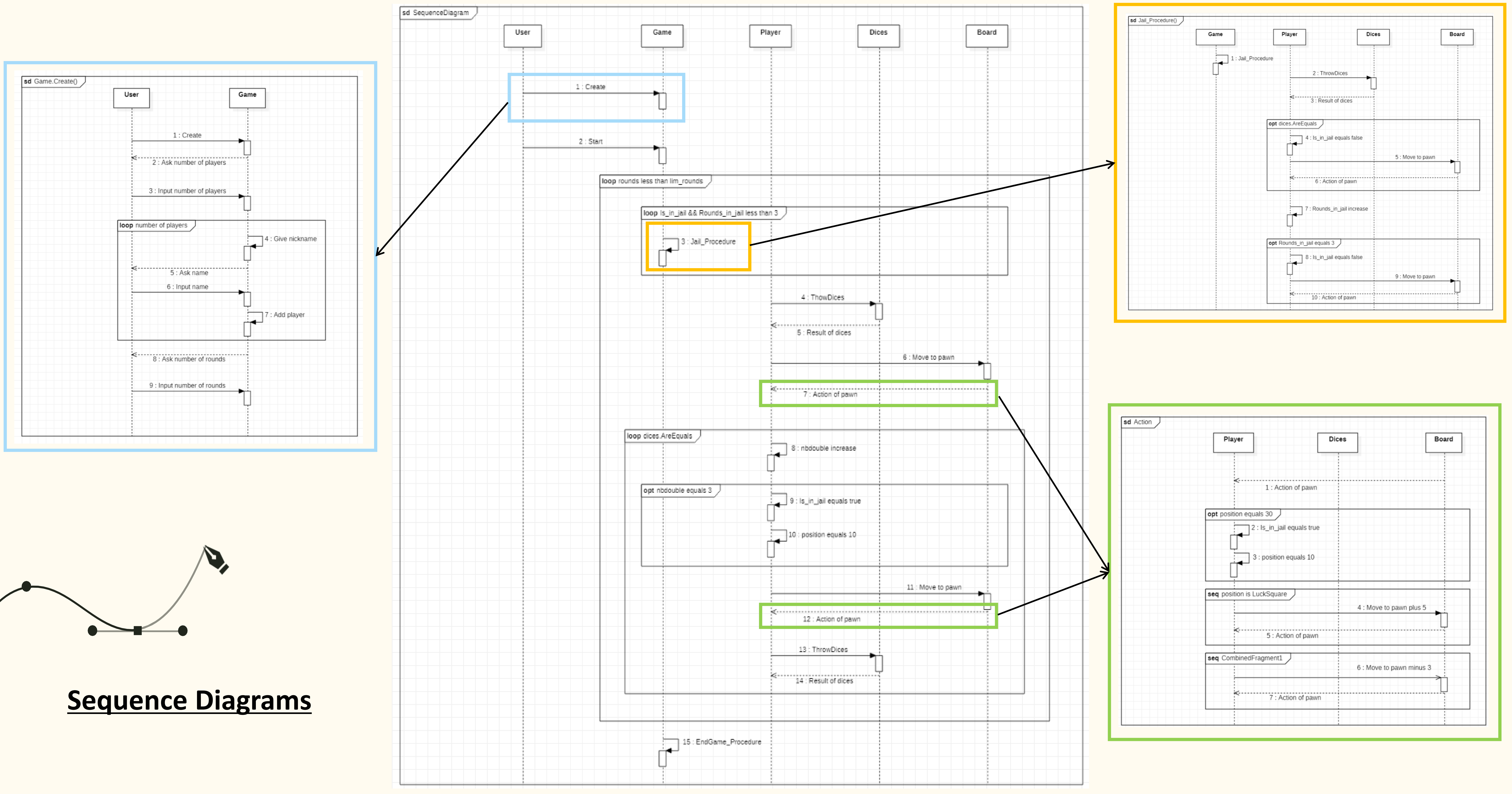
We finally chose to use the *Factory Method* design pattern in order to create **Squares** more efficiently in our Monopoly **Board**. This includes 6 different types of **Squares** each having actions when you step on it and clean user display.

Factory Method is a Design Pattern which defines an interface for creating an object but lets the classes that implement the interface decide which class to instantiate. It is used to replace class constructors, abstracting the process of object generation to allow type determination at run-time.

