

Final Projet Report - DPSDP

Exercise 1 – CustomQueue – Generics

1. Introduction (a brief description of the problem)

The goal of this exercise is to program a custom Queue data structure. It should be formed by linked nodes that carry content of generic data type, and we need to implement a FIFO policy. Moreover, it must provide all the needed characteristics so to be used by a C# foreach loop. We were asked to use our work of the first class.

2. Design Hypotheses (specification and explanation of details not explicitly indicated in the project assignments, but required for the solution's design and implementation)

The main goal of the exercise is to build a queue. A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

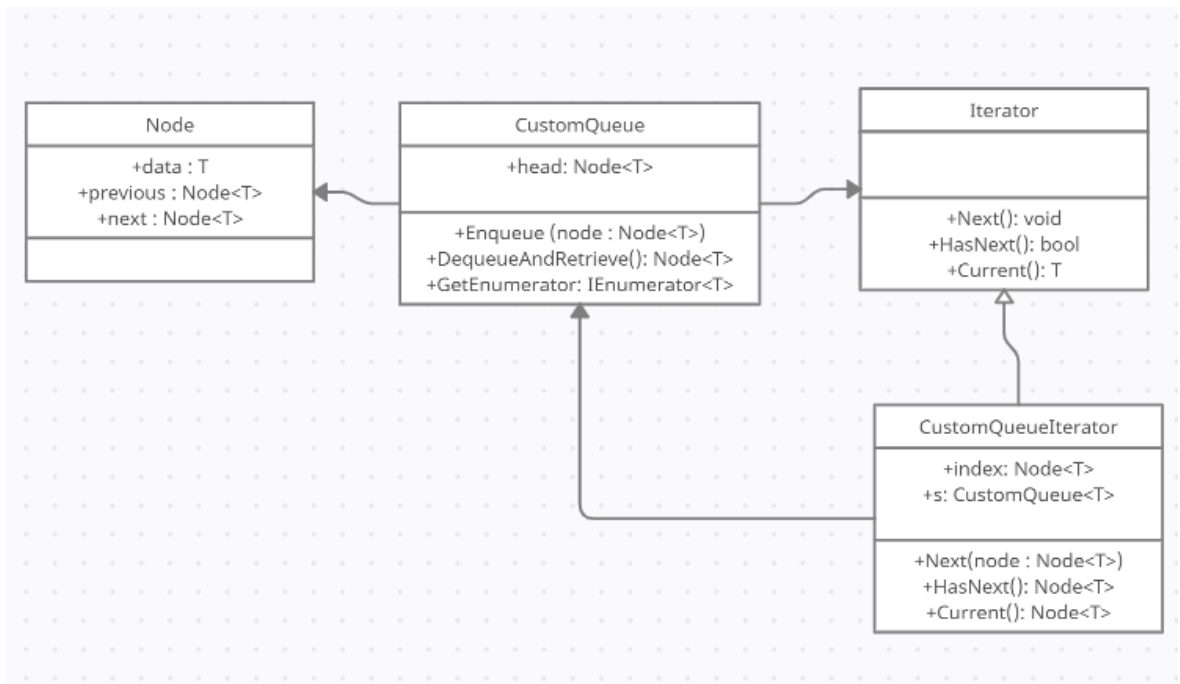
Here we have a collection, an “enumerable”: The Queue, and we would like to modify it and access to it. This is why we decided to use the Iterator Pattern. In fact, it provides a way to access the elements of an aggregate object without exposing its underlying representation. This allow the code to be more secure and to be more maintainable.

The whole point of designing an Iterator pattern is to use an iterator. It is an object making possible to browse the elements of another object which is an enumerable (an array, a list, a tree or here a queue). The iterator cycles through the container in the standard fashion, often starting with the first item, then moving to the next until arriving at the last.

As in most of the design patterns, we used generics in this exercise. Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. So here, it is possible to create queues of integers as well as queues of strings.

3. UML diagrams

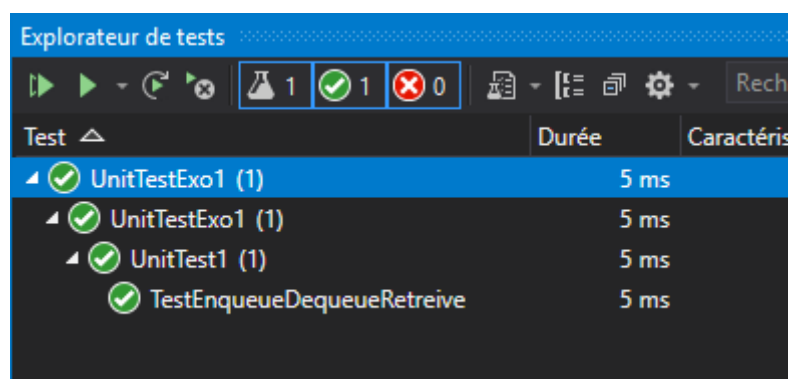
a. Class diagram of the solution



4. Test cases (description of the employed techniques ; specification of at least 2 executed test cases, a sample of input data (if present), expected and obtained results)

To test our program, we created a unit testing solution.

