Design Patterns and Software Development Process

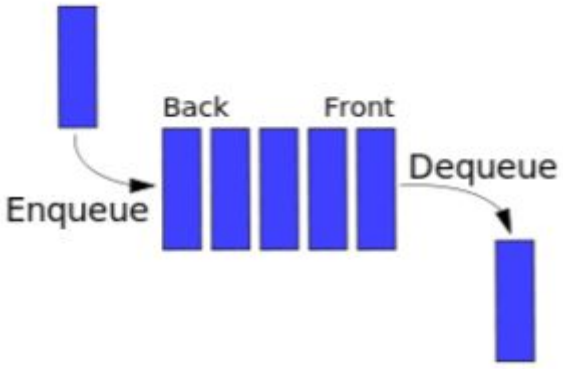
Exercise 1 – CustomQueue – Generics

1. Introduction

We have to design and develop a CustomQueue data structure, formed by linked nodes that carry content of generic data type.

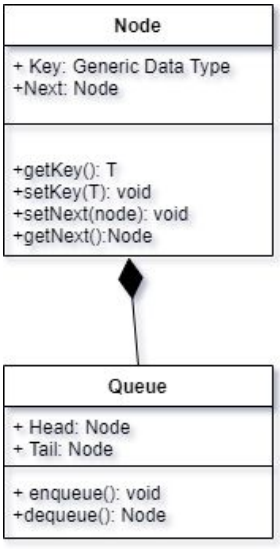
2. Design Hypotheses

New nodes are added on the back of the queue (enqueueing), and retrieved (and deleted, one retrieved) from the top of the queue (dequeuing), so to implement a FIFO policy, as shown here below:

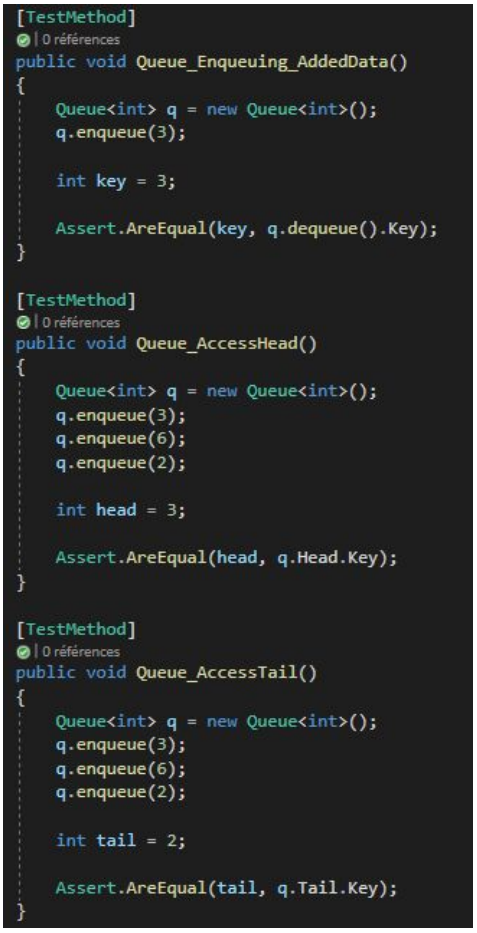


And it has to provide all the needed characteristics so to be used by a C# foreach loop, i.e it has to implement the IEnumerable interface.

3. UML diagram



4. **Test cases**



```
[TestMethod]
0 références
public void Queue_Enqueuing_AddedData()
{
    Queue<int> q = new Queue<int>();
    q.enqueue(3);

    int key = 3;

    Assert.AreEqual(key, q.dequeue().Key);
}

[TestMethod]
0 références
public void Queue_AccessHead()
{
    Queue<int> q = new Queue<int>();
    q.enqueue(3);
    q.enqueue(6);
    q.enqueue(2);

    int head = 3;

    Assert.AreEqual(head, q.Head.Key);
}

[TestMethod]
0 références
public void Queue_AccessTail()
{
    Queue<int> q = new Queue<int>();
    q.enqueue(3);
    q.enqueue(6);
    q.enqueue(2);

    int tail = 2;

    Assert.AreEqual(tail, q.Tail.Key);
}
```

We have three test cases: the first situation is to enqueue nodes and get the tail by dequeuing the queue and see if the value is the good one.
The second test tries to access the head of the queue and the last test tries to access the tail.

