Design Pattern and Software Development Process Report

MUFFAT-JOLY Clément
VAREZ Arthur
DIA4
10 January 2021
https://github.com/ClementMJ/DPSDP

Exercise 1 :

Generic

Custom

Queue

Introduction:

In this exercise we want to create our own Generic Custom Queue. This queue is composed of generic nodes which contain the data and are linked together. We want also to be able to iterate on this custom queue.

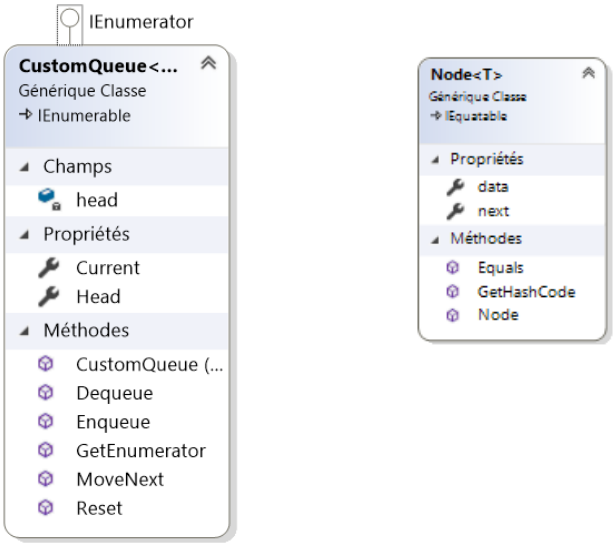
Design hypothesis:

FIFO policy is the structure we have to follow, new nodes are added at the end of the queue, and when we want to pop a node, it's the first one inserted that will be returned. The custom is composed of a head node which can be linked to an other one and so on.

We've also defined MoveNext(),Ie method to iterate on a CustomQueue.

Diagrams:

UML:



Sequence:

Exercise 2:

Map

Reduce

Introduction:

Here we want to simulate a map reduce in order to divide tasks and to decrease time complexity for doing some tasks. We give an input to our program and it divid our input into smaller problems so that we can apply some methods on them.

Design hypothesis:

To resolve our problem we defined 3 methods to do a MapReduce. Here are the methods :

- Mapping: Mapping contains the split step which split the different words. Meanwhile it assigns to them a value count equal to 1. Here we use a concurrent bag because this data structure allow us to have same key, value pair more than once.

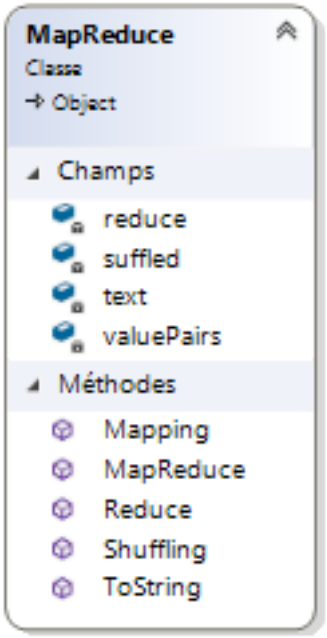
- Shuffling: for each key, value pair in our concurrent bag we access our key (the word) and we append a 1 to its list value.

- Reduce: Here we iterate on a concurrent dictionary. The aim of this method is to transform the list of 1 we got by the shuffling method into an int by counting the number of 1 in the list which correspond to the occurrence of the word.

- ToString: Finally this method displays the occurrences of the word in the text.

Diagrams:

UML:



Sequence:

Exercise 3:

A Monopoly Game

Introduction:

In this exercise we have to create a simple version of the famous game Monopoly. A player has to move on the board composed of 40 boxes. At each turn a player has to roll two dice, the number he got allows him to move the same numbers of boxes. If during a turn a player get three time the same number with the two dice, he goes to jail. When a player is in jail he has to wait 3 turn or to get a double number with the dice. In our version a player can't buy houses or draw cards like we usually did in a normal Monopoly game.

Design hypothesis:

Firstly to simulate a Game we had to define the board on which players can play. We chose to represented it as a list composed of 40 nodes with indexes from 0 to 39 named Square working like a circular list.

For the players, we defined them as a list of object Player. In the beginning a Player is initialised by his name and the board he plays on. It's state is FreeState(we will discuss later) and his index on the board is 0. Thank's to this class we can define the collection of players we'll use for a game.

Last class is Game which simulates the game process by creating a board, a list of players (defined above) by giving a number of rounds.

We also need the Die class to generate the number of squares a player can cross each time he roll the dice.

