<u>Design Patterns and Software Development Process: FINAL PROJECT:</u>

<u>Made By:</u> Bâ Nandy & Badouri Mouna

<u>Exercise 3 – A Monopoly™ game– Design patterns :</u>

<u>Introduction :</u>

In this exercise, we were asked to create and simulate a simplified version of the Monopoly game.

It was a rather exciting challenge since the game was fairly popular and well-known.

So we tried our best to respect the rules of the exercise but also try to reason and complete what wasn't mentioned, by our humbled knowledge of the assigned game.

<u>Design Patterns:</u>

Since this was a project in the subject of Design Patterns and Software Development Process, we felt it was rather necessary to implement at least one or two design patterns to be faithful to the whole process.

And we accomplished that, since as we were reading the rules and the instructions of the exercise, the need to implement these two design patterns was rather blatant and explicit.

We have chosen: **Observer design pattern** and **the Singleton design pattern**.

<u>Explaining the choice:</u>

<u>The Observer design pattern:</u>

Concerning the choice of this design pattern, it was a rather obvious choice, since we were handling a game, and there was a constant change of the players' positions (**0-39**) and state **(In Jail Or Not)** and as we have seen in the class, a fairly effective way to keep being notified is by implementing the **Observer Design Pattern**.

As you may know the **Observer Design Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The object which is being watched is called the *subject*. The objects which are watching the state changes are called *observers* or *listeners*.

We have two observers that we called:

```
Observer_Dices()
```

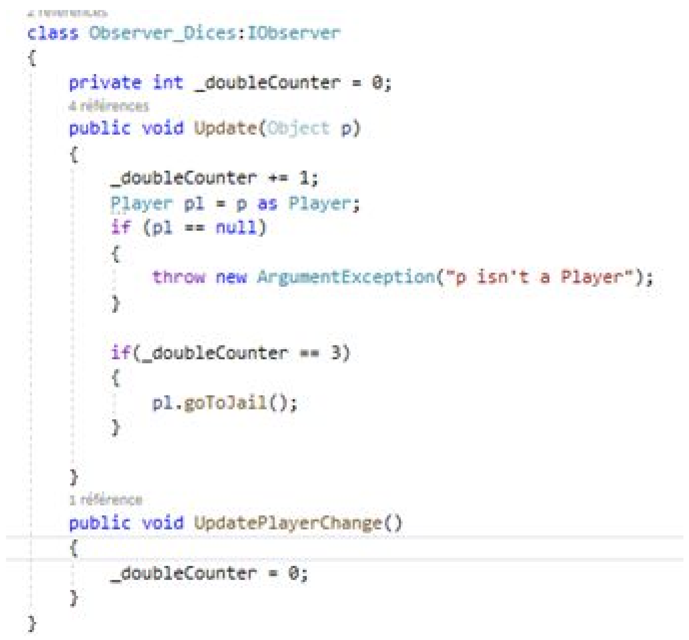
and

```
Observer_JailOrNot()
```

For the **Observer_Dices**: it's an observer that has a goal of notifying and updating when there's a player change by using the method: **UpdatePlayerChange() .**

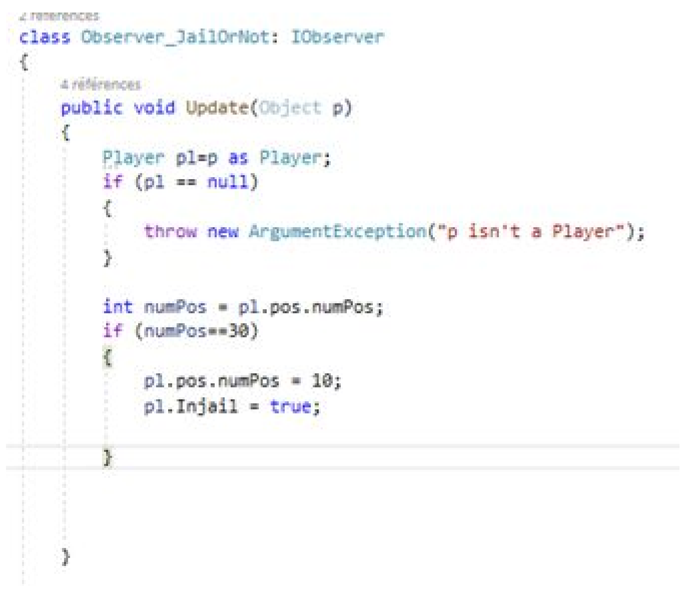
It also has a goal of Updating and keeping count of the results of the dices, especially when it comes to the doubles i.a playing and rolling the dices and incrementing the counter when we both the dices give the same result.

But as mentioned in the instructions: a player goes straight to jail if the counter (or number of doubles is superior than 3), by using the method **Update().**



```
class Observer_Dices:IObserver
{
    private int _doubleCounter = 0;
    4 références
    public void Update(Object p)
    {
        _doubleCounter += 1;
        Player pl = p as Player;
        if (pl == null)
        {
            throw new ArgumentException("p isn't a Player");
        }

        if(_doubleCounter == 3)
        {
            pl.goToJail();
        }

    }
    1 référence
    public void UpdatePlayerChange()
    {
        _doubleCounter = 0;
    }
}
```

As for the Observer_JailOrNot: it notifies when the player's position is in jail or not.



```
2 references
class Observer_JailOrNot: IObserver
{
    4 références
    public void Update(Object p)
    {
        Player pl=p as Player;
        if (pl == null)
        {
            throw new ArgumentException("p isn't a Player");
        }

        int numPos = pl.pos.numPos;
        if (numPos==30)
        {
            pl.pos.numPos = 10;
            pl.Injail = true;

        }



    }
```

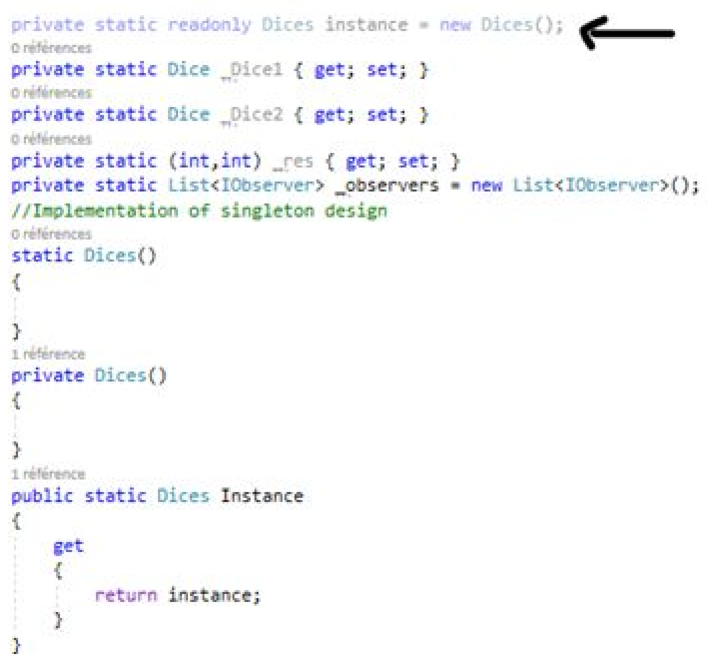
The singleton design pattern:

When it comes to this pattern, we thought that it was really important to implement, since if we see the dices class we need an instance of dices that will be protected and not modified.

