Design Patterns and Software Development Process – Final Project – A Monopoly™ game

**Github link : https://github.com/PaulCanalLDV/DP-SDP-Monopoly-game**

1. Introduction

*Objective of the project is to simulate a simplified version of the Monopoly™ game.*

The version of the Monopoly game that we have to program for this project is a simplified one. The rules of the game are the same of the basic game, however there is **no money** in this one. This means players cannot buy properties, and to leave the prison, they need to wait three rounds or roll doubles.

The board game is composed of 40 positions, like in the original game, players can occupy the same position and there are two dice to be rolled. Each player, one by one, rolls dice, moves forward according to the number from the dice (or not with the particularity of the jail) and the next player can then begin their turn.

The project is programmed in C#, as it is a good language for OOP.

1. Design Hypotheses

To program the Monopoly, we first have to define some design hypotheses.

In order to know where the players are on the board game, we need to keep their positions. We decided to link a player to their corresponding position and to keep the list of all players on a position. For the specificity of the jail, we have a list of players in Visit Only and a list for In Jail.

For the players in jail (not in Visit Only), we keep track of the number of turns after moving to jail, to know when they will have the right to leave.

At the beginning of the game, the list of the players is asked and the order for them to play. After each player turn, we go for the next player.

We saw that jail and GoToJail case were cases that appear only one time, so we decided to create a singleton pattern on these to be sure that only one instance could be created.

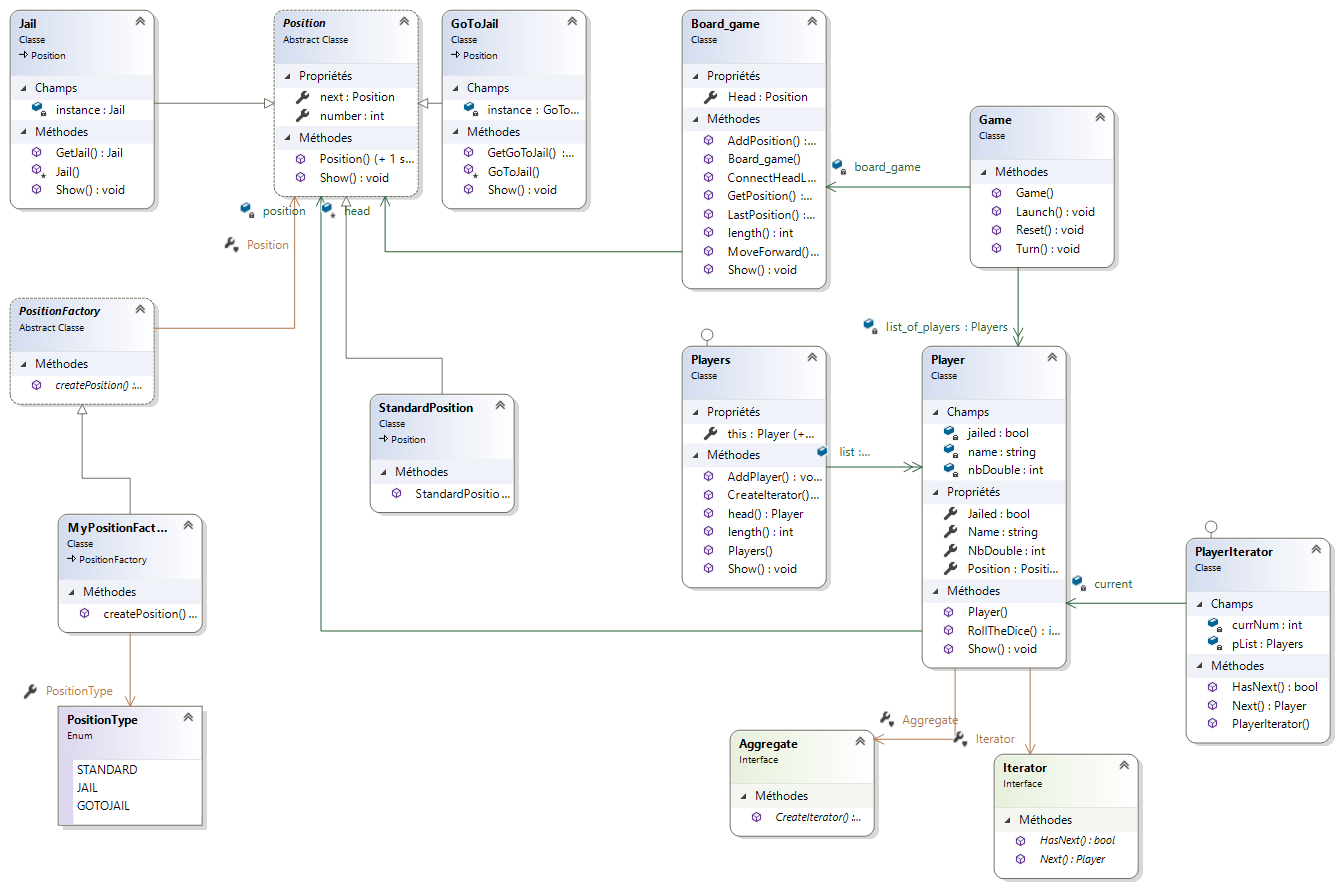
Also, players have to RollTheDice() one by one, so we decided to implement an iterator pattern on a list of player. They will take their turn from the first player to the last with this really useful pattern.