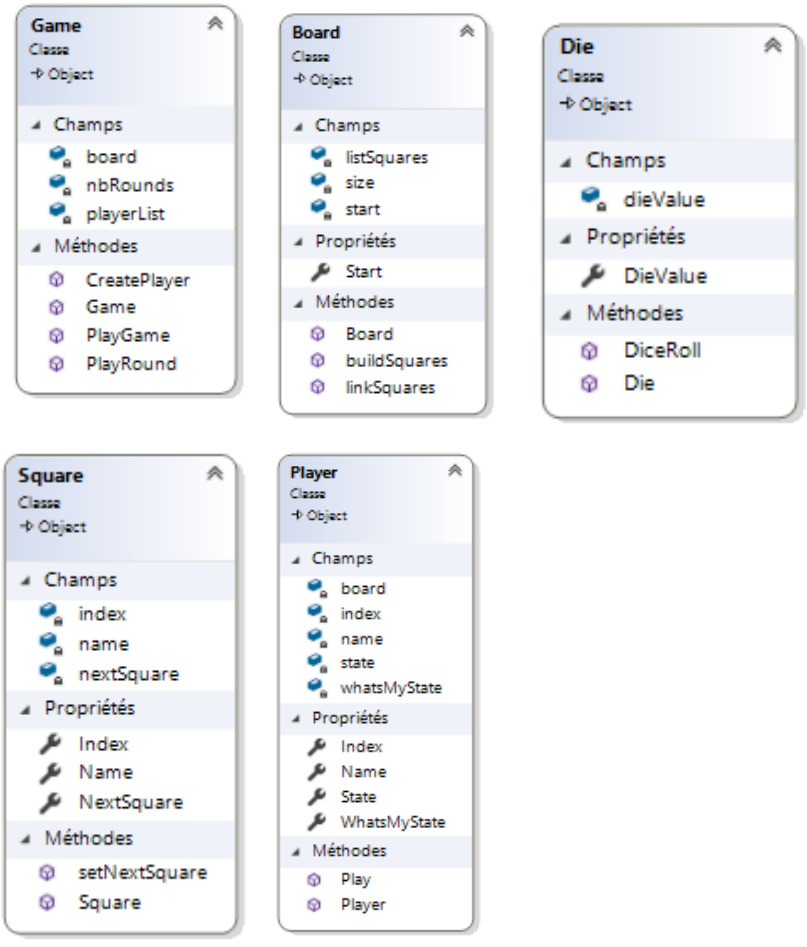
Once we've defined our main classes we can discuss about design patterns we used :

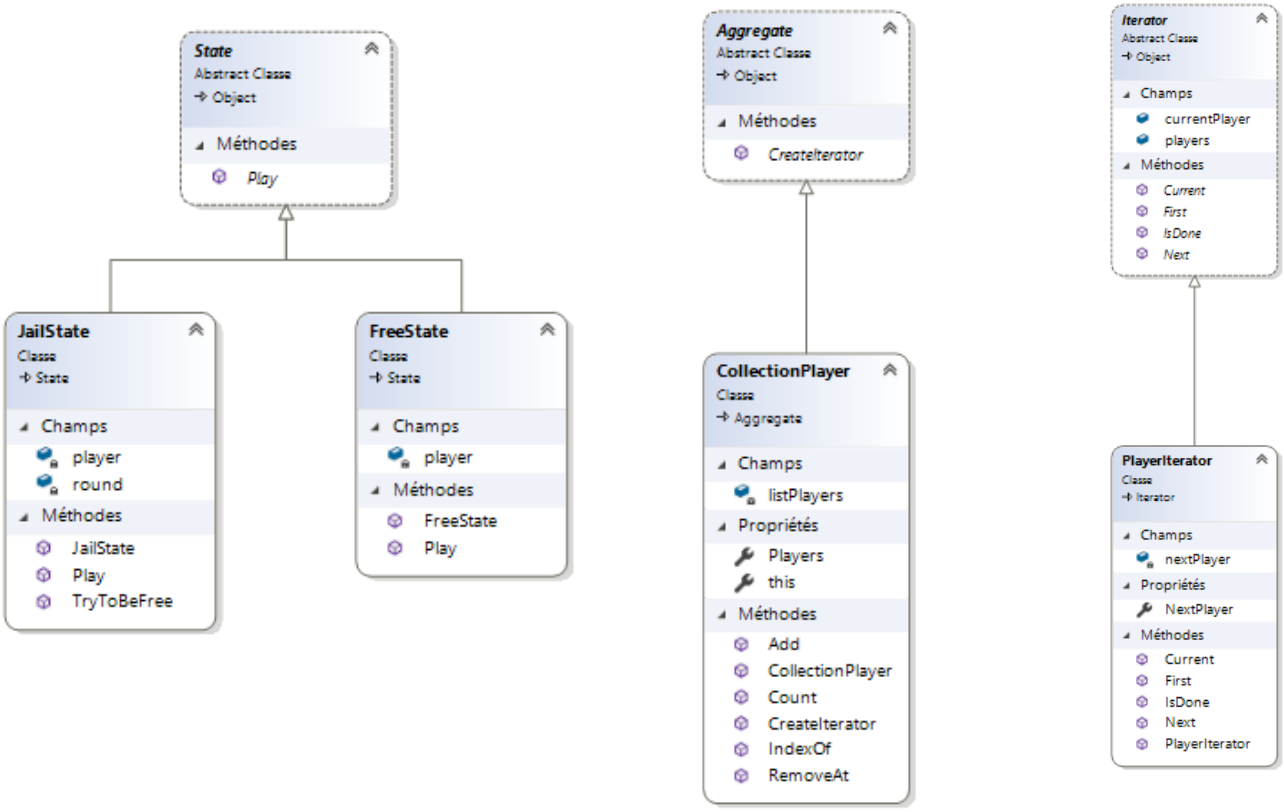
- State patterns : JailState, FreeState and State. This patterns allow us to modify the state of a player and send him to jail or let him free from jail.

- Iterator patterns : Aggregate, PlayerIterator, Iterator, CollectionPlayer. This patterns are designed to let players move on the board or to iterate on our player collection.

Diagrams:

UML:




State
Abstract Classe
Object
Méthodes
Play
Aggregate
Abstract Classe
Object
Méthodes
CreateIterator
Iterator
Abstract Classe
Object
Champs
currentPlayer
players
Méthodes
Current
First
IsDone
Next
JailState
Classe
State
Champs
player
round
Méthodes
JailState
Play
TryToBeFree
FreeState
Classe
State
Champs
player
Méthodes
FreeState
Play
CollectionPlayer
Classe
Aggregate
Champs
listPlayers
Propriétés
Players
this
Méthodes
Add
CollectionPlayer
Count
CreateIterator
IndexOf
RemoveAt
PlayerIterator
Classe
Iterator
Champs
nextPlayer
Propriétés
NextPlayer
Méthodes
Current
First
IsDone
Next
PlayerIterator

Sequence: