**Design Patterns and Software Development Process**   
  
  
  
Final Project – A Monopoly™ game  
  
1. Introduction

The project is based on a simplified version of the Monopoly game.

The goal is to find a way to design the solution according to the game’s rules. The design must include all the game’s rules while keeping it as simple as possible to understand to be simple to use and for the solution not to become overly complicated and overengineered.

For the solution’s implementation, design patterns might or might not be relevant if it fits a game feature’s use case or not.  
  
2. Design Hypotheses

In order to simplify our code, we separated all elements of the game in different classes. That’s why we created classes Board, Dice, Player, Position.

Creating the Dice class wasn’t an obligation for us but it was the easiest way we found to manipulate these objects.

Each position is a case of the board. It contains a index and a type (jail or normal).

In our version, we implemented two players, two dices, one board and 40 positions.

In each version we can assume that there will be only one board. So we created a **Singleton** on this class. It checks that there is only one board created during the game.

Board is a **circular linked list** of positions. In order to create a circular linked list, we needed to implement the node class. With this type of list, it simplifies the situation where the player have to move from the last position to the first one.

We had difficulties to change the role of each player, to change when he was playing or not. That’s why we implemented a **state design pattern**. For this pattern, we had to create classes ConcreteStatePlay, ConcreteStatePause, State and Context. When the turn of a player is over, his state switches to pause.

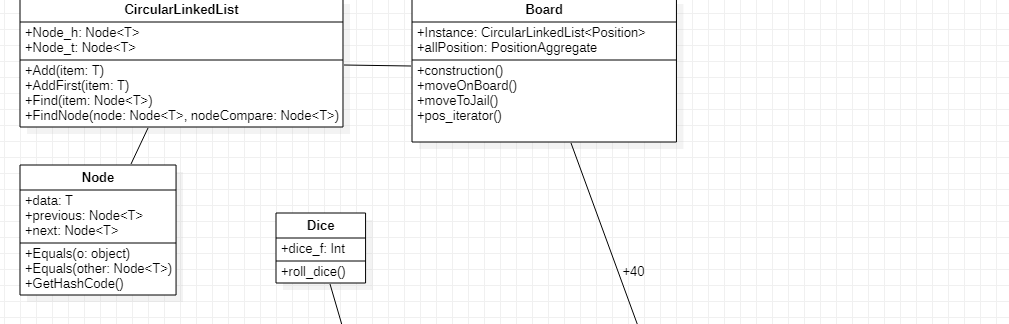
Moreover, we wanted to add the possibilty to check informations about the game. The easiest way to do it was to implement two **iterator patterns**. One with Player and the other one with Position. For this pattern we implemented classes Aggregate, Iterator, PlayerAggregate, PlayerIterator, PositionAggregate and PositionIterator.

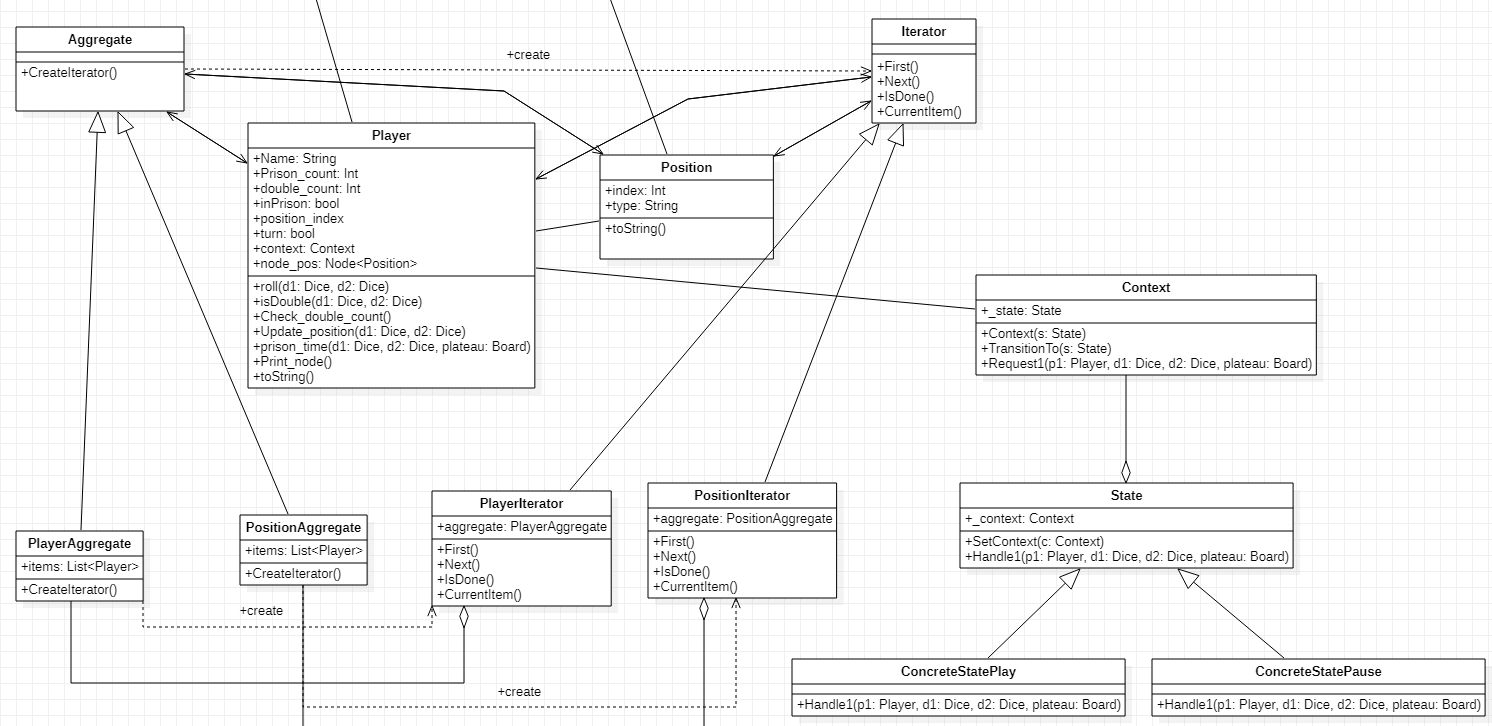
We also thought about creating an abstract class position and then add one class for each type of position (3 in our version) but it seems to be just a complication.

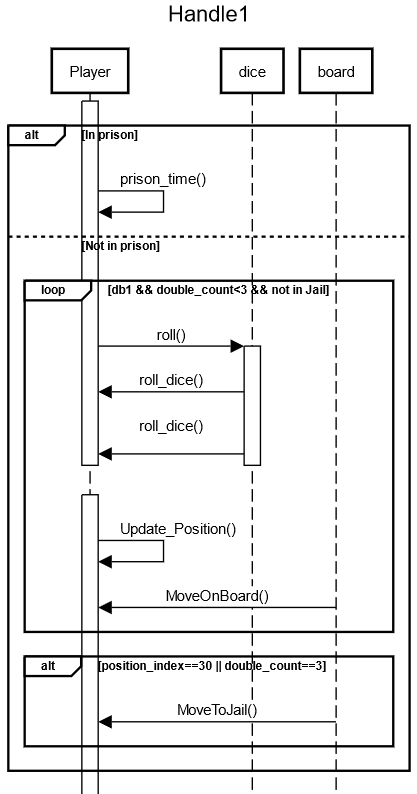
3. UML diagrams

a. Class diagram of the solution

You will find below the complete UML class diagram of our solution:

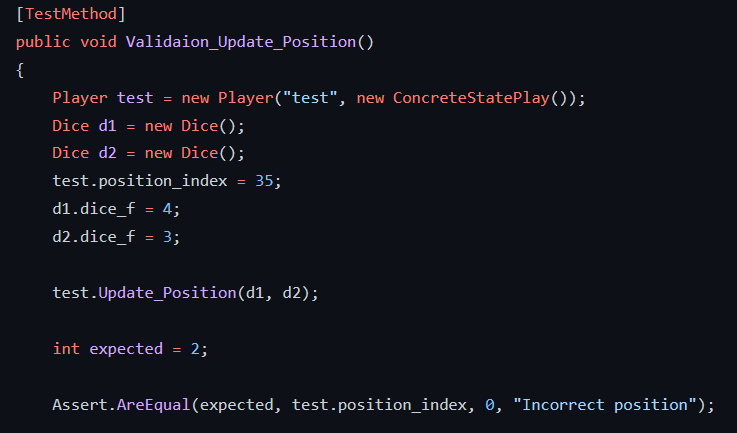


  
b. Sequence diagrams

You will find below the sequence diagram of the function that handles a player’s movement on the board.  
  
4. Test cases

We have made unit tests for two functions: Update\_Position() which update a player’s position on the board, and MoveToJail() which moves a player to the jail position.

For the first one, we take two dices number and a player’s position as input data. The output data is the expected player’s new position on the board.



For the second one, we just need to create an instance of Board and Player and move the latter to jail. The expected output is the jail position number.