Factory les cases ont une certaine valeur de base, on crée une position -> on la modifie en fonction de ce qu’on veut/standard et gotojail/jail

We also used a factory pattern : each case has a certain value (the position) and then we modify it if it’s not a standard case like “gotojail()” and jail case.

These patterns were kinda useful in our project, it adds security and global organization.

1. UML Diagrams
2. Class Diagram of the solution

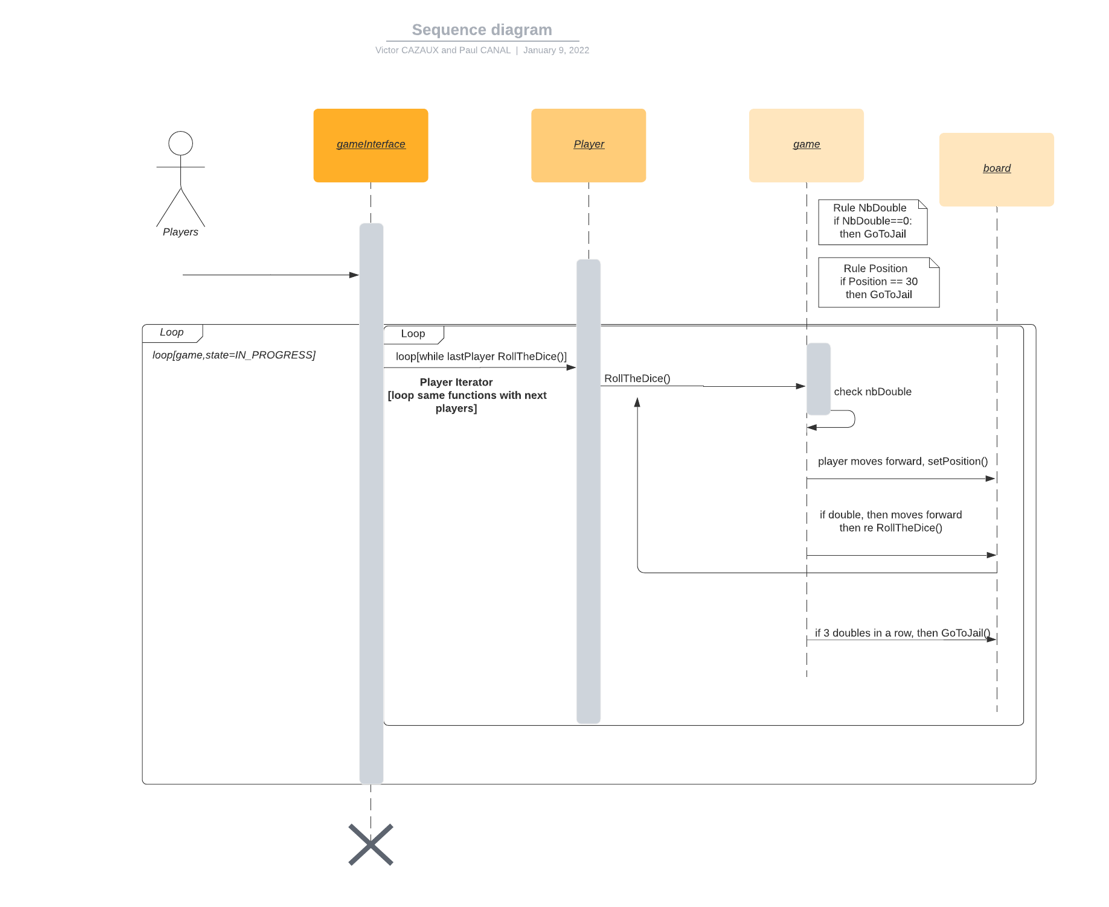


(we added the .png to the zipfile, to have the full quality)

Our class diagram describes the structure of our system by showing the classes, their attributes, theirs methods and their relationships.

