Project Report Design Pattern

Cyprien Beau – Mathieu Bué

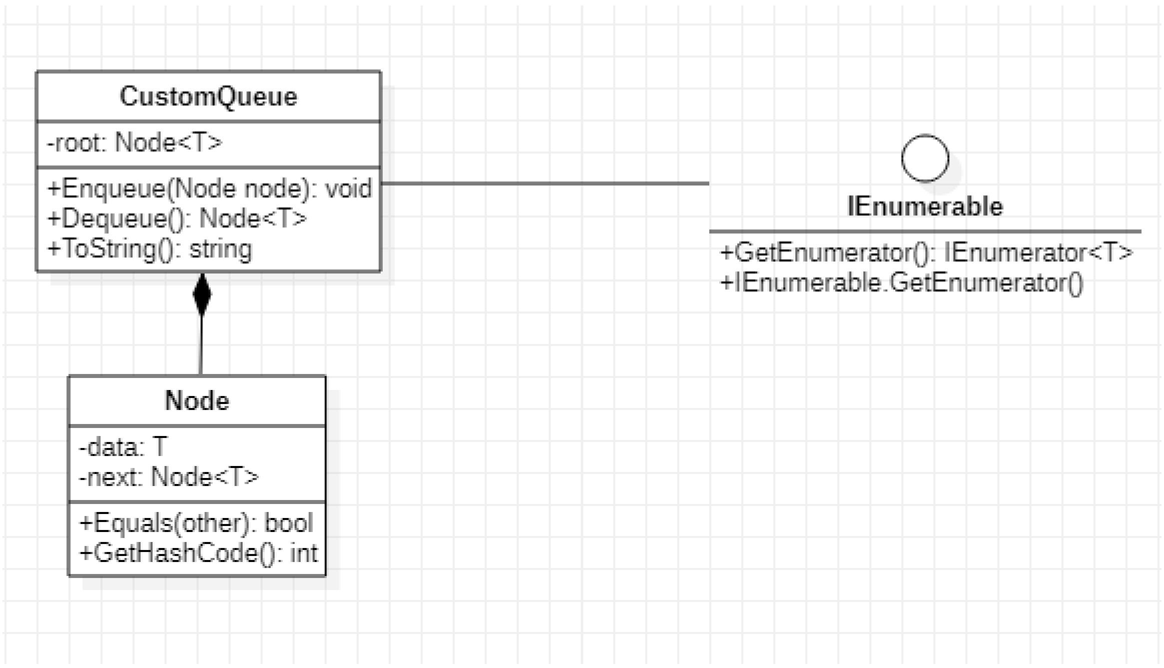
IOS 1

Exercise 1 – CustomQueue – Generics

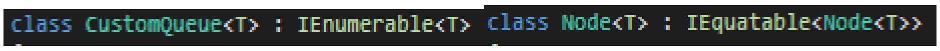
Introduction:

The first exercise asked us to create a custom queue with the FIFO concept (first in, first out). We had already done something similar in the previous TD. We reused the code created for the linked list and improved it to present all the needed methods. The list is composed of head node and the node class contains a pointer to the next node in the queue. Enqueueing means adding a node as the head and adding the previous head as his next node. Dequeuing implies the same process, we go to the second to last node and remove his pointer to the last element.

UML diagram



Generics class



```
class CustomQueue<T> : IEnumerable<T> class Node<T> : IEquatable<Node<T>>
```

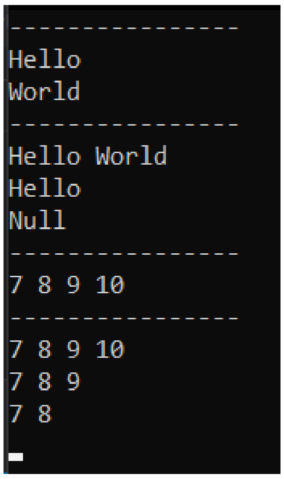
We have put the tag <T> after the class in order to create generics class.

