Alix JACQUEMIN

Louis ATALLAH

PROJECT

Table des matières

Exercise 1 – CustomQueue – Generics

1. Introduction:

In this exercise, we have to implement a queue which is based on the FIFO process (First In First Out).

Moreover, we need to design a foreach functionnality for the queue. We have realized that via the interfaces.

2. Design Hypotheses:

This exercise is formed by 2 classes : the Node class and the Custom Queue class.

The Node class defines a node with its parameters (its data and the node it is pointing to).

The CustomQueue() class is defined by a start and an end. The start is the last enqueued node and the end is the next node to be dequeued.

We have redefined the bool Equals method in order to compare according to the different types with the GetHashCode(). IEquatable interface is required to override it.

```
public bool Equals(T n)
{
    if (n == null)
    {
        return false;
    }
    return this.GetHashCode() == n.GetHashCode();
}
```

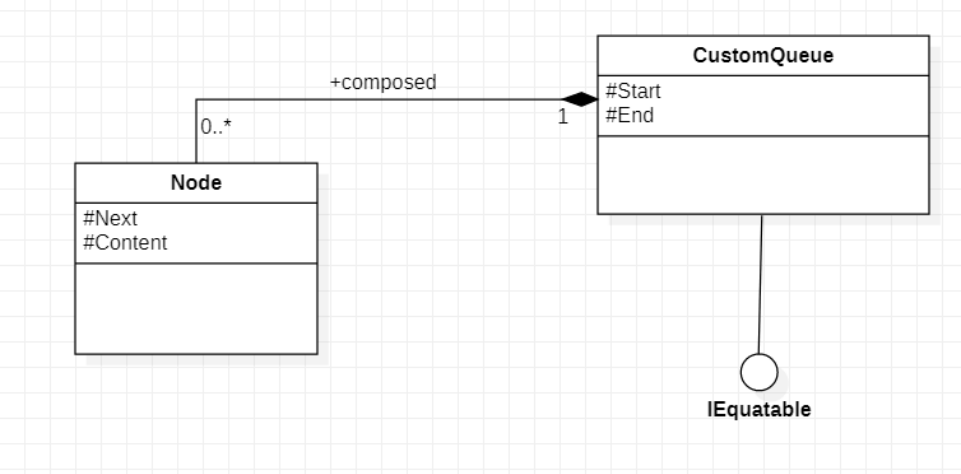
To enable the use of the Custom Queue in a foreach loop :

In order to add this characteristic, we have implemented the GetEnumerato() method from the IEnumerator interface.

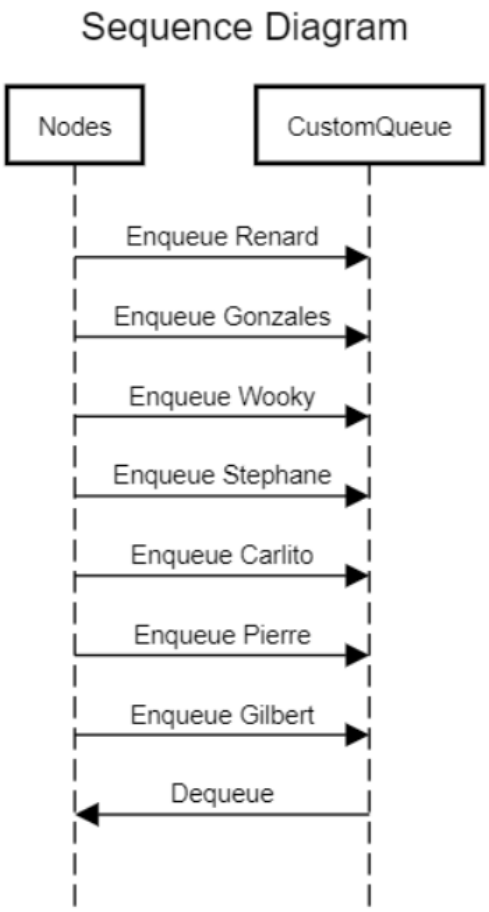
```
// implementation for the foreach loop
1 référence
public IEnumerator<T> GetEnumerator()
{
    Node<T> current = this.start;

    while (current != null)
    {
        yield return current.Content;
        current = current.Next;
    }
}
```

3. UML Diagrams :

Class Diagram :



Sequence Diagram **:**



In this example after the Enqueue we get the elements in this order: Gilbert Pierre Carlito Stephane Wooky Gonzales Renard

After the Dequeue, the elements in the queue are: Gilbert Pierre Carlito Stephane Wooky Gonzales

4. Test Cases:

Below, we can see an enqueue test. By enqueuing many nodes, a queue is created . Then, we print the queue to check out whether the process has been well respected.

```
static void Main(string[] args)


{
    Node<int> n1 = new Node<int>(1, null);
    Node<int> n2 = new Node<int>(2, null);
    CustomQueue<int> q = new CustomQueue<int>();
    q.Enqueue(n1);
    q.Enqueue(n2);
    q.Enqueue(new Node<int>(3, null));
    q.Enqueue(new Node<int>(4, null));
    q.PrintQueue();
```

```
4 -> 3 -> 2 -> 1 ->  X
```

As we can see, the queue has been successfully created.