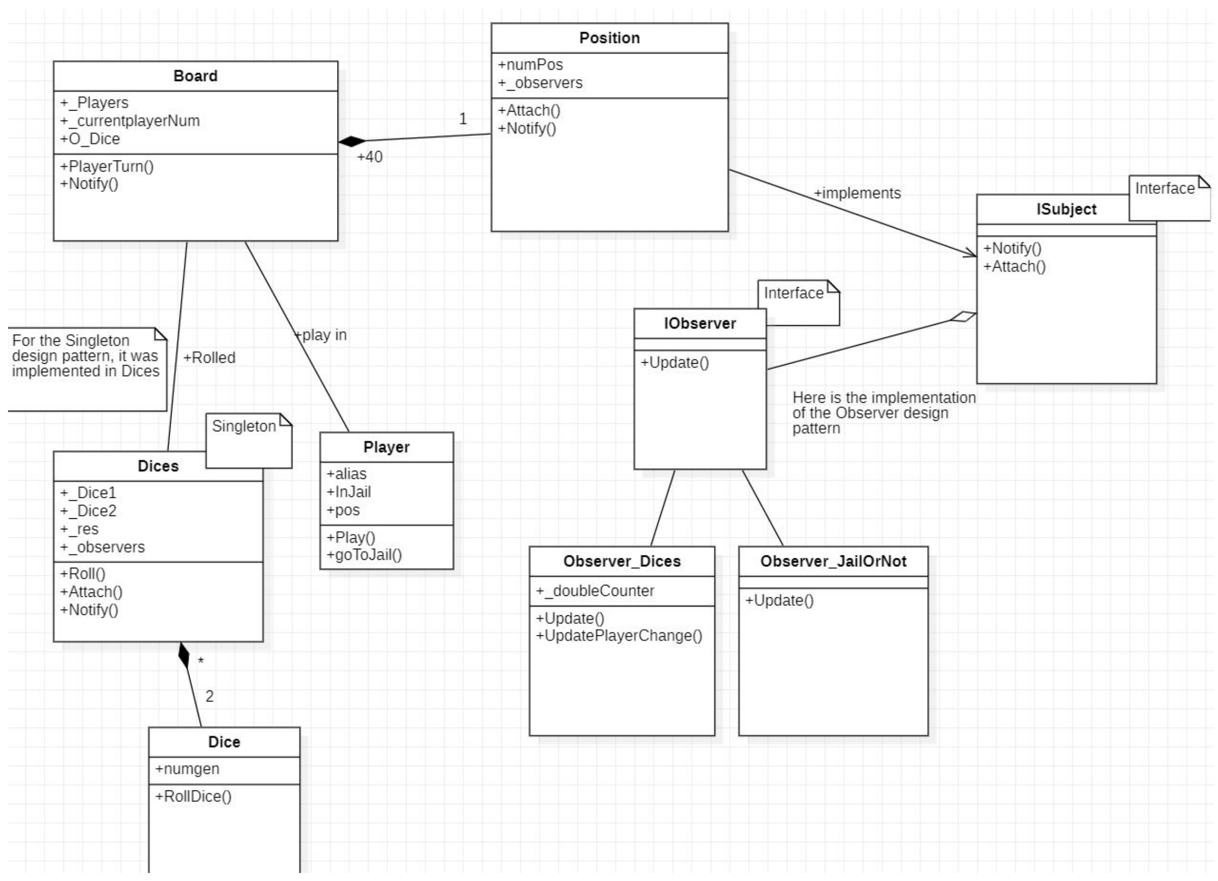
That's why we chose it to have the same observer, we had another observer and we didn't want to lose the aforementioned observer.

As we all know Singleton design pattern in C# is one of the most common design patterns is software design. The singleton design pattern ensures a class has only one instance in the program and provides a global point of access to it. A singleton is a class that only allows a single instance of itself to be created and usually gives simple access to that instance.

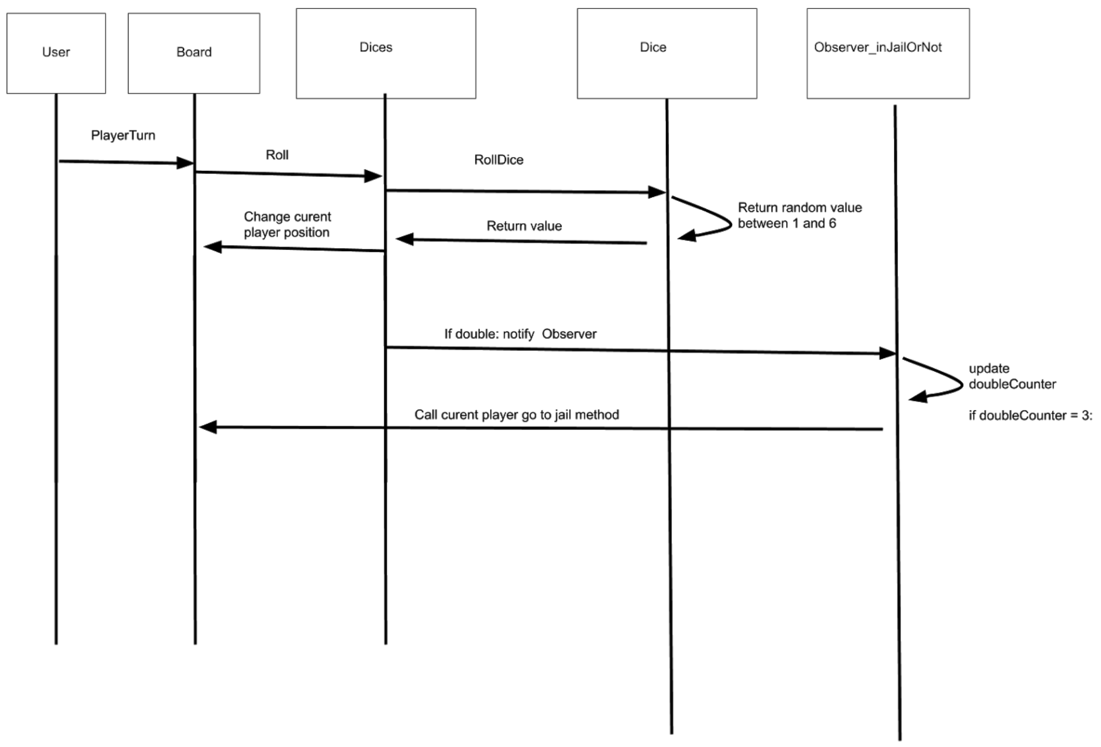
This is how we implemented it in our case .


```
private static readonly Dices instance = new Dices();
0 références
private static Dice _Dice1 { get; set; }
0 références
private static Dice _Dice2 { get; set; }
0 références
private static (int,int) _res { get; set; }
private static List<IObserver> _observers = new List<IObserver>();
//Implementation of singleton design
0 références
static Dices()
{

}
1 référence
private Dices()
{

}
1 référence
public static Dices Instance
{
    get
    {
        return instance;
    }
}
```

Class Diagram:



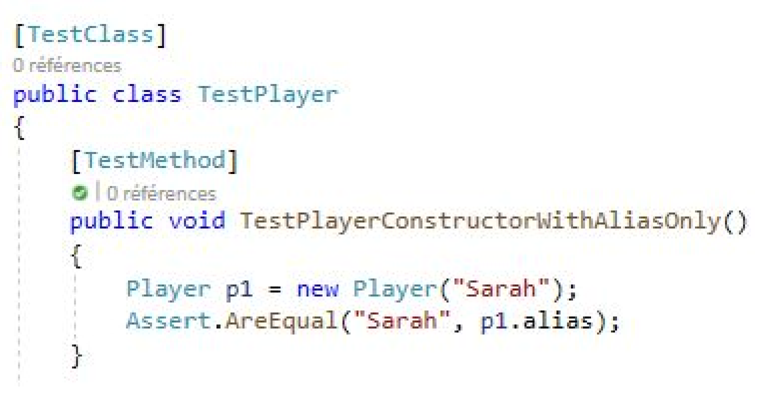
<u>Sequence Diagram:</u>


User
Board
Dices
Dice
Observer_inJailOrNot
PlayerTurn
Roll
RollDice
Return random value between 1 and 6
Change curent player position
Return value
If double: notify Observer
update doubleCounter
if doubleCounter = 3:
Call curent player go to jail method

<u>Test cases:</u>

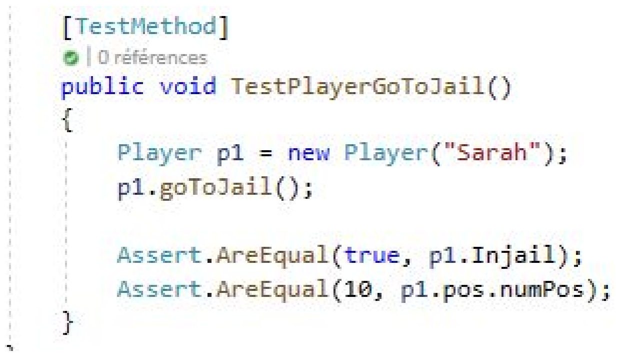
Player:

The first test enables us to verify If the Player Class implements in fact a "constructor" that takes in parameters the player's alias. It makes sure that the alias is conserved as an attribute of instance.



```
[TestClass]
0 références
public class TestPlayer
{
    [TestMethod]
    | 0 références
    public void TestPlayerConstructorWithAliasOnly()
    {
        Player p1 = new Player("Sarah");
        Assert.AreEqual("Sarah", p1.alias);
    }
```

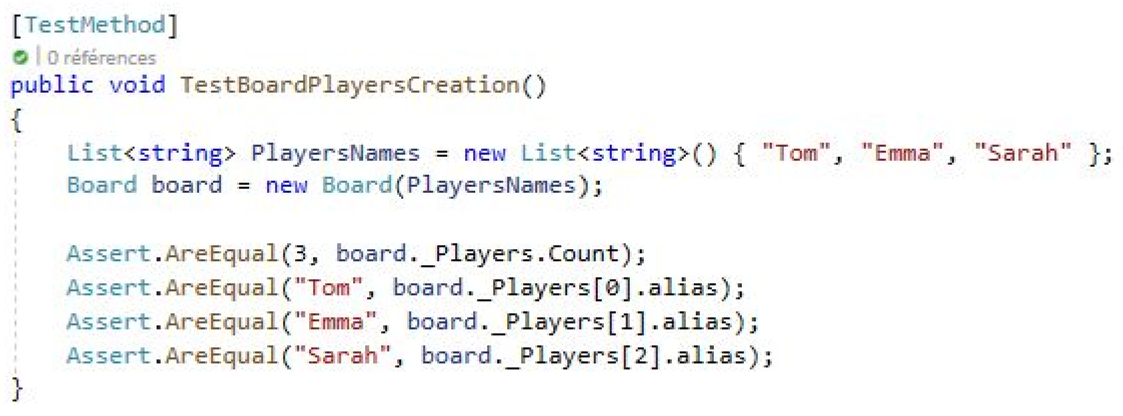
The second unit test verifies whether the goToJail method functions accurately (by verifying whether the Player’s InJail property is changed to True and that the Player’s has been placed in position 10) .



```
    [TestMethod]
    | 0 références
    public void TestPlayerGoToJail()
    {
        Player p1 = new Player("Sarah");
        p1.goToJail();

        Assert.AreEqual(true, p1.Injail);
        Assert.AreEqual(10, p1.pos.numPos);
    }
```

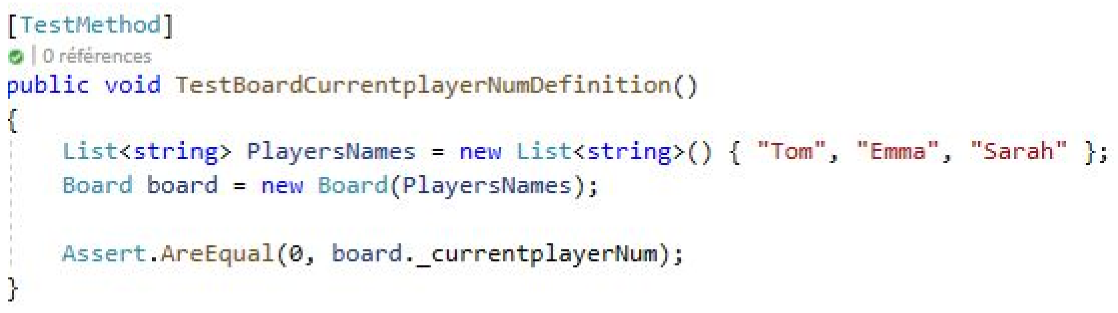
Board:

The first unit test makes sure that the board is initialized, especially focusing whether the Players’ list has been created.



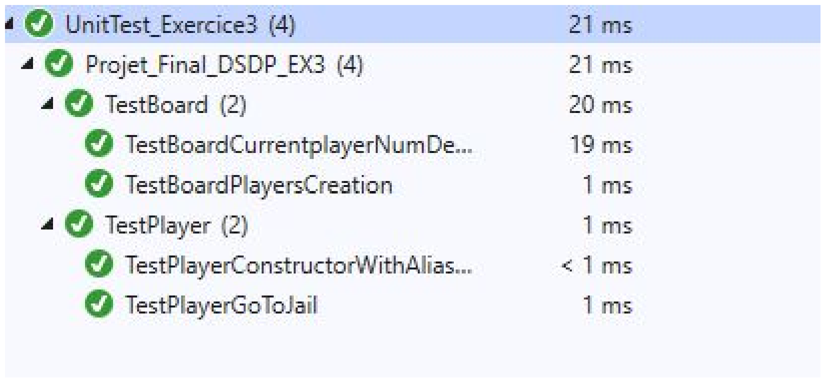
```
[TestMethod]
| 0 références
public void TestBoardPlayersCreation()
{
    List<string> PlayersNames = new List<string>() { "Tom", "Emma", "Sarah" };
    Board board = new Board(PlayersNames);

    Assert.AreEqual(3, board._Players.Count);
    Assert.AreEqual("Tom", board._Players[0].alias);
    Assert.AreEqual("Emma", board._Players[1].alias);
    Assert.AreEqual("Sarah", board._Players[2].alias);
}
```

The second unit test makes sure that the property currentPlayerNum is initialized.



```
[TestMethod]
0 références
public void TestBoardCurrentplayerNumDefinition()
{
    List<string> PlayersNames = new List<string>() { "Tom", "Emma", "Sarah" };
    Board board = new Board(PlayersNames);

    Assert.AreEqual(0, board._currentplayerNum);
}
```

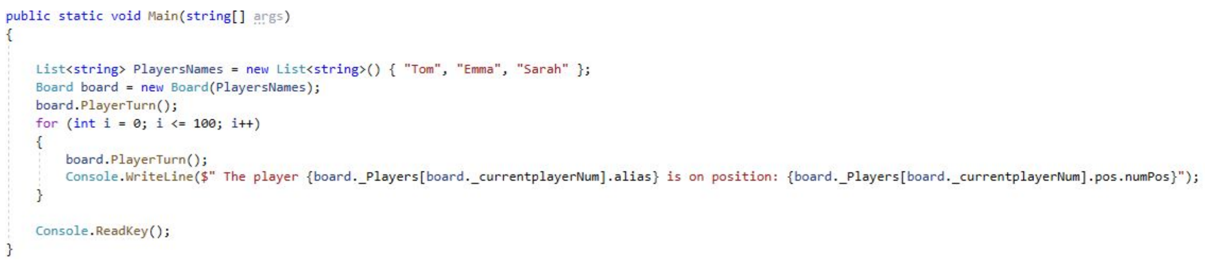
As we can see clearly in the screen below, the unit tests run perfectly, and there's no error.