In this test, we tried some methods to verify if our enqueue and dequeue methods were working. Here are the returned results:



As you can see, everything is working well.

Exercise 2 – MapReduce – Design patterns, Threads & IPC

1. Introduction (a brief description of the problem)

The goal of this exercise was to program the MapReduce Design Pattern. It was invented by Google, and it uses parallel and distributed calculations to perform very large sets of datas, typically greater than 1 terabyte in size.

2. Design Hypotheses (specification and explanation of details not explicitly indicated in the project assignments, but required for the solution's design and implementation)

First, MapReduce is a framework for processing parallelizable tasks using a number of computers (nodes, collectively referred to as a cluster or grid), typically employed to elaborate or conduct operations on large sets of data. A MapReduce program is composed of a map procedure (or method) – which could also perform preliminary filtering and sorting over data (such as sorting students by first name into queues, one queue for each name) – and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

In this exercise we have only one class, and everything will take place in this class “Program”. We created different method to follow the process shown with a diagram.

The splitting function allows us to read an external text file and to store it as an array of string, each string being a line.

The Map method is calculating the number of occurrences of each word per lines. It returns a dictionary, the key being the word and the value being the number of apparitions. This method also cleans the array as it removes all the dots, the commas, the semi-columns,...

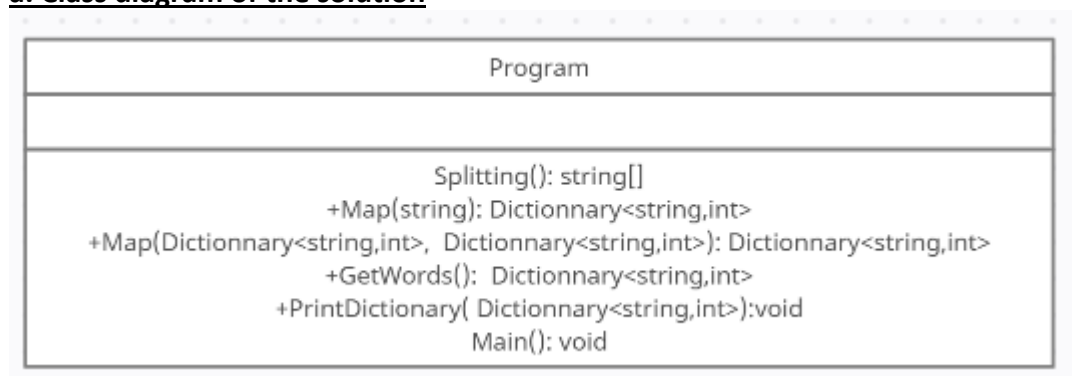
The reduce Method scans all the dictionaries given by the Map method and merge them to create a dictionary for the total counter.

PrintDictionnary is simply the method to show the dictionary in the Console.

And Finally, the GetWords method uses all the different methods to link them. It is using multithreading with the Parallel.ForEach method to make all the lines treat at the same time.

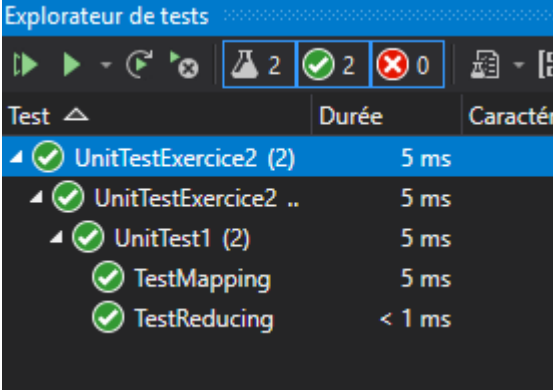
3. UML diagrams

a. Class diagram of the solution



4. Test cases (description of the employed techniques ; specification of at least 2 executed test cases, a sample of input data (if present), expected and obtained results)

To test our program, we created a unit testing solution. We created 2 method, one for checking if the mapping part was working, and another one to check if the reducing part was working. Here are the returned results:



The screenshot shows the 'Explorateur de tests' (Test Explorer) window in Visual Studio. At the top, there are icons for running tests, a summary bar showing 2 tests passed (green checkmark) and 0 failed (red X), and a list of tests. The tests are organized in a tree structure:

Test	Durée	Caractéristiques
✔ UnitTestExercice2 (2)	5 ms	
✔ UnitTestExercice2 ..	5 ms	
✔ UnitTest1 (2)	5 ms	
✔ TestMapping	5 ms	
✔ TestReducing	< 1 ms	

As you can see, everything is working well.

Exercise 3 – A Monopoly™ game – Design patterns

1. Introduction (a brief description of the problem)

The goal of this exercise is to simulate a simplified version of the Monopoly game. We are given a summary of the rules that we should respect in our design and implementation.

2. Design Hypotheses (specification and explanation of details not explicitly indicated in the project assignments, but required for the solution's design and implementation)

We use simply a Player class to represent our game because we just need player's position in order to make the game flow. Our class player has a 'position' attribute between 0 and 39 corresponding to the case he's on, a 'pseudo' attribute, a currentState attribute that defines his state during the game (detailed below) and finally a 'jailPlayCounts' attribute to count how many turns he has been in jail.

To implement the game, we figured out that the State Design Pattern would be very useful in order to define and apply different behaviors to the players according to their state in the game. Indeed, we have identified two different states that players may be in during the game, the 'default state', where the player can move 'freely' on the board and the 'jail state' when the player is in jail and can't move on the board.

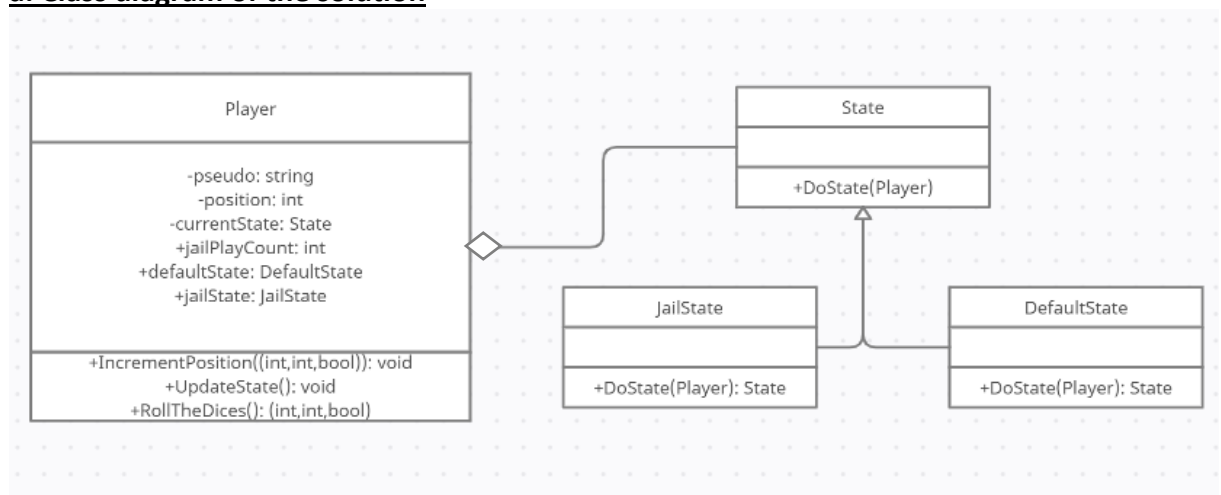
So we a State base class with a DoState() and two derived classes for the two states mentioned in which we describe in a state-specific DoState() function the behavior of the player in that state according to the dices.

The state of the player updates itself after each turn.

We added options to choose how many players will play the game, and how many turns there will be before stopping the game as there is no real end points for that version of the games.

3. UML diagrams

a. Class diagram of the solution



4. Test cases (description of the employed techniques; specification of at least 2 executed test cases, a sample of input data (if present), expected and obtained results)

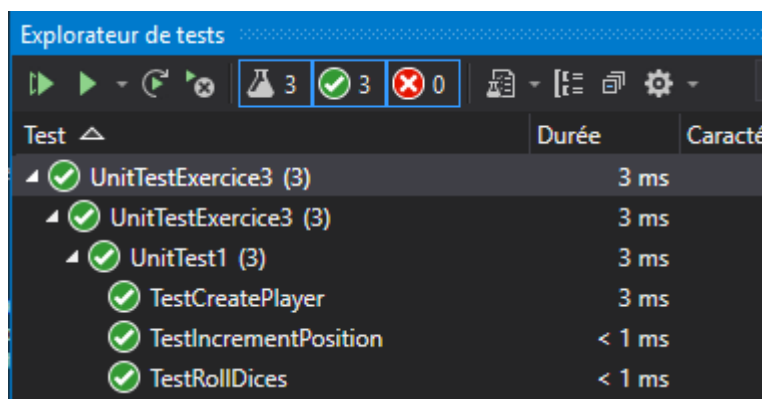
To test our program, we created a unit testing solution with different tests:

TestCreatePlayer() : We create a player and test if its variables have the values that we specified or the default ones.

TestRollDices() : We test our RollTheDices() method of our Player class and verify if the output is in the correct type (tuple) and if the numbers for each dices are between 1 and 6.

TestIncrementPosition(): We test our IncrementPosition() method from our Player class and verify that the player's position has correctly been modified.

Here are the returned results:



As you can see, everything is working well.