Now let's try to deqeue an element and implement the queue in the foreach loop :

```
    q.Dequeue();
    Console.Write("After the dequeue : ");
    q.PrintQueue();
    foreach(int n in q)
    {
        Console.Write(n +" -> ");
    }
    Console.WriteLine("X");
    Console.ReadKey();
```

```
After the dequeue : 4 -> 3 -> 2 ->  X
```

1 has been removed as it was the first node in the queue.

Let's print the queue by using a foreach loop. It prints the same as the PrintQueue() method does :

```
4 -> 3 -> 2 -> X
```

It means that our queue can be implemented in a foreach loop.

Exercise 2 – Map Reduce – Design Patterns, Threads & IPC

1. Introduction:

In this problem, we need to implement a MapReduce program, using multithreading and IPC techniques. The mapreduce program is composed of a map procedure to filter and sort the data, and of a reduce procedure to count and sum up.

2. Design Hypotheses:

In Our program, we used the Parallel.ForEach, from the System.Threading.Tasks namespace. This allows us to create mapping functions that can execute in parallel.

We also use some Blocking Collections (from the System.Collections.Concurrent namespace) which allows us to add and remove objects during processing of multiple threads. The blocking collection can manage timing differences. When the collection is empty, it can block until new items are added or stop processing once all items have been processed and the collection is marked as complete. We also used a ConcurrentBag (which is a thread-safe, unordered collection of objects).

We also used a concurrent Dictionary, which is also thread safe, and acts as a key-value pair repository for the final reduction results in our process. The Concurrent Dictionary is ideal for a reduction.

To implement a solution, we have realized 4 methods, one for each step of the process, as it follows : Splitting – Mapping – Shuffling – Reducing.

First, we need to split the data, to do so we call Splitting which is an IEnumerable interface with a string type because of the conversion of the data into string. Thanks to Splitting we poduce some word blocks that we return one by one. We don’t have to wait to return all in once, when the first block is created the Mapping process starts while other word blocks are produced.

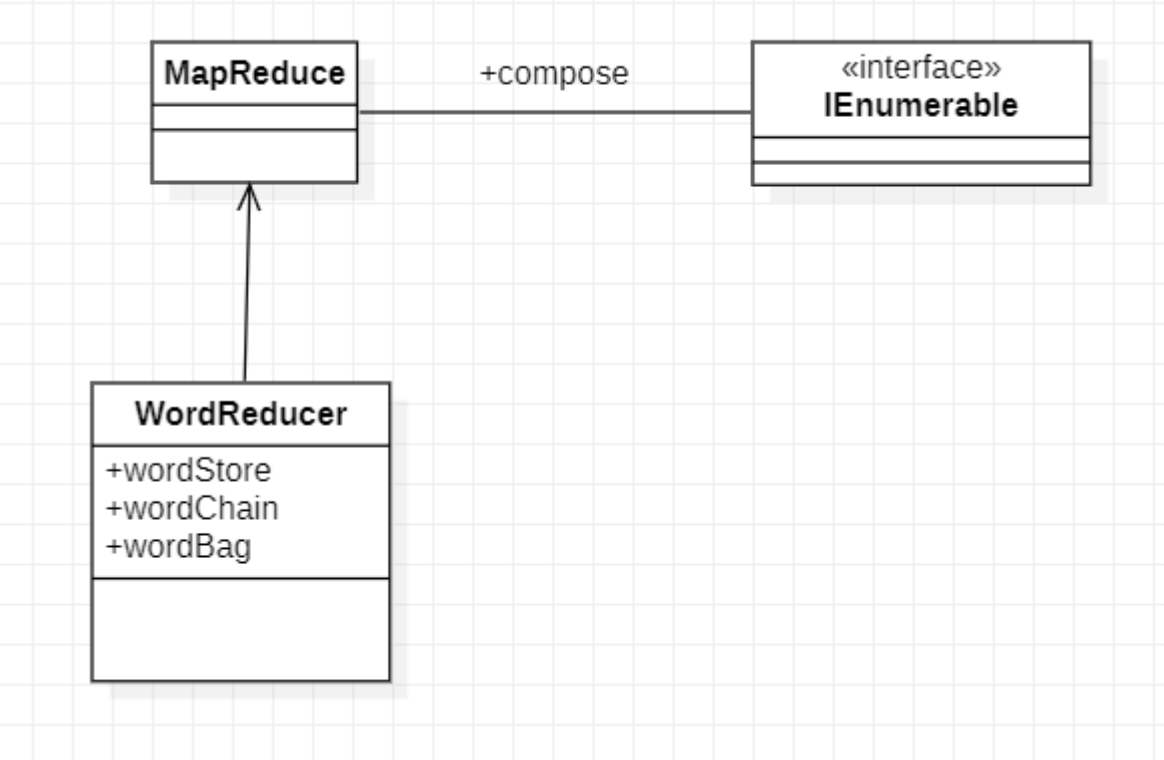
Second, the Mapping method is used to separate each word in the word block and store them in a safe-thread list called wordChain.