UML Diagrams – Class & Sequence

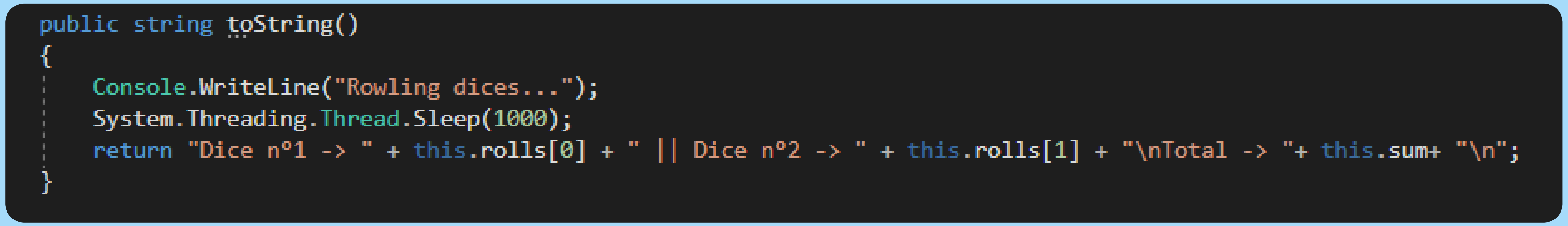
Class Diagram of the solution





Sequence Diagrams

Test cases



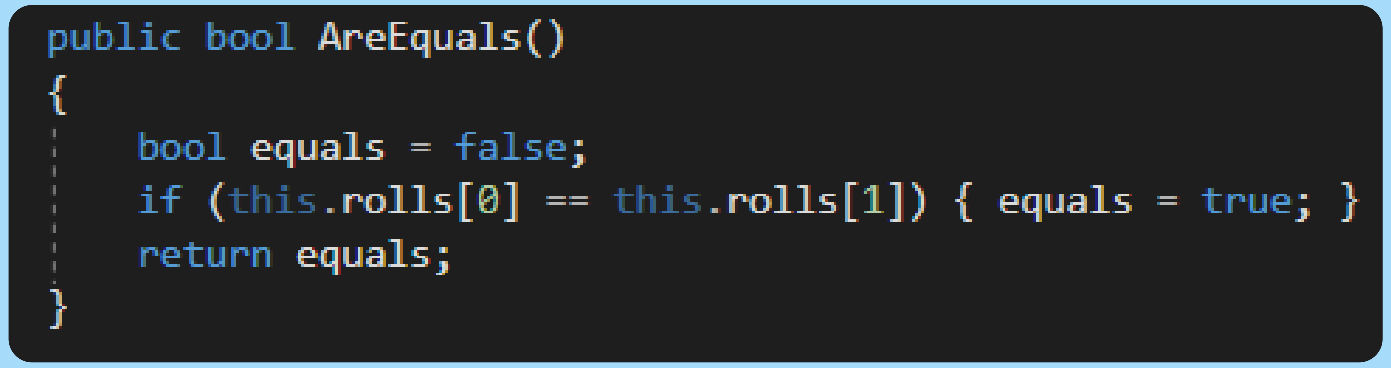
```
public string toString()
{
    Console.WriteLine("Rowling dices...");
    System.Threading.Thread.Sleep(1000);
    return "Dice n°1 -> " + this.rolls[0] + " || Dice n°2 -> " + this.rolls[1] + "\nTotal -> "+ this.sum+ "\n";
}
```

Overview

Test cases are a set of actions performed on a system to **determine if it works correctly**. The purpose of a test case is to determine if different features within a system are performing as expected and to confirm that the system satisfies all requirements. Furthermore, the process of writing a test case and executing it can also help reveal errors or defects within the system.

During the conception of our solution, we chose to use mostly functions that return nothing (void functions) because it made our code much clearer by not putting too many unnecessary variables into the larger functions like **Game**.*Start()*.

Hence, we had just a few non-void functions left, which we are going to test using the implemented unit testing tool of Visual Studio.



```
public bool AreEquals()
{
    bool equals = false;
    if (this.rolls[0] == this.rolls[1]) { equals = true; }
    return equals;
}
```

How ?

To test these two functions using Visual Studio 2017, we will start by adding a new project to our solution: a unit testing project called MonopolyTest.