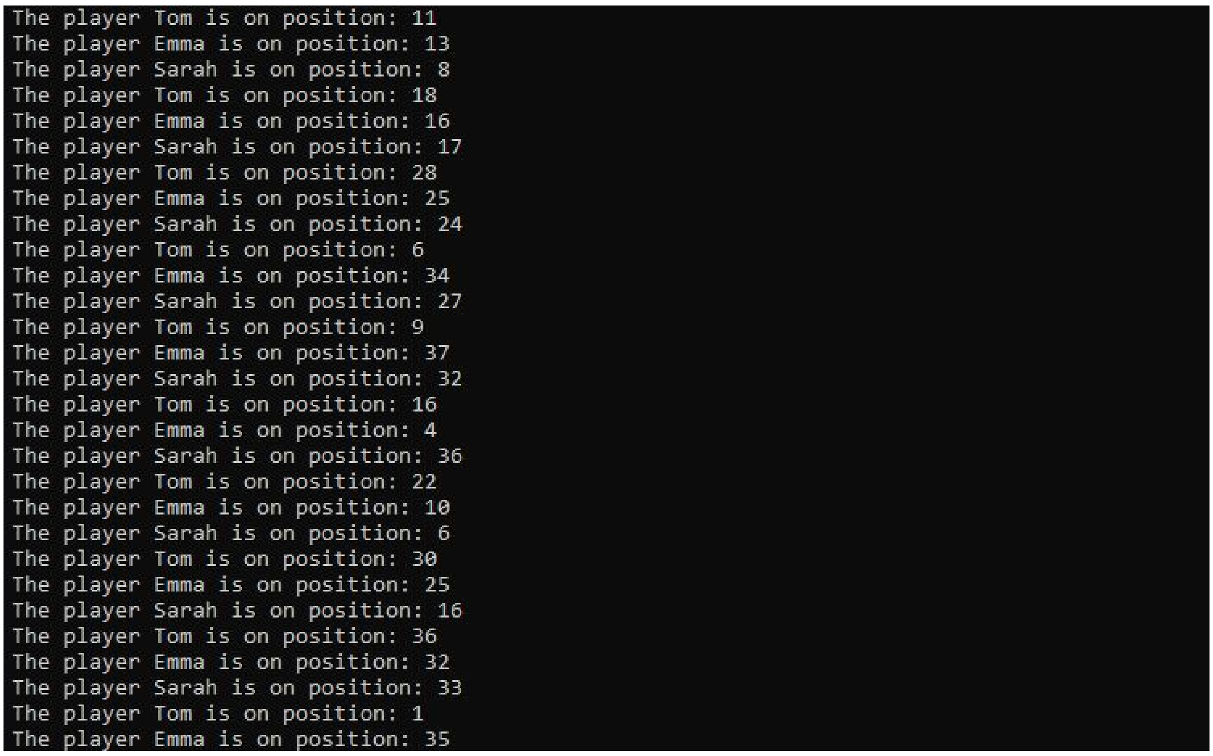
```
◢ ✓ UnitTest_Exercice3 (4)                    21 ms
  ◢ ✓ Projet_Final_DSDP_EX3 (4)                21 ms
    ◢ ✓ TestBoard (2)                          20 ms
        ✓ TestBoardCurrentplayerNumDe...       19 ms
        ✓ TestBoardPlayersCreation              1 ms
    ◢ ✓ TestPlayer (2)                          1 ms
        ✓ TestPlayerConstructorWithAlias...   < 1 ms
        ✓ TestPlayerGoToJail                    1 ms
```

Additional Remarks:

In the Player Class we decided that, when initializing the Player instance for the first time all players have an initial position of 0.
In the main we have a couple of tests, but in fact we have done a few other ones that we have mentioned in the unit test section.



```csharp
public static void Main(string[] args)
{

    List<string> PlayersNames = new List<string>() { "Tom", "Emma", "Sarah" };
    Board board = new Board(PlayersNames);
    board.PlayerTurn();
    for (int i = 0; i <= 100; i++)
    {
        board.PlayerTurn();
        Console.WriteLine($" The player {board._Players[board._currentplayerNum].alias} is on position: {board._Players[board._currentplayerNum].pos.numPos}");
    }

    Console.ReadKey();
}
```



```
The player Tom is on position: 11
The player Emma is on position: 13
The player Sarah is on position: 8
The player Tom is on position: 18
The player Emma is on position: 16
The player Sarah is on position: 17
The player Tom is on position: 28
The player Emma is on position: 25
The player Sarah is on position: 24
The player Tom is on position: 6
The player Emma is on position: 34
The player Sarah is on position: 27
The player Tom is on position: 9
The player Emma is on position: 37
The player Sarah is on position: 32
The player Tom is on position: 16
The player Emma is on position: 4
The player Sarah is on position: 36
The player Tom is on position: 22
The player Emma is on position: 10
The player Sarah is on position: 6
The player Tom is on position: 30
The player Emma is on position: 25
The player Sarah is on position: 16
The player Tom is on position: 36
The player Emma is on position: 32
The player Sarah is on position: 33
The player Tom is on position: 1
The player Emma is on position: 35
```

Exercise 1 – CustomQueue – Generics :

Introduction :

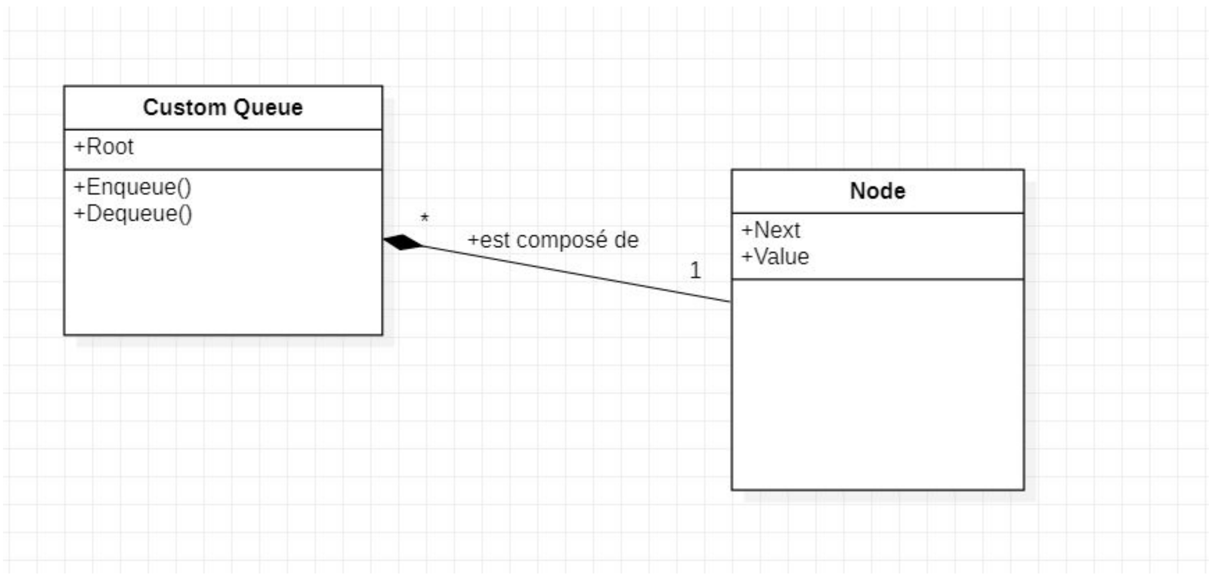
In this exercise we were asked to design and implement a Custom Queue which is a data structure that can contain all sorts of data hence why we're dealing with generics. As we all know The use of generics is an actual generalization of any type, generic means the general form, not specific.

In C#, generic means not specific to a particular data type.since we can have a list of not just strings, booleans or int, but we can also have a list with objects that we have defined in a certain class, thus it's name being a custom queue since it's personalized in a way, by only indicating the type when we want to use the actual list or structure, (for example: List<T> in which T could be a type that we can define later on)

Design Patterns:

The goal of this exercise was not to implement design patterns, but it was rather a matter of knowing how to apply generics and go around them in an attempt to create a custom data structure that will contain a certain undisclosed type T. Hence why we didn't use any design patterns.

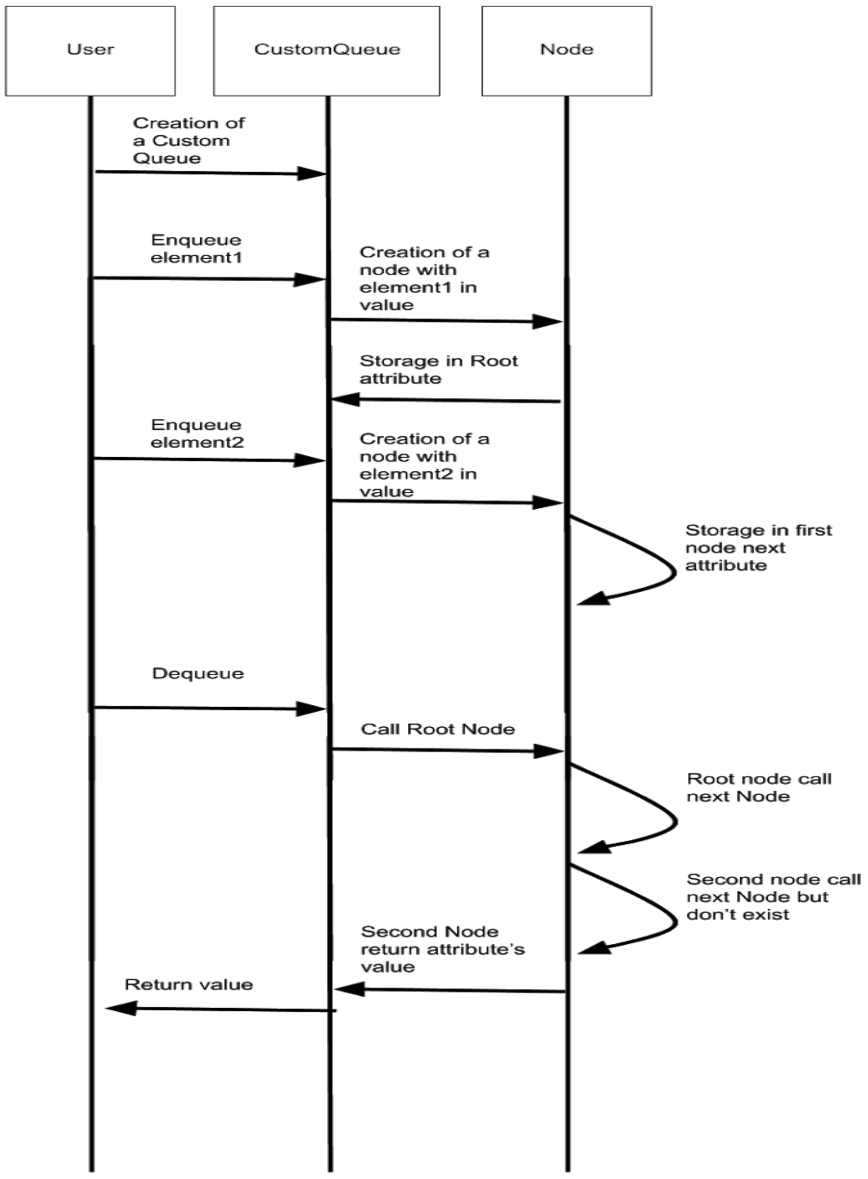
Class Diagram:



In here we have opted for two classes named : CustomQueue and Node. The first one being the actual data structure and the second one being the nodes that will be contained inside the aforementioned structure.
We have chosen to implement a composition relationship between the two since the custom queue is in fact composed of nodes, we also took notice of the cardinality as you can clearly see in the diagram.

Sequence Diagram:

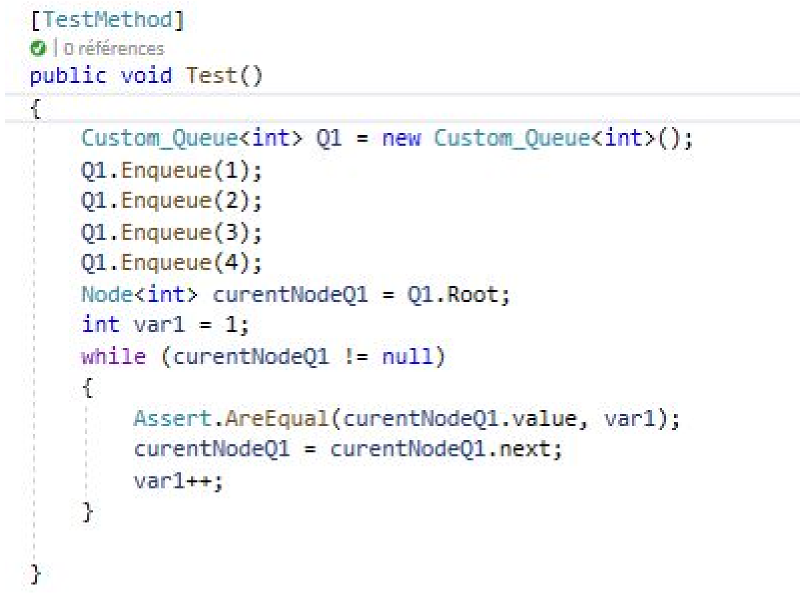
In this sequence diagram we tried to simulate what our code would do in a user case.



User
CustomQueue
Node
Creation of a Custom Queue
Enqueue element1
Creation of a node with element1 in value
Storage in Root attribute
Enqueue element2
Creation of a node with element2 in value
Storage in first node next attribute
Dequeue
Call Root Node
Root node call next Node
Second node call next Node but don't exist
Second Node return attribute's value
Return value

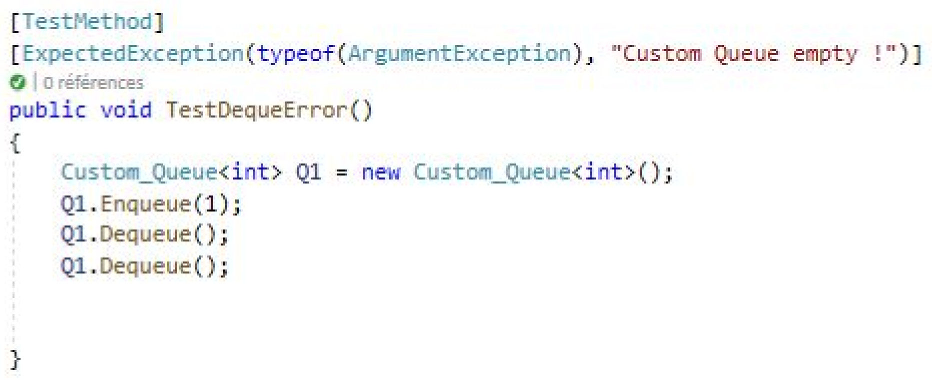
Test cases:

For the first unit test we tried to see whether the method Enqueue worked or not and it did; as you can see in the screen we have started off with a blank queue with a type T = int, and we filled it with integers. and we have checked whether as the customqueue moved from one value to another by using the Next attribute that refers to the next value after the current one; if it really did change as we go, and it worked.



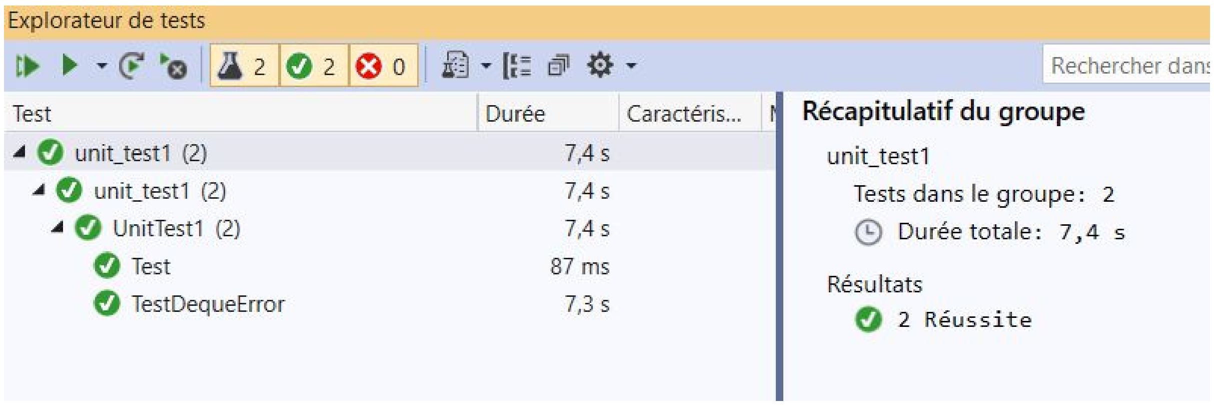
```
[TestMethod]
0 références
public void Test()
{
    Custom_Queue<int> Q1 = new Custom_Queue<int>();
    Q1.Enqueue(1);
    Q1.Enqueue(2);
    Q1.Enqueue(3);
    Q1.Enqueue(4);
    Node<int> curentNodeQ1 = Q1.Root;
    int var1 = 1;
    while (curentNodeQ1 != null)
    {
        Assert.AreEqual(curentNodeQ1.value, var1);
        curentNodeQ1 = curentNodeQ1.next;
        var1++;
    }

}
```

As for the second unit test, it was actually a unique one since we tried to throw an exception when the customqueue is empty, as we could foresee its utility in future runs of the code especially when the data structure is emptied by the method Dequeue; so in fact, to be more accurate it’s both a test unit to make sure to notify when the custom queue is empty, but it also serves as a way to notice if the dequeue method really worked, if we have taken in fact a full queue.



```
[TestMethod]
[ExpectedException(typeof(ArgumentException), "Custom Queue empty !")]
0 références
public void TestDequeError()
{
    Custom_Queue<int> Q1 = new Custom_Queue<int>();
    Q1.Enqueue(1);
    Q1.Dequeue();
    Q1.Dequeue();


}
```

As we can see clearly in the screen below, the unit tests run perfectly, and there's no error.


Explorateur de tests

2 ✓ 2 0

Rechercher dans

Test	Durée	Caractéris...
unit_test1 (2)	7,4 s	
unit_test1 (2)	7,4 s	
UnitTest1 (2)	7,4 s	
Test	87 ms	
TestDequeError	7,3 s	

Récapitulatif du groupe

unit_test1

Tests dans le groupe: 2

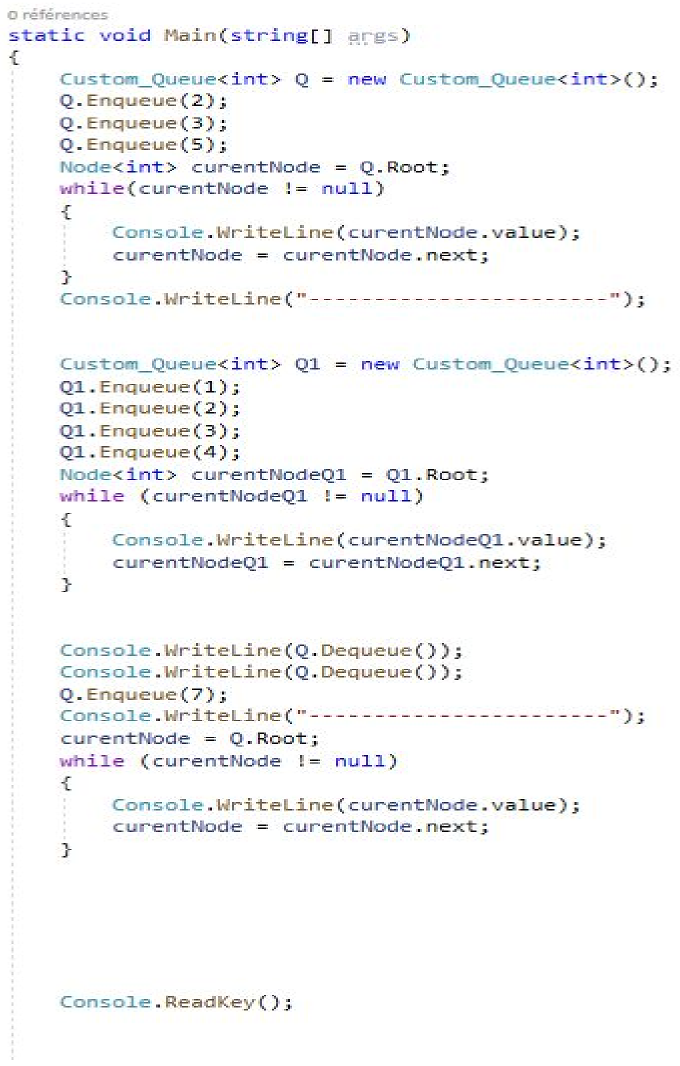
Durée totale: 7,4 s

Résultats

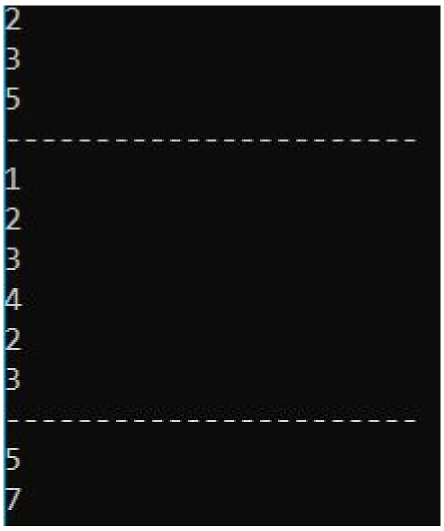
✓ 2 Réussite

Additional Remarks:

In the main we have a couple of tests, but in fact we have done a few other ones that we have mentioned in the unit test section.



```
0 références
static void Main(string[] args)
{
    Custom_Queue<int> Q = new Custom_Queue<int>();
    Q.Enqueue(2);
    Q.Enqueue(3);
    Q.Enqueue(5);
    Node<int> curentNode = Q.Root;
    while(curentNode != null)
    {
        Console.WriteLine(curentNode.value);
        curentNode = curentNode.next;
    }
    Console.WriteLine("-----------------------");


    Custom_Queue<int> Q1 = new Custom_Queue<int>();
    Q1.Enqueue(1);
    Q1.Enqueue(2);
    Q1.Enqueue(3);
    Q1.Enqueue(4);
    Node<int> curentNodeQ1 = Q1.Root;
    while (curentNodeQ1 != null)
    {
        Console.WriteLine(curentNodeQ1.value);
        curentNodeQ1 = curentNodeQ1.next;
    }


    Console.WriteLine(Q.Dequeue());
    Console.WriteLine(Q.Dequeue());
    Q.Enqueue(7);
    Console.WriteLine("-----------------------");
    curentNode = Q.Root;
    while (curentNode != null)
    {
        Console.WriteLine(curentNode.value);
        curentNode = curentNode.next;
    }




    Console.ReadKey();
```



```
2
3
5
-----------------------
1
2
3
4
2
3
-----------------------
5
7
```

Exercise 2 – MapReduce – Design patterns, Threads & IPC:

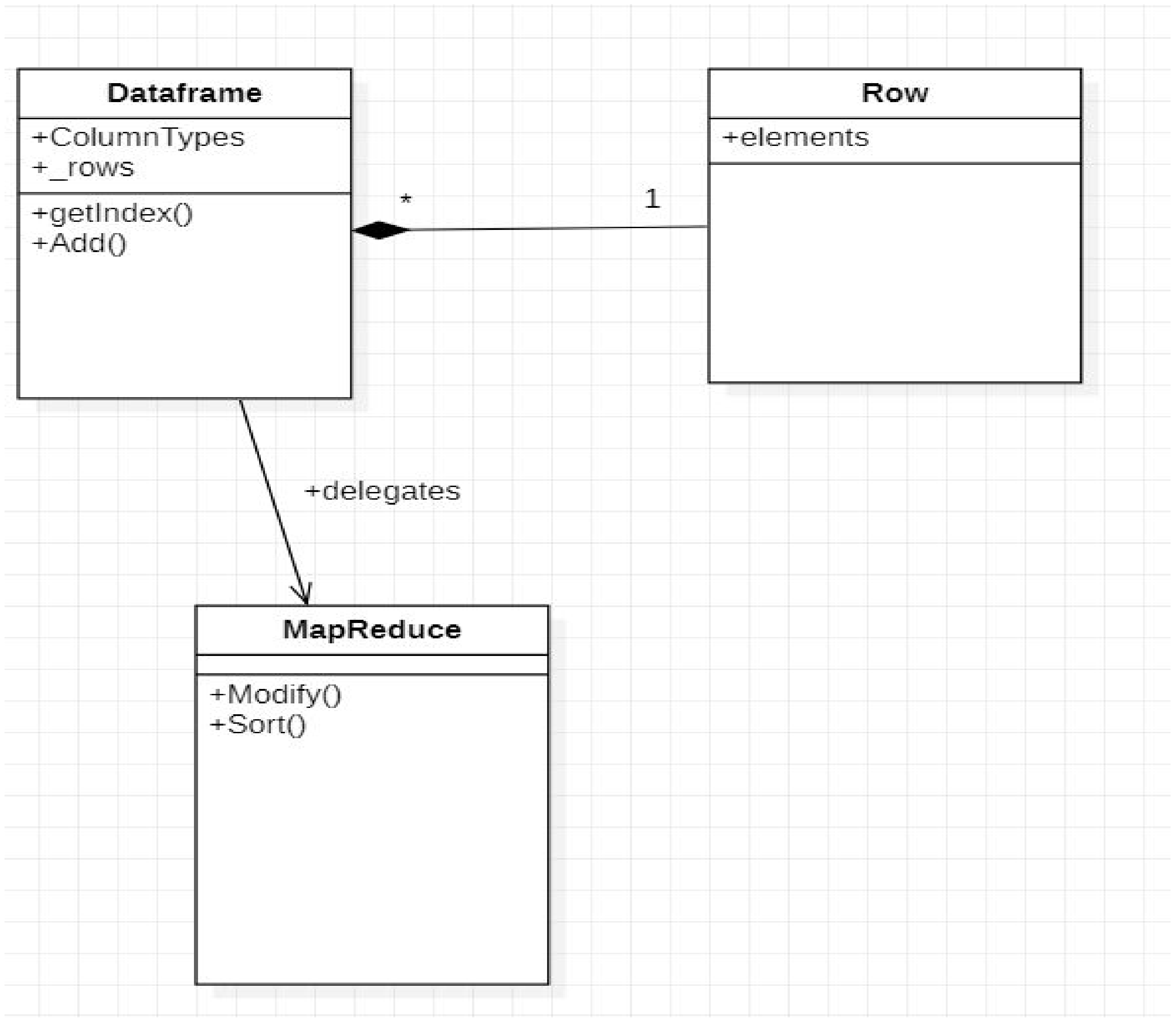
Introduction :

The goal of this exercise was to have a dataset and that each dataset have a row. But in reality the crucial thing was to manipulate the data inside this dataframe, and to work on different methods to change the data and information, but the trick was to have a parallel way of working with these methods i.a to run methods in simultaneous way, we also decided to implement the tests before development and the continuous development, that's what we had in mind.

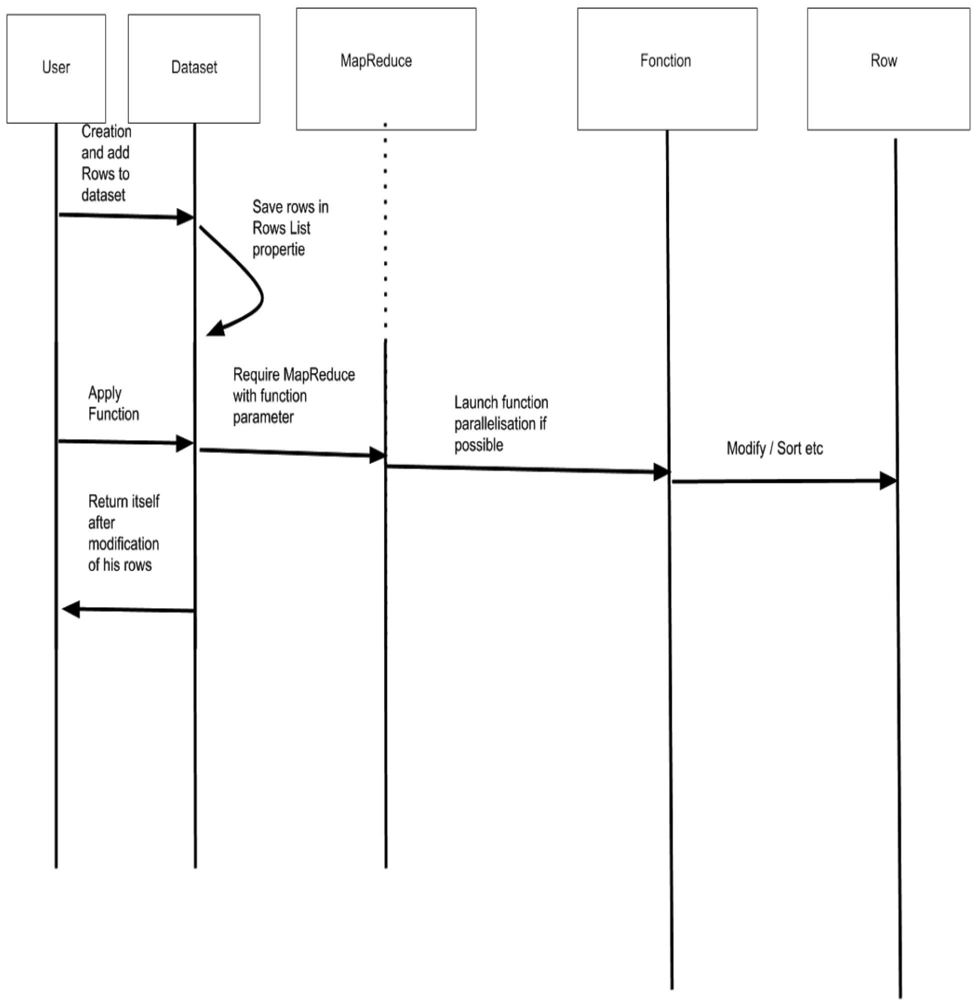
Design Patterns:

In this exercise also, the goal was not to implement design patterns, but it was rather a matter of knowing how to implement the multithreading process and the IPC. Hence why we didn't use any design patterns.

Class Diagram:



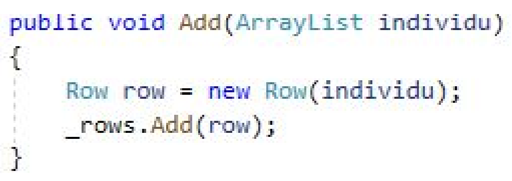
Sequence Diagram:


User
Dataset
MapReduce
Fonction
Row
Creation and add Rows to dataset
Save rows in Rows List propertie
Apply Function
Require MapReduce with function parameter
Launch function parallelisation if possible
Modify / Sort etc
Return itself after modification of his rows

Test before development:

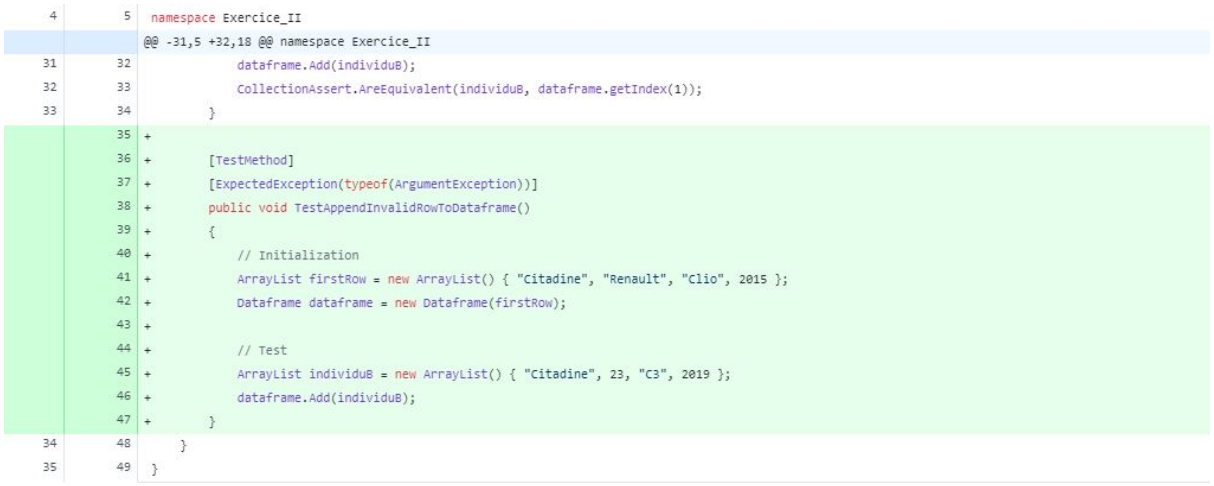
We make the implementation for this exercise step by step.

For example here we have a function who gets individu and adds them to the dataset.



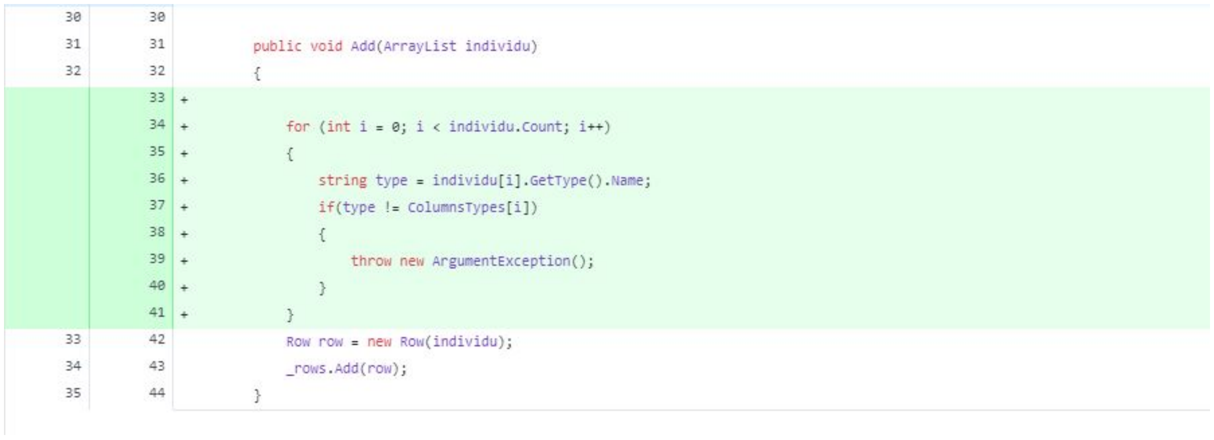
```
public void Add(ArrayList individu)
{
    Row row = new Row(individu);
    _rows.Add(row);
}
```

We decide to verify if the user adds data with the accurate type. So we create a unit test to fake the user insertion and raise errors when needed .



```
 namespace Exercice_II
@@ -31,5 +32,18 @@ namespace Exercice_II
             dataframe.Add(individuB);
             CollectionAssert.AreEquivalent(individuB, dataframe.getIndex(1));
         }
+
+        [TestMethod]
+        [ExpectedException(typeof(ArgumentException))]
+        public void TestAppendInvalidRowToDataframe()
+        {
+            // Initialization
+            ArrayList firstRow = new ArrayList() { "Citadine", "Renault", "Clio", 2015 };
+            Dataframe dataframe = new Dataframe(firstRow);
+
+            // Test
+            ArrayList individuB = new ArrayList() { "Citadine", 23, "C3", 2019 };
+            dataframe.Add(individuB);
+        }
     }
 }
```

After creating the unit test. The test failed because there is no error preventing the malicious insertion. So the goal is to detect bad insertion and raise error when one is detected. It was done by the following code.



```
 
         public void Add(ArrayList individu)
         {
+
+            for (int i = 0; i < individu.Count; i++)
+            {
+                string type = individu[i].GetType().Name;
+                if(type != ColumnsTypes[i])
+                {
+                    throw new ArgumentException();
+                }
+            }
             Row row = new Row(individu);
             _rows.Add(row);
         }
```