Third, once every word in the dataframe are separated and stored in the same list, we implement the Shuffling method with a parallel loop to cast each word in the wordStore thread-safe “SortedList”. If there are some duplicates entries the value is incremented by 1. If not the word is added with the value 1.

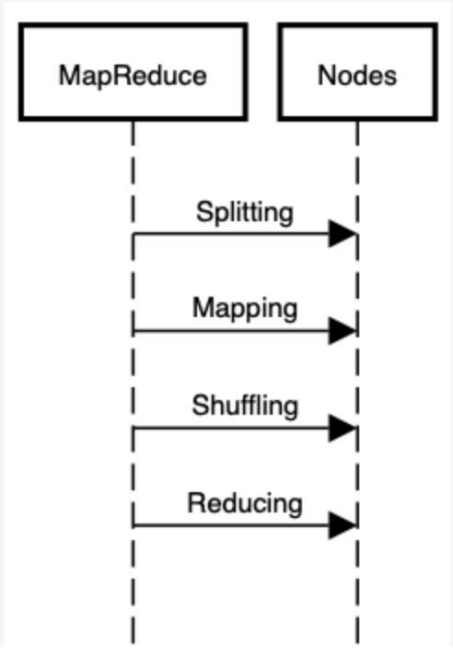
The Reducing method as written previously is used to set up the process, calling each function to organize the MapReduce functionnality.

3. UML diagrams

Class diagram



Sequence diagram



4. Test Cases

In our produceWordBlocks function (which produce 100 characters or less text parts, we use a YieldReturn. This allows us to have all phases of mapping process executed

asynchronously, in order for both mapping and reduction worker threads to work together. This could have been also accomplished by using backgroun threads (alone or with the YieldReturn).

This returns immediatly each identified text block instead of returning all the blocks at the end of the process. This also allow other worker threads to begin working on mapping the blocks of text.

The final results of the mapping stages are added to a Blocking Collection.

In our MapWords function, we use a Parallel.ForEach mapping function, using multiple worker threads to identify and clean words from the blocks of input text provided. This process begins as soon as the first block of text has been identified with the produceWordBlocks function.

Once all individual words have been identified from all word blocks, the wordChunks Blocking Collection is notified that no more words will be added to the collection. This is important because if the Blocking Collection becomes empty during processing (because of the reducing), the collection will continue to block or wait until the CompleteAdding() method is called or additional items are added in the collection.

We have tested our solution with a .txt file which contains many different words as shown below :

```
hello 1 1 1 1 2 3 4 5
```

As we can see there are many duplicates entries, hence we expect to have no duplicate keys in the final list.

Indeed, the result is what we have predicted :

```
Key = hello, Value = 1
Key = 3, Value = 1
Key = 1, Value = 4
Key = 5, Value = 1
Key = 4, Value = 1
Key = 2, Value = 1
```

Several 1 keys but only an unique emplacement assigned with 4 as value because of the four ones given by the dataframe.

Exercise 3 – – A Monopoly™ game – Design Patterns:

1. Introduction:

To implement this Monopoly™ game, we have to deal with the several rules of the historical game and the implementation of a design pattern. The skeleton of the algorithm and the design pattern chosen are explained in the following section.

2. Design Hypotheses:

For this Exercise, we decided to base our program on a Behavioural Design Pattern : the Observer Pattern. The Algorithmic part and the display part are completely separated in the code, thus in order to make one communicate with the other, we go through an observer. The game takes care of sending events and the Displayer will take care of retrieving and displaying consequences.