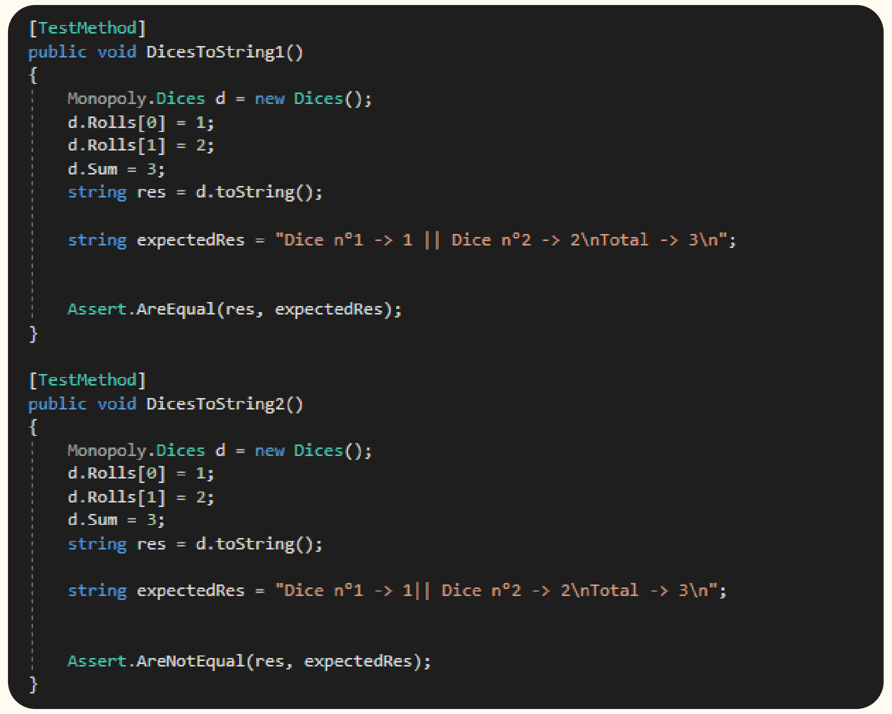
After doing so, we will just add the Monopoly project we have been working on as a reference of this newly created project, to be able to call its different classes.

Finally, we will design various methods to test our previous functions.

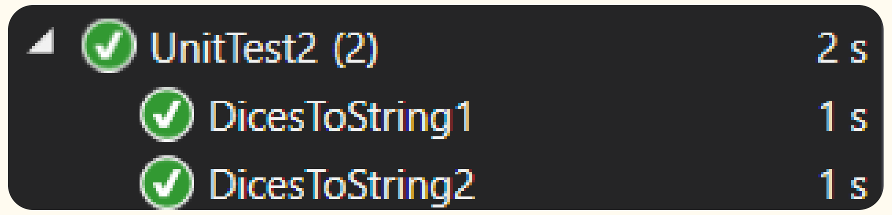
Test cases

2 examples of Test cases

To test the function *toString()* of the **Dices** class, we will create various strings with different values. The first one will be the expected result, and the second one will be missing just one blank space. Then we will see if our function works as expected by executing these tests.

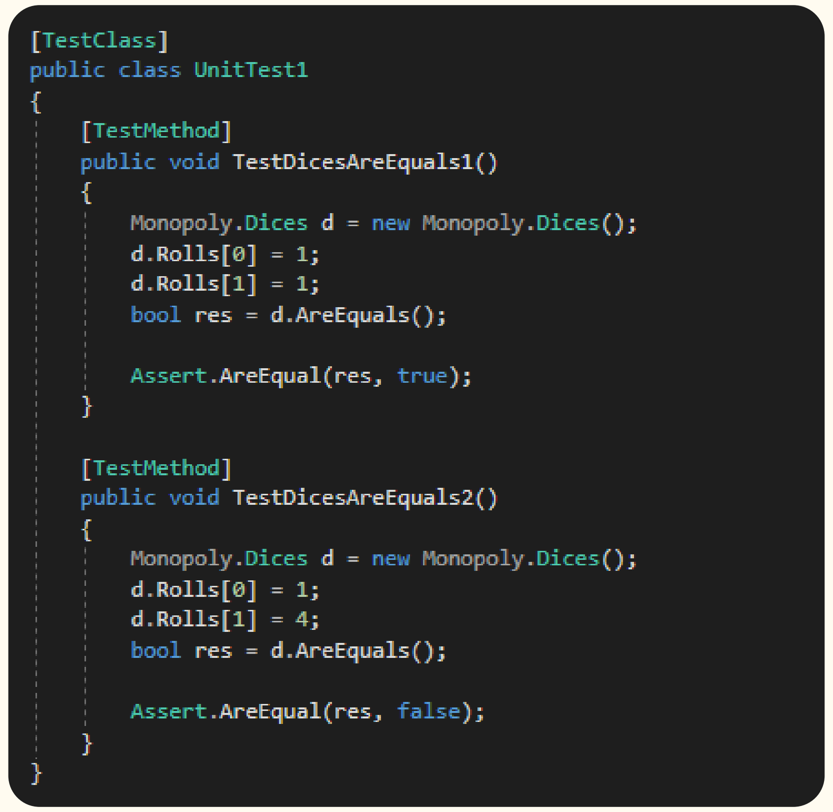


```
[TestMethod]
public void DicesToString1()
{
    Monopoly.Dices d = new Dices();
    d.Rolls[0] = 1;
    d.Rolls[1] = 2;
    d.Sum = 3;
    string res = d.toString();

    string expectedRes = "Dice n°1 -> 1 || Dice n°2 -> 2\nTotal -> 3\n";


    Assert.AreEqual(res, expectedRes);
}

[TestMethod]
public void DicesToString2()
{
    Monopoly.Dices d = new Dices();
    d.Rolls[0] = 1;
    d.Rolls[1] = 2;
    d.Sum = 3;
    string res = d.toString();

    string expectedRes = "Dice n°1 -> 1|| Dice n°2 -> 2\nTotal -> 3\n";


    Assert.AreNotEqual(res, expectedRes);
}
```

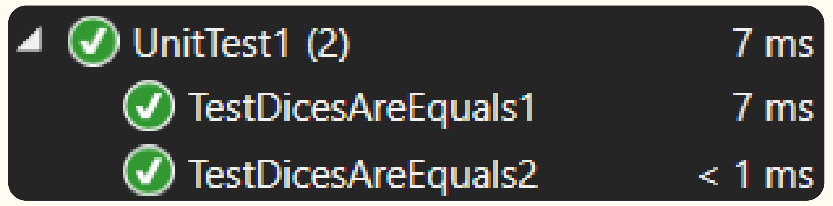


```
◢ ✓ UnitTest2 (2)          2 s
    ✓ DicesToString1        1 s
    ✓ DicesToString2        1 s
```

To test the function *AreEquals()* of this same class, we will create two dices with fixed value. The same value during the first test, and a different one during the second one. Then we will see if we have the results we expected.



```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestDicesAreEquals1()
    {
        Monopoly.Dices d = new Monopoly.Dices();
        d.Rolls[0] = 1;
        d.Rolls[1] = 1;
        bool res = d.AreEquals();

        Assert.AreEqual(res, true);
    }

    [TestMethod]
    public void TestDicesAreEquals2()
    {
        Monopoly.Dices d = new Monopoly.Dices();
        d.Rolls[0] = 1;
        d.Rolls[1] = 4;
        bool res = d.AreEquals();

        Assert.AreEqual(res, false);
    }
}
```



```
◢ ✓ UnitTest1 (2)              7 ms
    ✓ TestDicesAreEquals1      7 ms
    ✓ TestDicesAreEquals2    < 1 ms
```

Extra Features & Final Remarks

Extra Features

Luck & Community Squares - We implemented **2** additional squares to the more-classic set of squares. A **LuckSquare** that makes you go *5 squares ahead* and a **CommunitySquare** that takes you *3 squares backwards*. These new features make the game more dynamic and fun to play !

Victory Condition – To set a more interesting end-game and victory condition, we implemented a *distance* property for each **Player** and ask at the start of the game how many rounds does the user want to play. Doing so, we can display the winner at the end based on the distance each player covered.

Final Remarks

For our final remarks, we would like to emphasize on **2 essential points** for us and **1 lesson** we learnt while realizing this project:

- First, one of the main point we focused on during the implementation of our solution is the **user-friendly interface** that we are proud of. This goes through using colors, *System.Threading.Thread.Sleep()* to mimic the real playing game, pseudo for the players and of course clear display of the different information to follow the game easily.
- Then, in our opinion, the challenge of this project was to **deal with the extreme cases** and think about all the possibilities such as triple double and landing on *GoToJail* **Square**. However, we manage to deal with it appropriately.
- Finally, **working together** on the same code was complicated but thanks to the solution *Visual Studio* offers and good commenting habits we did great !

Feel free to check our GitHub: *https://github.com/Nibleash/Monopoly*