ESILV

Report
Final project

Design Pattern & Soft dev

GOUBET Victor - Daems Chloé – DIA2
11/01/2021

Exercice 1 – CustomQueue – Generics

1. Introduction

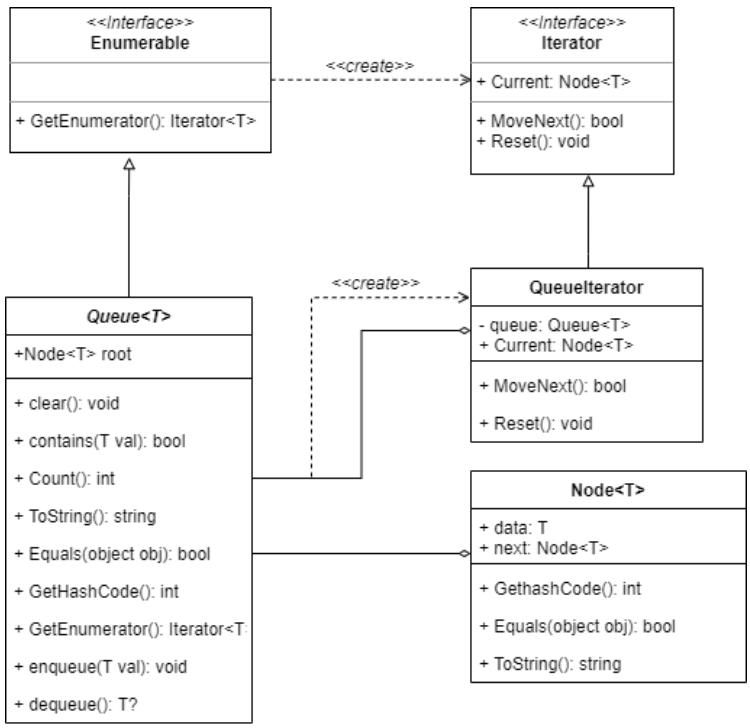
The goal here is to create from scratch an entire generic queue class. It must have all the major functions and characteristics of the provided queue as the capacity to be used in a foreach loops.

2. Design Hypotheses

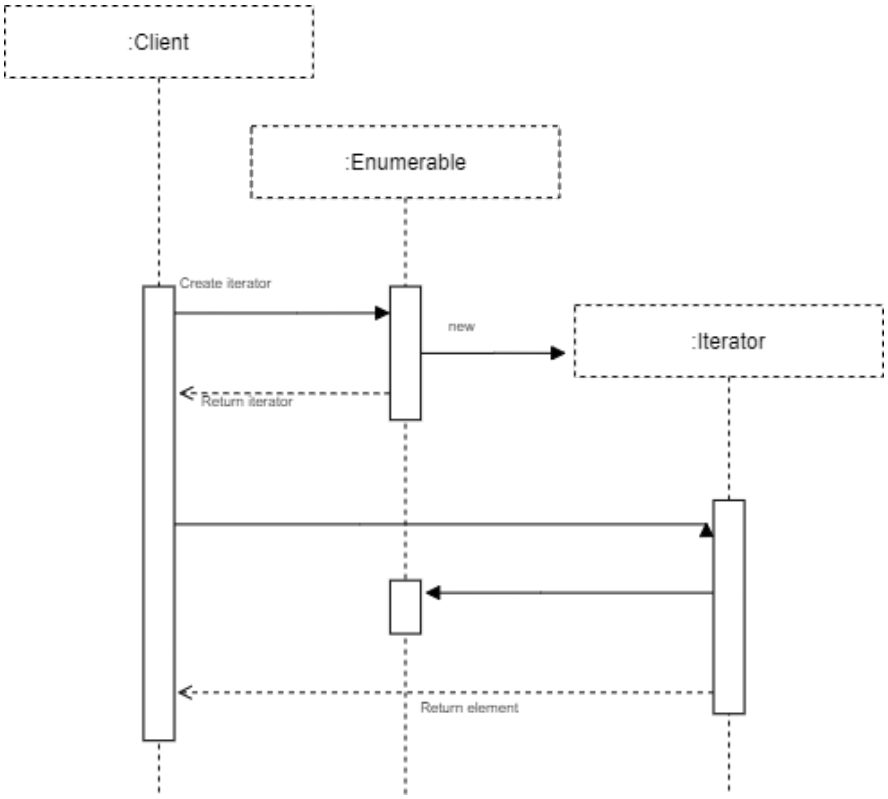
For this problem, we use the model of a queue composed of nodes. Each Nodes have a generic value and a next node. A queue has a root which is the foot of the queue. In order to make the custom queue iterable, we have implemented the iterator model. Thus, we can iterate throw the nodes of a queue.

3. UML diagrams

Class diagramm :



Sequence diagramm :



4. Test cases

We made a total of 10 tests in order to assert that our Custom queue and its methods works properly. We checked that dequeue(), clear(), enqueue(), count(),contains() works well even with empty queues. We're happy to say that all our tests appeared to be correct, which lets us think that our code works properly.

5. Additional / Final remarks

We add the constraint where T: struct to be able to return a nullable type when we dequeue an empty queue. Thus, the dequeue method return a T? type. Moreover, an improvement would be to add the possibility to iterate throw the T data instead of Node<T>

Exercice 2 – MapReduce

1. Introduction

We want here to create a basic implementation of the MapReduce function. The goal is to give an input to the algorithm, it divides it in subset and apply a map function on each subset. Then it reduce / resume all results in one. It allows to process huge number of data quickly.

2. Design Hypotheses

For this exercise we will make some hypotheses. First, we simulate independent machine thanks to a class called $Worker$. This class take an input and a really generic function as arguments. Thus, we have a first step where we apply all map function on each subset throw different workers (each worker executes a map function with a thread), then we shuffle the result and finally we apply all the reduce function in parallel ways.

Moreover, we must define a data exchange protocol so that the data and map/reduce function be understandable by our MapReduce class. To do that we use three types: $< K1, V1 >$ the type of the key and value of the input, $< K2, V2 >$ the type of the intermediate values after the map step and finally $< K3, V3 >$ the type of the result. The input must be an iterable collection of Key/Value pair $< K1, V1 >$, for example it can be a $dictionnary < K1, V1 >$. This choice seems to be relevant because all type of input can be formatted under this shape. In all our class we use a lot this type: $\text{IEnumerable} < \text{KeyValuePair} < K_k, V_k >>$ because it is really generic and that why we want!

To force users to define well their map/reduce function we define delegates with these signatures:

- $\text{IEnumerable} < \text{KeyValuePair} < \text{K2}, \text{V2} > \text{map\_function}(\text{K1 key}, \text{V1 val})$
- $\text{KeyValuePair} < \text{K3}, \text{V3} > \text{ reduce\_function}(\text{K2 key}, \text{IEnumerable} < \text{V2} > \text{ val})$

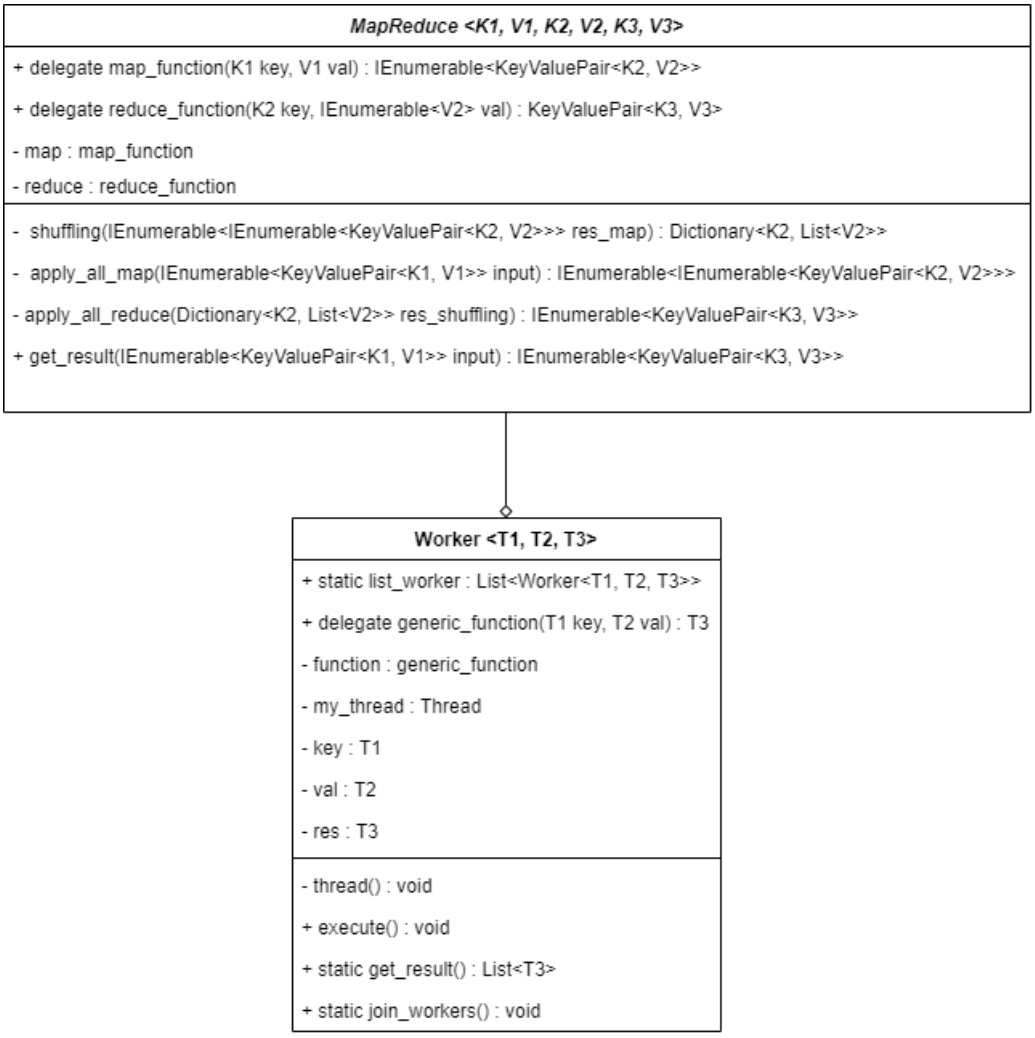
And the signature of the function of a worker have this signature (more generic):