Queue

We have created only two class in order to implement a simple linked list. And then we have created the Enqueue and Dequeue functions so that we have a functional queue.

Foreach method

Costuming the foreach method requires to implement the GetEnumetor method from IEnumerable interface. We have used the key word yield in order to avoid a list creation. The foreach method will print the data of all the queue's nodes.

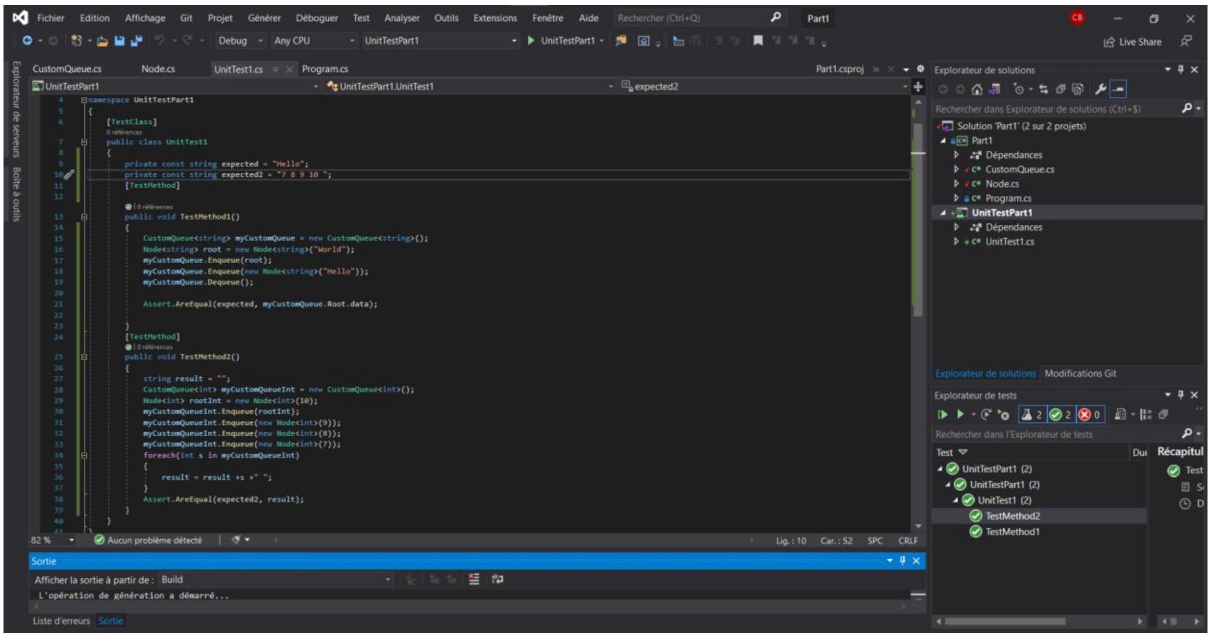


```
----------------
Hello
World
----------------
Hello World
Hello
Null
----------------
7 8 9 10
----------------
7 8 9 10
7 8 9
7 8
```

Here is an example of output. It shows first the element of our list by using the forereach method. Then we remove all the element in the queue and print all the elements in the queue. When null is returned, it means that there is no more element in the queue and trying to dequeue again won't work.

We did the same with integers as our code is using generics.

We also did unit testing on this exercise. We checked that our dequeuing method was working just fine. We also checked that the queueing method worked as we expected.



```
namespace UnitTestPart1
{
    [TestClass]
    public class UnitTest1
    {
        private const string expected = "Hello";
        private const string expected2 = "7 8 9 10 ";
        [TestMethod]

        public void TestMethod1()
        {
            CustomQueue<string> myCustomQueue = new CustomQueue<string>();
            Node<string> root = new Node<string>("World");
            myCustomQueue.Enqueue(root);
            myCustomQueue.Enqueue(new Node<string>("Hello"));
            myCustomQueue.Dequeue();

            Assert.AreEqual(expected, myCustomQueue.Root.data);

        }
        [TestMethod]
        public void TestMethod2()
        {
            string result = "";
            CustomQueue<int> myCustomQueueInt = new CustomQueue<int>();
            Node<int> rootInt = new Node<int>(10);
            myCustomQueueInt.Enqueue(rootInt);
            myCustomQueueInt.Enqueue(new Node<int>(9));
            myCustomQueueInt.Enqueue(new Node<int>(8));
            myCustomQueueInt.Enqueue(new Node<int>(7));
            foreach(int s in myCustomQueueInt)
            {
                result = result +s +" ";
            }
            Assert.AreEqual(expected2, result);
        }
    }
```

```
Aucun problème détecté
Afficher la sortie à partir de : Build
L'opération de génération a démarré...
```

```
UnitTestPart1 (2)
  UnitTestPart1 (2)
    UnitTest1 (2)
      TestMethod2
      TestMethod1
```

As the screen shot shows, it works perfectly fine.

Exercise 2 – MapReduce – Design patterns, Threads & IPC

Introduction:

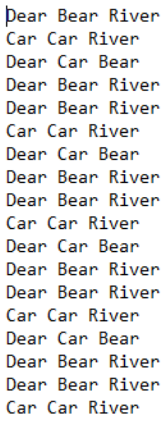
The second exercise asks us to implement an algorithm using multiprocessing. In the example, the user gives a text as an input and the code divide this input in different part to speed the process. As a result, it gives the number each word is found in the input.

In order to implement the **Map Reduce Count Process** we have chosen to return a dictionary which has the word as a key and the number of this word in the txt file as value. This value is set to one by default in our code.

We decided to create a certain number of threads based on the number of processors of the environment used.

Input

The input is a txt file named data.txt.



```
Dear Bear River
Car Car River
Dear Car Bear
Dear Bear River
Dear Bear River
Car Car River
Dear Car Bear
Dear Bear River
Dear Bear River
Car Car River
Dear Car Bear
Dear Bear River
Dear Bear River
Car Car River
Dear Car Bear
Dear Bear River
Dear Bear River
Car Car River
```

part of data.txt

Splitting

The splitting process start in the Parelle.ForEach(), in the method getResult() we split the lines of the txt file between the threads.

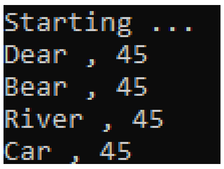
Mapping

In order to map the data in each thread we have used a local dictionary on the model of the final one. So, at the end of mapping, we have a dictionary with the number of occurrences of each word in the line treated by the thread.

Reducing

For updating the final dictionary, we have used the lock keyword which enables the threads to write one by one in the final dictionary.

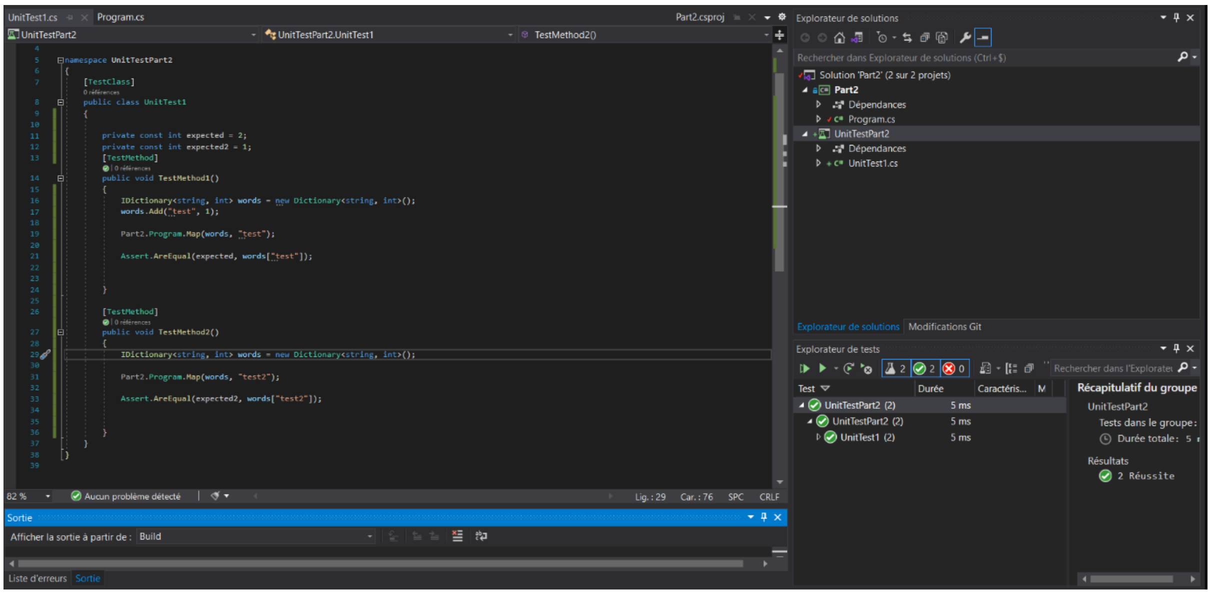
Output (Key: Word, Value: number of occurrences)



```
Starting ...
Dear , 45
Bear , 45
River , 45
Car , 45
```

Our code works perfectly for this exercise, but it can be improved. Indeed, we use a dictionary for comfort, but it is not the most adapted class. Using the Map class would pull all the problem related to integer vectors away. This is an aspect to take into consideration if we want to improve the code.

This exercise was no exception, we did some unit testing. We tried the two cases of our Map method. It should augment in the first case the value of the Key “test” and create a value for the key “test2” in the second case. We can see with the screen shot that it works fine as well.



```
namespace UnitTestPart2
{
    [TestClass]
    0 références
    public class UnitTest1
    {

        private const int expected = 2;
        private const int expected2 = 1;
        [TestMethod]
        0 références
        public void TestMethod1()
        {
            IDictionary<string, int> words = new Dictionary<string, int>();
            words.Add("test", 1);

            Part2.Program.Map(words, "test");

            Assert.AreEqual(expected, words["test"]);


        }

        [TestMethod]
        0 références
        public void TestMethod2()
        {
            IDictionary<string, int> words = new Dictionary<string, int>();

            Part2.Program.Map(words, "test2");

            Assert.AreEqual(expected2, words["test2"]);


        }
    }
}
```

```
82 %    Aucun problème détecté    Lig. : 29    Car. : 76    SPC    CRLF
Sortie
Afficher la sortie à partir de : Build
Liste d'erreurs    Sortie
```

```
Explorateur de solutions
Solution 'Part2' (2 sur 2 projets)
  Part2
    Dépendances
    Program.cs
  UnitTestPart2
    Dépendances
    UnitTest1.cs
```

```
Explorateur de tests
2    ✓ 2    ✗ 0
Test                    Durée    Caractéris...    M
UnitTestPart2 (2)       5 ms
  UnitTestPart2 (2)     5 ms
    UnitTest1 (2)       5 ms

Récapitulatif du groupe
UnitTestPart2
  Tests dans le groupe:
  Durée totale: 5
Résultats
  2 Réussite
```

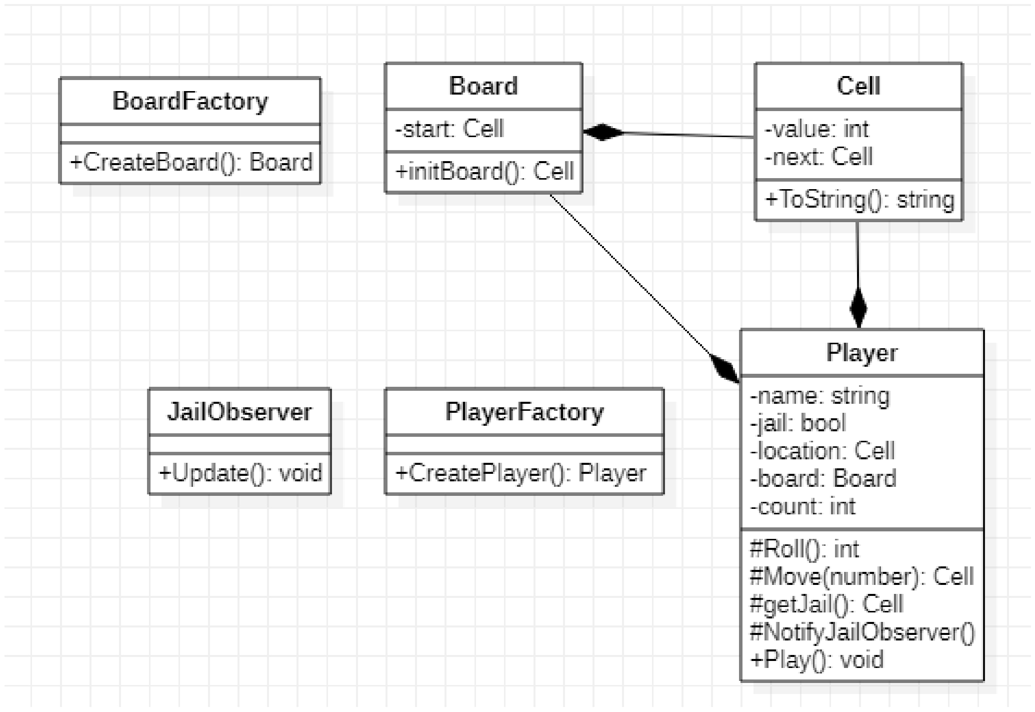
Exercise 3 – A Monopoly™ game

Introduction:

The third exercise is about simulating a simplified version of the Monopoly board game. In this exercise, no precise design patterns are asked. We can use every pattern we think useful to the project.

The statement gives many constraints to respect and rules to add to the game. There are many ways of doing the project. A simple class Player with a position parameter could be enough to simulate the board but we did otherwise. Indeed, we decided to code the board using what we did in the previous exercise. We thought that if an improvement was needed on the board (like adding the cities for instance), it would be easier to implement.

UML diagram



The Board is circular linked list, when you pass the 39th cell, you are on the cell n°0. The value of a cell is the n° of the cell on the board. The function **initBoard** create a board with all the cells and links the last one to the first one.
The **Roll** function return a random list with 2 integers between 1 and 6.
The **Move** function changes the location of a player by the number given in input.
The **getJail** function returns the cell of the jail.
The **NotifyJailObserver** calls the update method of the **JailObserver** class.
The function **Play** enables a player to play a turn and implements all the conditions about jail (3 double rolls in a row or move on the case 30th). The function prints the starting position and the ending position.

Once the code was implemented, we could think about 2 main design patterns to add.

<u>Observer Design pattern</u>

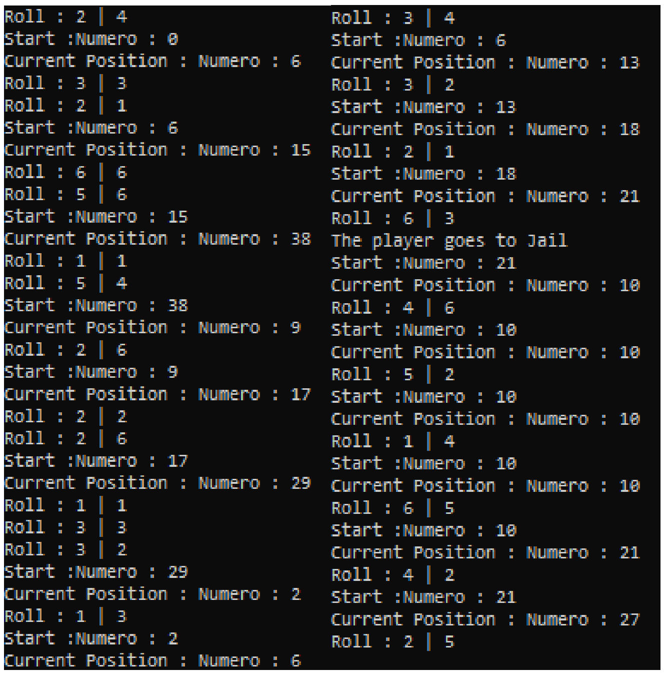
We have used the observer design pattern in order to notify in the console if a player is jailed.
When a player is jailed the **Play** function of the Player class calls **NotifyJailObserver** which calls Update of the **JailObserver** Class. The function Update notifies that the status of the player changes from free to jailed.

<u>Factory Design pattern</u>

In order to remove the creation keywords, we have used the factory design pattern.

In the main method we call the methods **CreatePlayer** and **CreateBoard** instead of initializing a new Board and player.

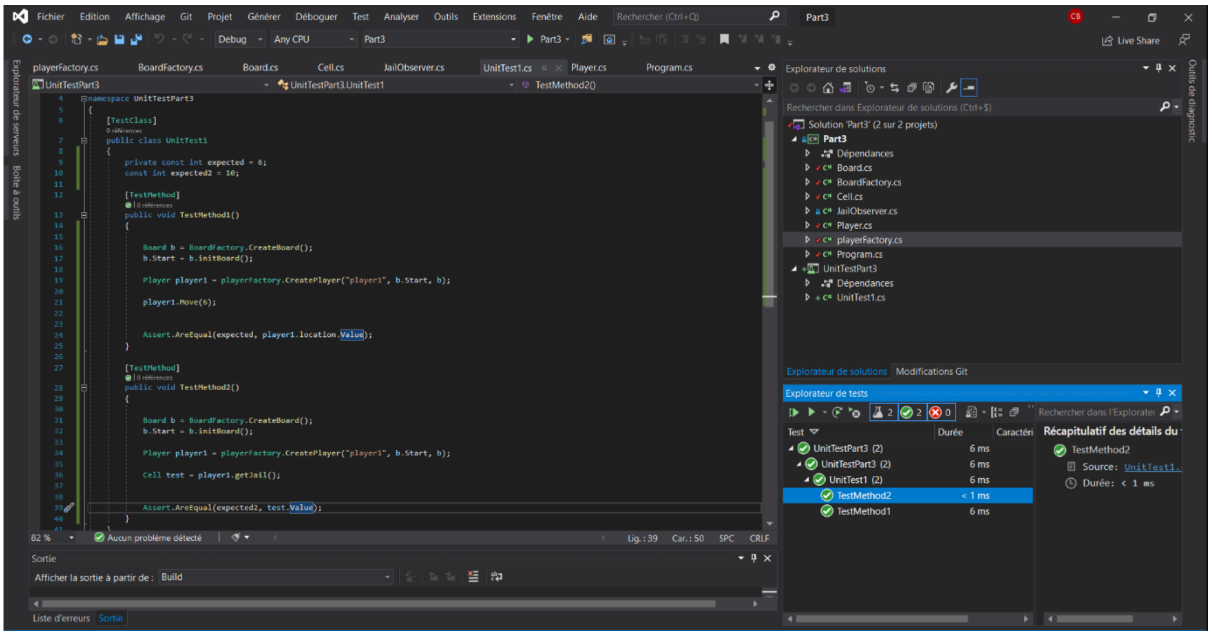
We did not let the user create player and play in our code we just created a player and let him throw the dice many times. Each time, the console shows the result of the dices, the starting cell and the last one. When in jail, the player needs to have 2 dices with the same value to leave or to wait for 3 turns. Here is an example of the output received. Here we can see that once a player goes to jail, he missed 3 times and then leaves the jail on the fourth turn. We can see that the observer class indicated that the player went to jail (he was on the 30th cell).



```
Roll : 2 | 4
Start :Numero : 0
Current Position : Numero : 6
Roll : 3 | 3
Roll : 2 | 1
Start :Numero : 6
Current Position : Numero : 15
Roll : 6 | 6
Roll : 5 | 6
Start :Numero : 15
Current Position : Numero : 38
Roll : 1 | 1
Roll : 5 | 4
Start :Numero : 38
Current Position : Numero : 9
Roll : 2 | 6
Start :Numero : 9
Current Position : Numero : 17
Roll : 2 | 2
Roll : 2 | 6
Start :Numero : 17
Current Position : Numero : 29
Roll : 1 | 1
Roll : 3 | 3
Roll : 3 | 2
Start :Numero : 29
Current Position : Numero : 2
Roll : 1 | 3
Start :Numero : 2
Current Position : Numero : 6
```

```
Roll : 3 | 4
Start :Numero : 6
Current Position : Numero : 13
Roll : 3 | 2
Start :Numero : 13
Current Position : Numero : 18
Roll : 2 | 1
Start :Numero : 18
Current Position : Numero : 21
Roll : 6 | 3
The player goes to Jail
Start :Numero : 21
Current Position : Numero : 10
Roll : 4 | 6
Start :Numero : 10
Current Position : Numero : 10
Roll : 5 | 2
Start :Numero : 10
Current Position : Numero : 10
Roll : 1 | 4
Start :Numero : 10
Current Position : Numero : 10
Roll : 6 | 5
Start :Numero : 10
Current Position : Numero : 21
Roll : 4 | 2
Start :Numero : 21
Current Position : Numero : 27
Roll : 2 | 5
```

Once more, we did unit testing on this code. We tried to verify the method Move of the player and the method GetJail. The first should just move the player around the board (i.e. modifying its location). The second one was just to check that the cell returned by the function was indeed the 10th cell of the board AKA the jail cell.



```
namespace UnitTestPart3
{
    [TestClass]
    public class UnitTest1
    {
        private const int expected = 6;
        const int expected2 = 10;

        [TestMethod]
        public void TestMethod1()
        {

            Board b = BoardFactory.CreateBoard();
            b.Start = b.initBoard();

            Player player1 = playerFactory.CreatePlayer("player1", b.Start, b);

            player1.Move(6);


            Assert.AreEqual(expected, player1.location.Value);
        }

        [TestMethod]
        public void TestMethod2()
        {

            Board b = BoardFactory.CreateBoard();
            b.Start = b.initBoard();

            Player player1 = playerFactory.CreatePlayer("player1", b.Start, b);

            Cell test = player1.getJail();


            Assert.AreEqual(expected2, test.Value);
        }
```

```
Test                     Durée
UnitTestPart3 (2)        6 ms
  UnitTestPart3 (2)      6 ms
    UnitTest1 (2)        6 ms
      TestMethod2        < 1 ms
      TestMethod1        6 ms

Récapitulatif des détails du
  TestMethod2
    Source: UnitTest1.
    Durée: < 1 ms
```

Conclusion : Design Pattern and Software Development

To conclude on this project and on this course, we will say that it was and will be challenging. Indeed having to implement design pattern into our code change a lot of things in our way of approaching the exercise. We learnt a lot in this course and this project (even though it is hard sometimes to see the real use of a design pattern in small projects).

We are proud of the project we are filing on Brightspace even if it can still be improved. Indeed, we know that other design patterns can be used in the 3rd exercise (state pattern for the jail status for instance), that using the Map class in the second is a way to reduce compatibility errors.

We hope you enjoyed reading this report as much as we add fun writing and coding it.

Cyprien Beau & Mathieu Bué