Exercise 2 – MapReduce – Design patterns, Threads & IPC

1. **Introduction**

We have to implement a program that simulates a network of elaborating nodes where each node can execute an arbitrary task (i.e a function) on a given data frame received in input.

The data is supposed to be in the right format expected by the task. We will use the MapReduce paradigm which is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware).

Processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead.

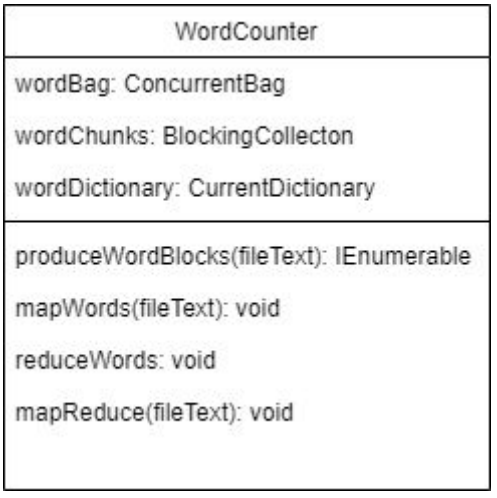
2. Design Hypotheses

A MapReduce framework (or system) is usually composed of three operations (or steps):

1. Map: each worker node applies the map function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of the redundant input data is processed.
2. Shuffle: worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node.
3. Reduce: worker nodes now process each group of output data, per key, in parallel.

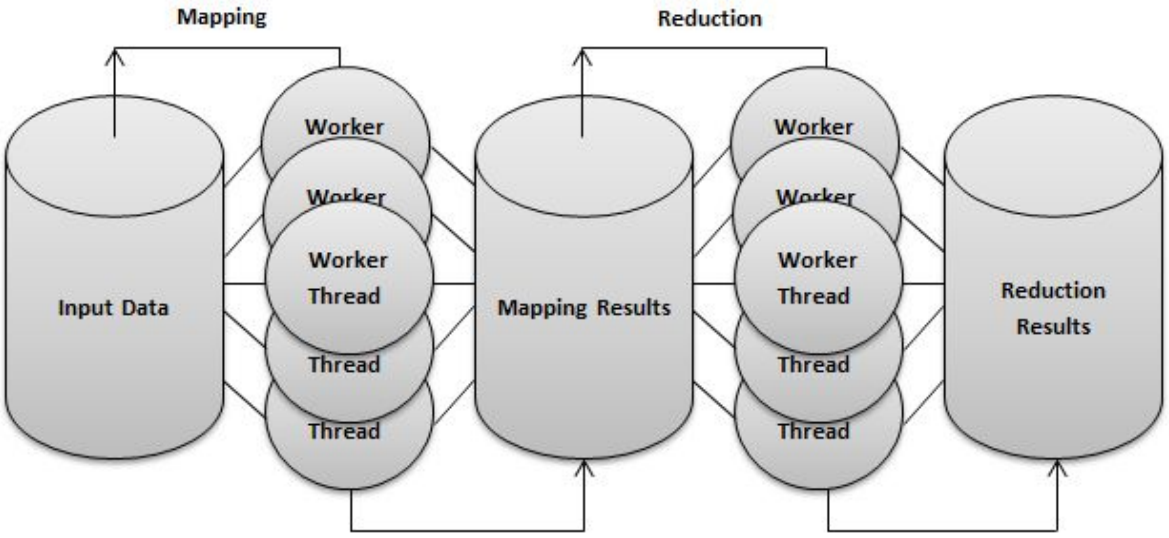
3. UML diagram

There is the WordCounter Class that we implemented for the second exercise. We can see that it has 3 characteristics: wordBad, wordChunks, and wordDictionary, each of them has the role to keep the data at the given moment. Our application that counts unique words from large amounts of input text, a mapping results Blocking Collection is created called "wordChunks". This particular Blocking Collection uses a ConcurrentBag as its base collection. Since words are added to and removed from the ConcurrentBag in no particular order, using a "bag" yields performance gains over a "stack" or "queue" which must internally keep track of processing order. WordDictionary is used to keep as keys the words and as values their frequency.



How we can see from the UML and from the next illustration, there are two key methods of this class and this paradigm, which are the mapWords, which takes the input and “split” it to threads, and mapReduce, which use other threads in order to

gather all the results and to put them together, composing our final result and resolving our problem.



4. Test cases

We tested our class with 3 different scenarios. The first one is the trivial case, when the input file is empty. So our dictionary, at the end of the execution, is empty. After, the second one is taking a simple input data which is just the same word repeated 4 times, so at the end of the execution, we find in the dictionary the respective word as a key and the value 4, the number of occurrences. The last test is more complex, taking as input the Lorem Ipsum text and counting each different word. We know that this test has 194 different words, and as we can see, the number of pairs in the dictionary is the same. All 3 test methods are passing.

```
[TestClass]
0 references
public class UnitTest
{
    [TestMethod]
    | 0 references
    public void mapReduce_wordCat4Times_expectedCorrectAns()
    {
        string text = System.IO.File.ReadAllText(@"F:\Facultate\Anul 2\ESILV\Disign pattern and soft development\Project DP\ConsoleApp1\testex2\testtext.txt");
        WordCounter reducer = new WordCounter();
        reducer.mapReduce(text);
        Assert.AreEqual(4, reducer.wordDictionary["cat"]);
    }
    [TestMethod]
    | 0 references
    public void mapReduce_noWord_working()
    {
        string text = System.IO.File.ReadAllText(@"F:\Facultate\Anul 2\ESILV\Disign pattern and soft development\Project DP\ConsoleApp1\testex2\empty.txt");
        WordCounter reducer = new WordCounter();
        reducer.mapReduce(text);
        Assert.AreEqual(true, reducer.wordDictionary.IsEmpty);
    }
    [TestMethod]
    | 0 references
    public void mapReduce_bigTextFile_bigOutput194Words()
    {
        string text = System.IO.File.ReadAllText(@"F:\Facultate\Anul 2\ESILV\Disign pattern and soft development\Project DP\ConsoleApp1\ConsoleApp1\message.txt");
        WordCounter reducer = new WordCounter();
        reducer.mapReduce(text);
        Assert.AreEqual(194, reducer.wordDictionary.Count);
    }
}
```

Exercise 3 – A Monopoly Game– Design patterns

1. Introduction

We have to simulate a simplified version of the Monopoly game where basic rules are defined. We are free to add rules and we have to use design patterns to implement our solution. The game is played turn by turn, one player at a time.

2. Desing hypotheses

We chose to use the State Pattern because a player can have two states :

- in jail
- out of jail

So, we implemented the State interface that has the playerTurn() method, and the two concrete state classes: InJailState and OutOfJailState that implements the State interface.

The Player class has a State attribute that is either InJailState or OutOfJailState. We also initialized this attribute as an OutOfJailState since the players are out of jail when they start the game.

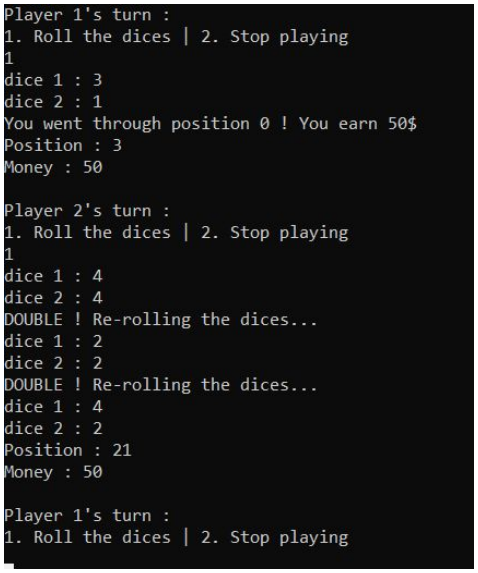
We also add the “money” attribute to the player: he will receive 50$ when he goes through position 0, 200$ if he stops at this position and he will lose 100$ when go to jail.

The turn of the player will be different since he is in jail or not :

- if he’s in jail: he has three tries to do a double and he does not succeed, he goes out of jail at the end of the three tries.
- if he’s not: he rolls the dices and changes his position. He can play again when he does a double. He can also earn money or lose money following the cases. He can go to jail too.

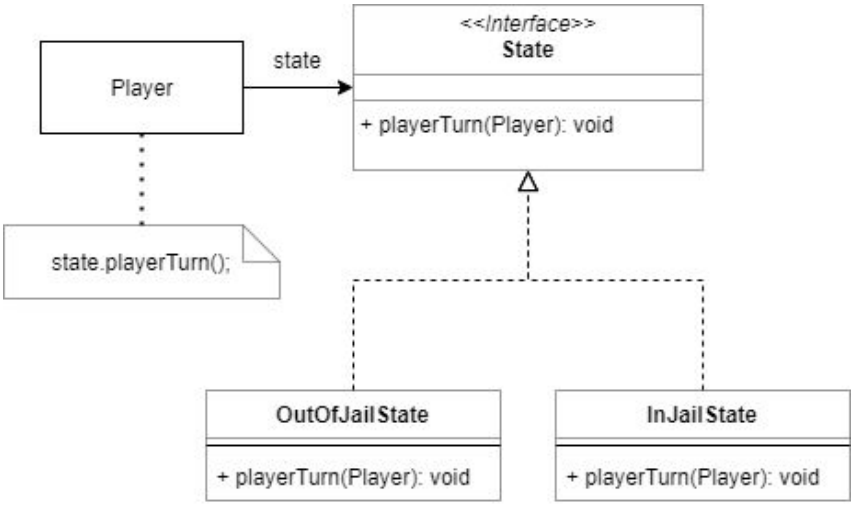
We implemented a little menu to make the user interface a bit simpler: he just has to type 1 to continue playing and roll the dices, or 2 to stop playing since the game has no end.

Here is a screenshot of a part of a game :

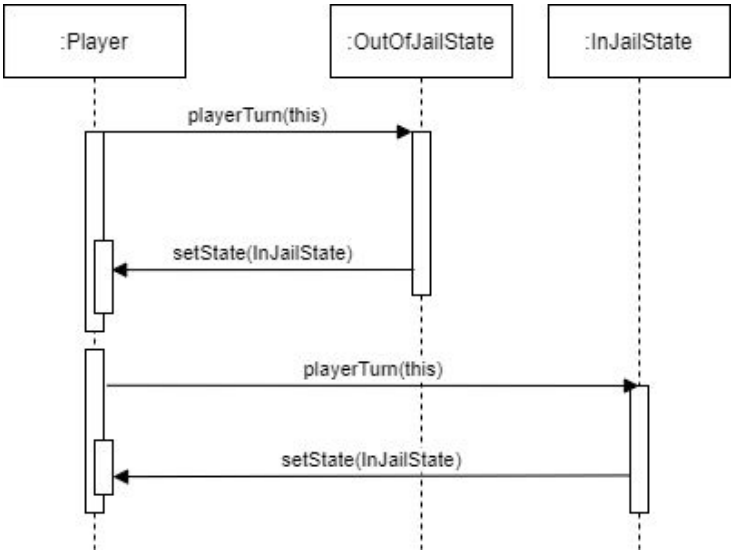


3. UML diagram

a. Class Diagram



b. Sequence Diagram



4. Test cases

For each test, we create a new Player which state will be OutOfJailState by default, then we play the turn of the player.

For the first test, we check if he moves, because if he's out of jail he should move. We also check that his position is between 2 and 12 because the result of 2 dices must be between 2 and 12.

For the second test, we set the player's state to InJailState and play his turn, we check that if he misses the double, he doesn't move, and if he succeeds, he moves.

The two tests are passing.

```
[TestMethod]
0 références
public void Player_OutOfJail_Moves()
{
    Player player = new Player("Player");

    player.playerTurn();
    bool ok = false;
    if(player.Position <= 12 && player.Position >= 2)
    {
        ok = true;
    }

    Assert.AreEqual(true, ok);
}

[TestMethod]
0 références
public void Player_InJailAndMiss_DoesntMove()
{
    Player player = new Player("Player");

    player.setState(new InJailState());
    player.playerTurn();
    bool ok = false;
    if (player.NbRollsInARow == 1 && player.Position == 0)
    {
        ok = true;
    }
    else if (player.NbRollsInARow == 0 && player.Position != 0)
    {
        ok = true;
    }

    Assert.AreEqual(true, ok);
}
```