- $T3\ generic\_function(T1\ key, T2\ val)$

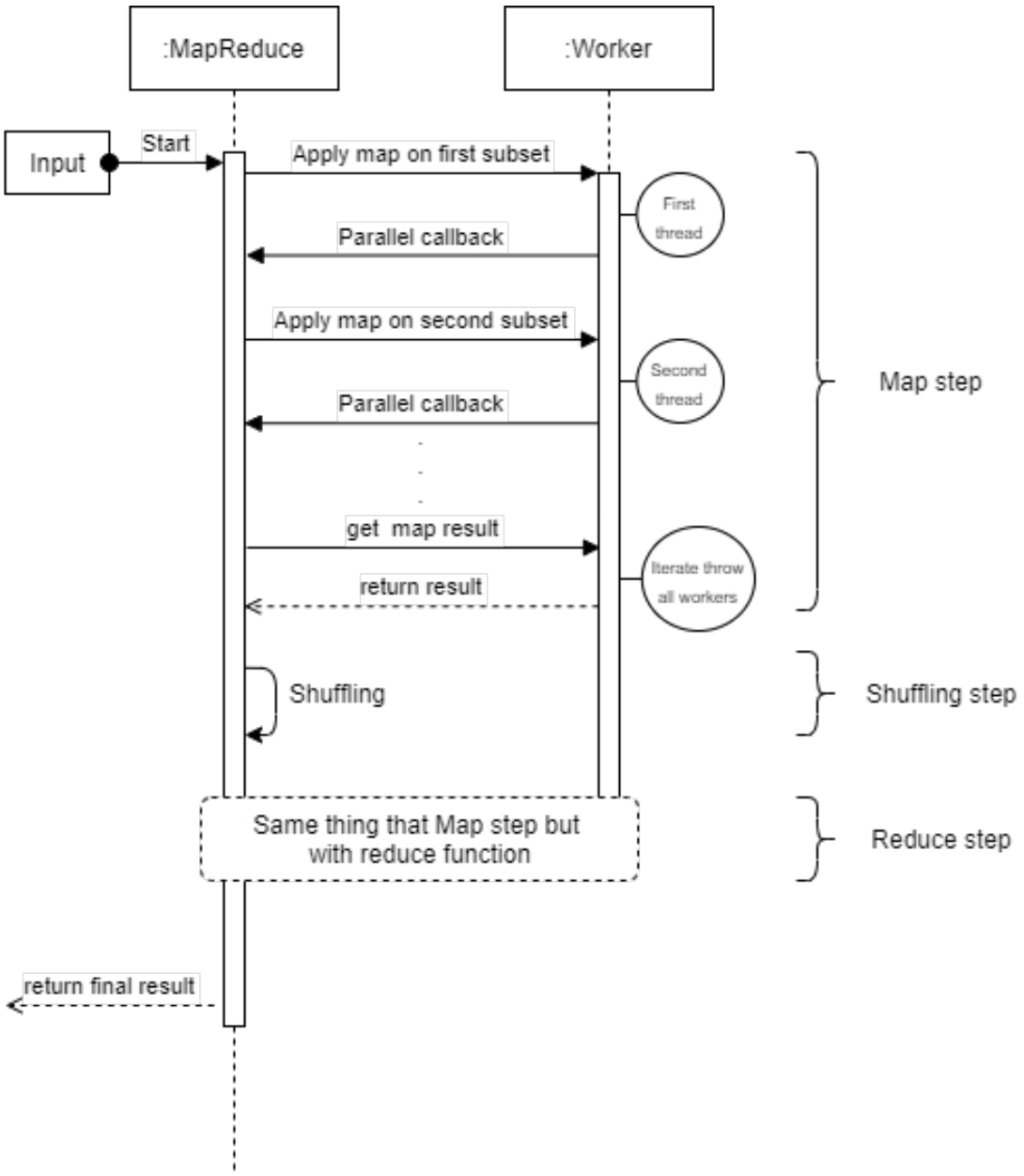
Finally, to get the result of workers, each worker keeps his result in his property “res” and at the end of a step we iterate throw all workers and get all results.

3. UML diagrams

Class diagramm :



Sequence diagramm :



4. Test cases

For the tests, we have defined two-unit test:

- The first check if our MapReduce works on the word counting example
- The second check if our MapReduce works on the matrix columns sum. By the way, for the example we have define a dictionary of matrix. We can imagine that the original matrix, before being split, was all these matrixes concatenate by rows.

Exercice 3 – Monopoly – Design Patterns

1. Introduction

This exercise asks us to code a simplified monopoly game with only the case "Go to jail" and "jail". The players have to roll the dices and move of as many boxes as the dices indicate. As in a normal monopoly game, if a player makes a double, he has to play again, if 3 doubles are made in a row then the player goes to jail. To get out of jail the player has to either make a double or pass 3 times his turn.

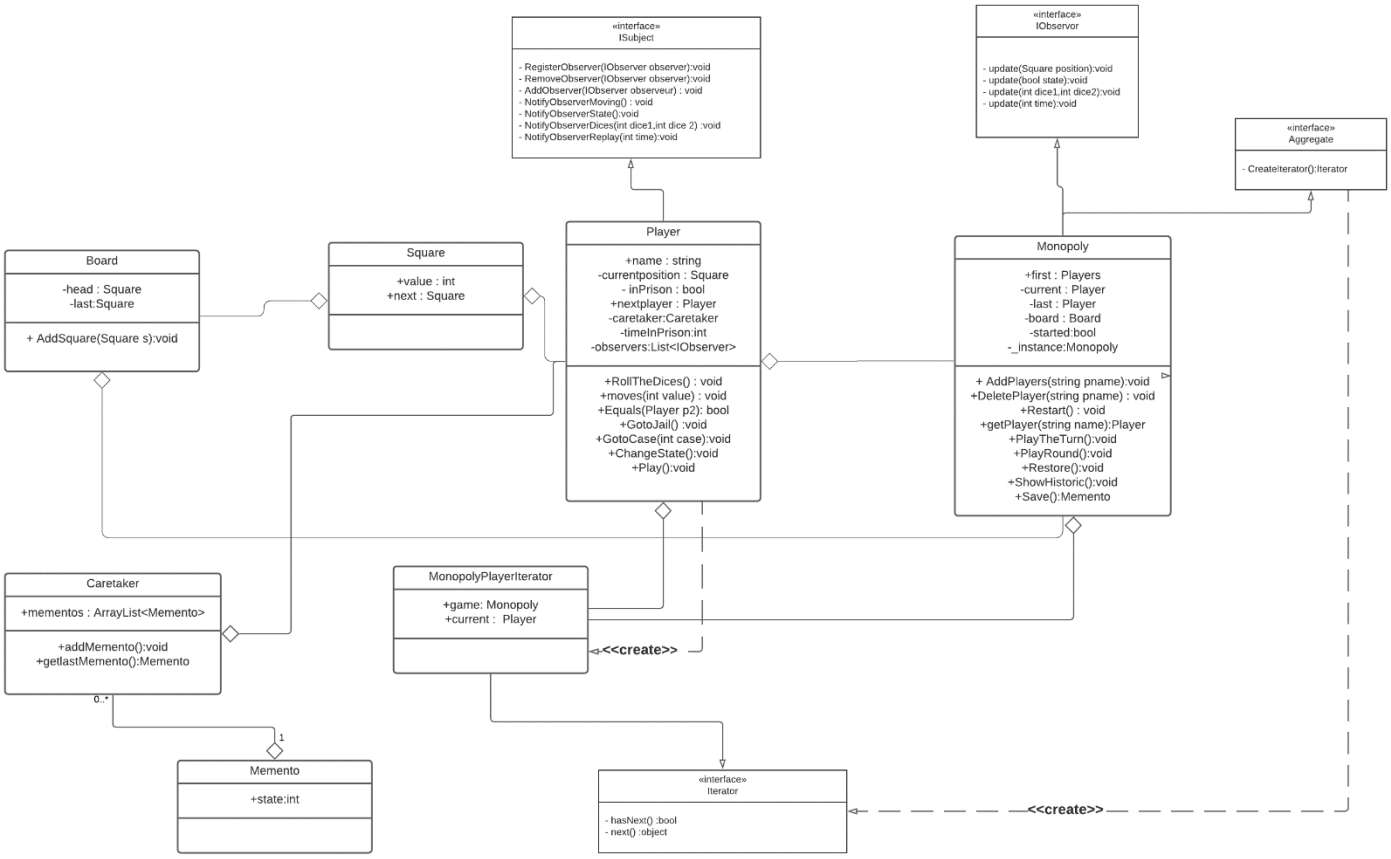
2. Design Hypotheses