We use several Classes in our Program:

- The Constant Class is used to keep all constants used in our program in one class, we can modify them if needed (if we want to change the rules of the game, for example the number of dices used)
- The Dice Class is used to create a random number for the dice roll
- The MonopolyEvent Class shows an enumeration of all Events happening in the game:

  PlayerMoved : When a player move

  PlayerJailed : When A player Goes to Jail

  PlayerRolledDice : When A Player Rolled the Dice

  PlayerRolledDouble : When A Player Rolled a Double

  PlayerPardonned : When A Player Can Go Out Of Jail

  PlayerRolledTooMuchDouble : When A Player Rolled too Many Doubles

  PlayerIsForcedToLeaveJail : When the Player leaves Jail because they didn't roll a double 3 times in a row

  PlayerWin : When The Player wins And the Game has ended

All those events are Observed by the same Observer in the Displayer Class, excepted for the **PlayerWin** event that is also managed by the Monopoly Class.

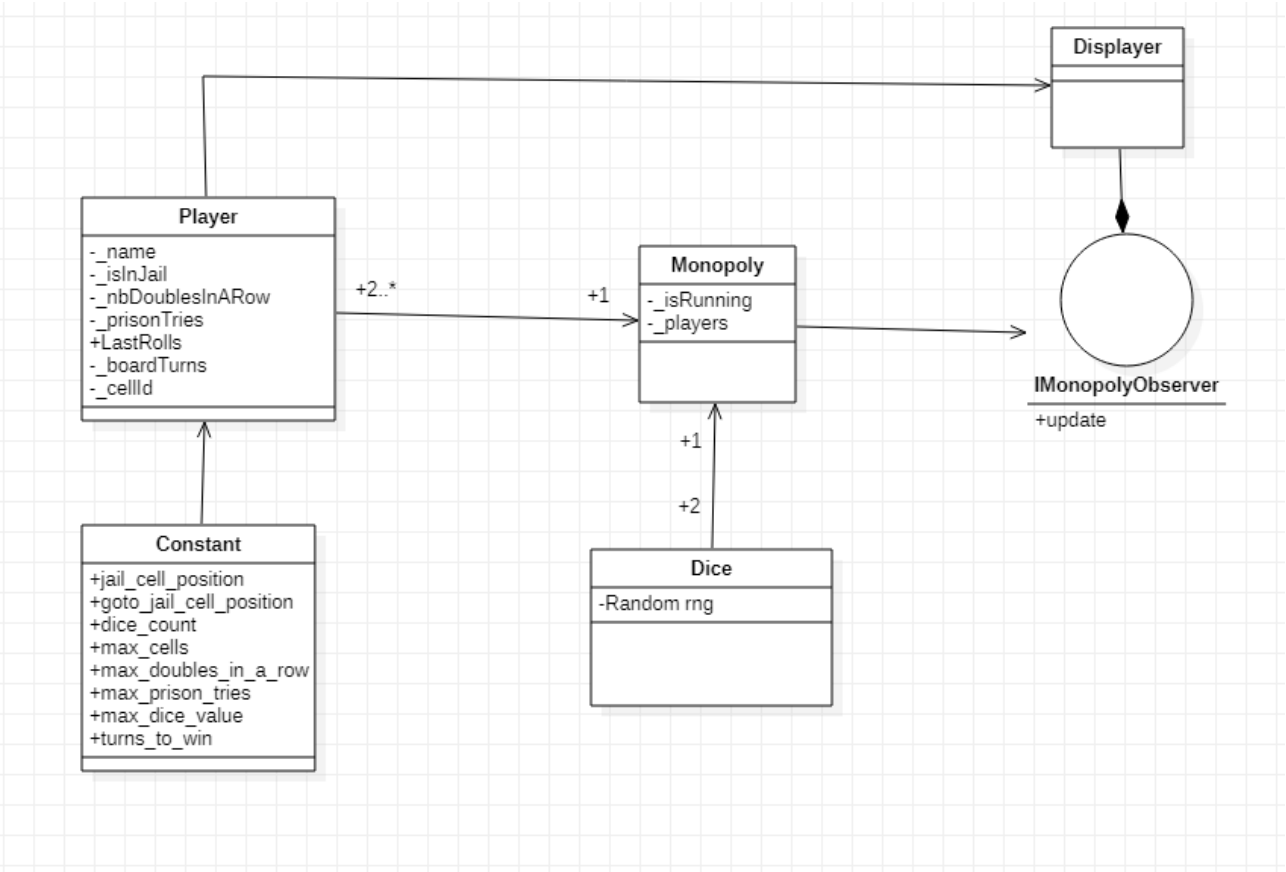
- The Displayer Class : Observes events notified in the Player Class in order to Display what happens in the game.