This is a great tool in software development process, it really helps to organize everything, and at the same time, it offers a quick overview of the synergy happening among the different system elements.

1. Sequence diagrams :

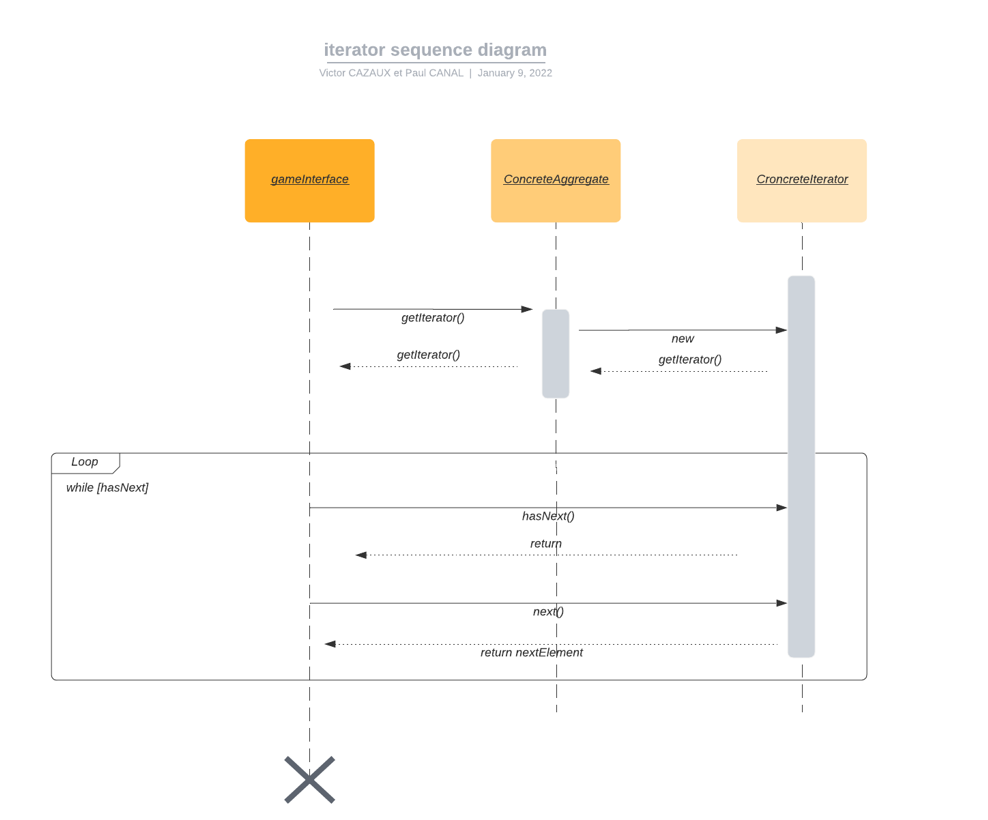


This sequence diagram shows how the game is running. We managed to use an iterator pattern to select our different players, then they play their turns by rolling the dice.

We have 2 loops as well, the first one where the condition is that the game must be in progress to trigger the second one, where the condition is that all players must play.

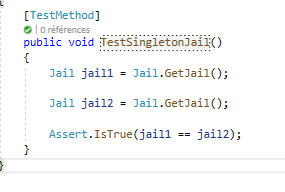
Sequence diagrams are also very usefuls in the development process because it modelizes the logic of our procedure. We can see how objects and components interact with each other. Monopoly game is quite simple, but it can really help in really complex system.

We made another sequence diagram, this one is focused on the iterator pattern.



This one quickly explains how the iterator pattern works. We can see that there are interactions between the concreteAggregate and croncreteIterator objects. This one is also useful, it shows really well the architecture of the iterator pattern.

4) Use Cases



So we wanted to make sure our singleton pattern was working.

We instancied two different Jail objects and we checked if there were the same. It’s a success. In our small project, this is not really useful because we don’t create tons of objects. But we understood easily why this type of “security” pattern are useful for bigger projects.