To code this monopoly, we chose to represent the board as a circular linked list with all the 40 boxes, in a way that when being on the 39th box it will directly move to box 0. We also chose to represent the game as a circular linked list of players so that, in the same way, after the last player played it will be back to the first player to roll the dices. Therefore, we created 4 main classes: Player, Board, Box and Monopoly.
Now, we wanted to integrate usual design patterns in order to simplify our code. We added:

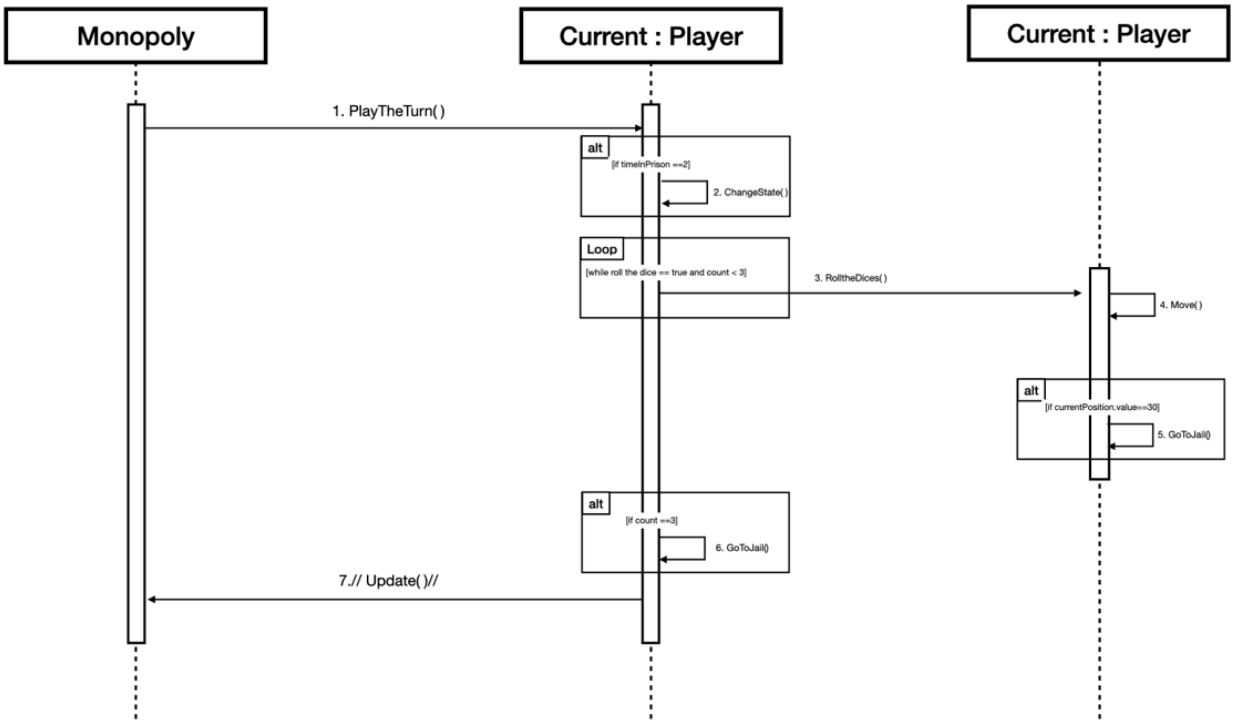
- Iterator Pattern: Just on the players so that we can go through all the players and know the position of each player.
- Observer Pattern: On the players too, to be notified when the player goes in/out of prison, on what dice values did he get and to know the new position of the player when he moved.
- Singleton Pattern: Because we want to create only one game, you cannot play two games on the same program.
- Memento Pattern: To know the historic of moves of all the players.

3. UML diagrams

Class diagram



Sequence Diagram of the PlayTheTurn() method :



4. Test cases

We made several tests on the things that were asked in the subject:

- When a player does 3 times in a row a double dice he goes to jail:

```
[TestMethod]
public void TestGoToJailwDouble()
{
    Monopoly.Monopoly game = Monopoly.Monopoly.GetInstance();
    game[0].Play(4, 4);
    Assert.AreEqual(game[0].InPrison, true);

}
```

- When a player lends on case 30 (Go to prison) he goes to jail:

```
[TestMethod]
public void TestGoToJailOn30()
{
    Monopoly.Monopoly game = Monopoly.Monopoly.GetInstance();
    game.Restart(); //Because with the sigleton pattern it keeps the ulterior state
    //Adding players
    game.AddPlayer("Jean");
    game.AddPlayer("Paul");
    game.AddPlayer("Jacques");

    game[0].Play(4, 6);
    game[0].Play(4, 6);
    game[0].Play(4, 6);
    Assert.AreEqual(game[0].InPrison, true);

}
```

- When a player is in jail and does a double dice he leaves jail and moves of the sum vale of the dices:

```
[TestMethod]
public void OutOfJail1wDouble()
{
    Monopoly.Monopoly game = Monopoly.Monopoly.GetInstance();

    game[0].Play(4, 4);

    Assert.AreEqual(game[0].InPrison, false);
    Assert.AreEqual(game[0].CurrentPosition.value, 18);

}
```

- When a player was in jail for 3 rounds he leaves jail and move of the sum value of the dices:

```
[TestMethod]
public void OutOfJail1w3Rounds()
{
    Monopoly.Monopoly game = Monopoly.Monopoly.GetInstance();

    game[0].Play(4, 4); //Go to Jail (it simulate 3 doubles in a row)

    game[0].Play(2, 6);
    game[0].Play(3, 6);
    game[0].Play(4, 6);


    Assert.AreEqual(game[0].InPrison, false);
    Assert.AreEqual(game[0].CurrentPosition.value, 20);

}
```

- The board is circular meaning that after case 39 there is case 0:

```
[TestMethod]
public void TestSetPosition()
{
    Monopoly.Monopoly game = Monopoly.Monopoly.GetInstance();
    game[0].GoToCase(39);
    Assert.AreEqual(game[0].CurrentPosition.value, 39);

}

[TestMethod]
public void TestCircularity()
{
    Monopoly.Monopoly game = Monopoly.Monopoly.GetInstance();
    Assert.AreEqual(game[0].CurrentPosition.next.value, 0);

}
```

5. Additional / Final remarks

Thanks to design patterns, a lot of things are simpler to be done especially the Observer which allow to display all the changes in the game that occurs, making the course of the game more understandable. A remark would be that this monopoly was really simplified, but if we had to make a real one, with all the different type of case, we would have implemented the factory pattern that would have been really helpful to organize the game. We would also have created subclass of Square to be more organized on the different possible actions. Here we could have done it for the "GoToJail" case and the "Jail" case but as there would be only one of each we preferred doing it all in the Player class.

Note on the design of the monopoly

We decided to do just one class named Box after different thoughts about making an abstract class Box with subclasses "NormalBox", "GoToJail" and "Jail", if we had done that we would have given the abstract function "void Action()" that would've done the actions depending on the type of box. But we saw that this Action function would've been empty for "NormalBox "and "Jail" and would've been "GoToJail()" for the "GoToJailBox", also, as there are only one "GoToJailBox" and one "JailBox" we thought it would've been useless. We would've done that if we had to do many kinds of boxes that have distinct actions on it. If we had multiple kind of boxes, we would've implemented the Factory Pattern to be more efficient. We hope you understand our process of thinking.