- The Monopoly Class: Initiate and ends the game. As the **PlayerWin** event is also notified in this Class, the Monopoly Class is also an Observer: the game ends when the **PlayerWin** event happens.
- The Player Class: The Player Class represents the game itself, it's in this Class that the events are notified to both the Displayer and the Monopoly classes.

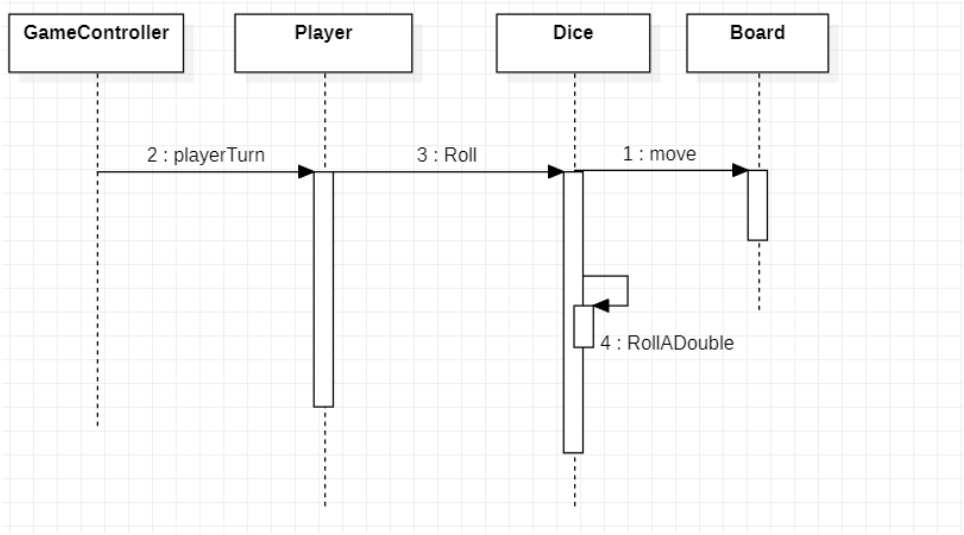
The IMonopolyObserver Interface is the Interface which needs to be implemented by the Observers. Thus the notify function is implemented in both the Displayer and Monopoly Class.

3. UML diagrams

Class diagram



Sequence diagram



4. Test Cases

By Launching the Game a few times, we tested a few possibilities:

- When A Player does a Double, and thus plays twice in a row:

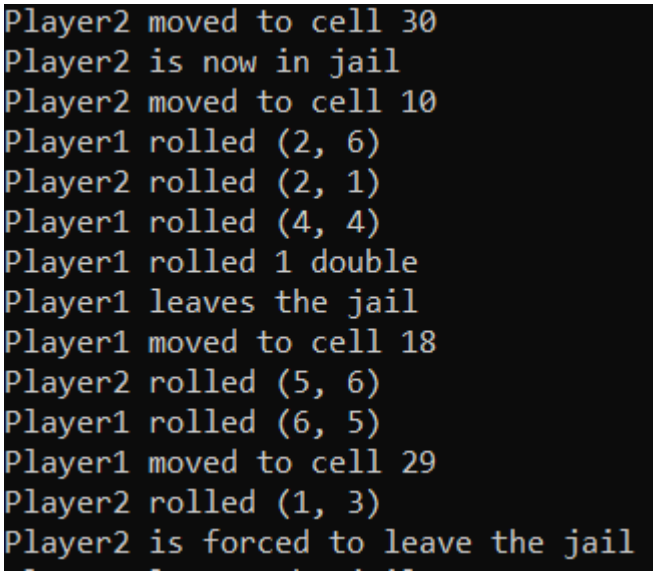
```
Player1 rolled (4, 4)
Player1 rolled 1 double
Player1 moved to cell 22
Player1 rolled (1, 2)
Player1 moved to cell 25
Player2 rolled (1, 5)
Player2 moved to cell 9
```

- When A Player goes to jail, and exits because he rolled a double:

```
Player1 moved to cell 30
Player1 is now in jail
Player1 moved to cell 10
```

```
Player1 rolled (4, 4)
Player1 rolled 1 double
Player1 leaves the jail
```

- When When a Player lands in jail after landing on the cell 30 and leaves Jail because he failed to roll a double three times:



- When a Player lands in jail after having done too many doubles: