Final Project – A MonopolyTM game

Design Patterns and

Software Development Process

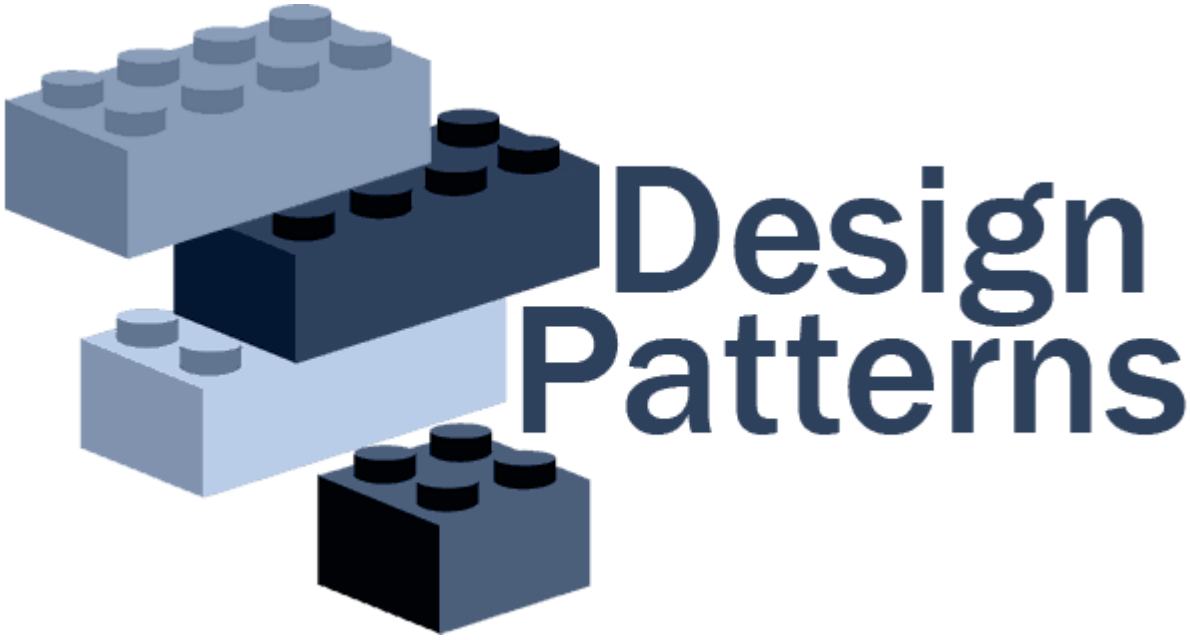


Table des matières

1. Introduction

As ESILV 4th year students, we attended the 'Design Pattern and Software Development Process' course. Along the course, we learnt theory and practice of many design patterns, processes and tools relevant to our studies. This project brings to a close this course, offering a final opportunity to use our newly acquired skills through a practical project: the goal was to simulate a simplified version of the Monopoly game.

The concept offers several possibilities for design patterns. After exploring possibilities, we went with the one that seemed most fitting to us as explained below: **Singleton, Strategy and State patterns**. We put into practice the good manners we were accustomed to regarding the modelling. You shall find all explanations in the corresponding sections.

As for the simplified version of the game, it consists of a similar board, having 40 possible positions that the players travel across during the game. The part where the positions represent famous avenues has been removed, then there is no buying or deals at all during the game. The only remaining game features are the jail, located at tile 10, and the go-to-jail tile at the 30th position. All the moving around also remains the same as in the original game (2 dices, moving by the sum of the dices, 1 double and the player plays again, 3 doubles make the player go to jail, 1 double to go out of jail or after 3 turns).

Let's move on to the design hypothesis.

2. Design Hypotheses

During this project, we have faced some difficulties regarding the choice of implementation of design patterns in the structure of the code, whose handling was not intuitive at first. We then had to rethink every action required to design the game intelligently.

Our project is grouped around 3 blocks, 3 specific design patterns, grouping the major challenges:

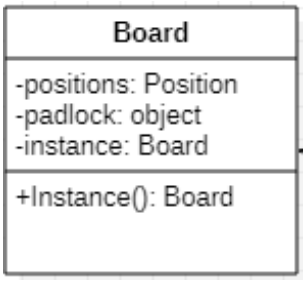
- the creation of a unique board
- the player's state (free or in jail)
- the positioning of the players on the board

We implemented 8 classes and 1 interface.

Singleton Pattern

To begin with, the first step was about the **creation of the board**. We wanted to ensure that the board could only be created once. In order to do so, we used a **Singleton Pattern**.

Our "Board" class implements a private constructor (or a sealed class), and a private field ("instance") which will be used to check if an instance has already been created. Moreover, we decided to make it "**Thread Safe**" inside our property Instance(), using a **lock** (padlock) to make sure that two threads won't create two boards.



Secondly, we had to think about the framework of our board and the organization of the positions. Each position is independent from their neighbours in our code. In order to know where the case is on the board, we created the attribute “positions”, a list of instances of the class Position, which allows us to have access to the attribute “caseNumber” of Position. Thus, on the board, position 0 will point to 1, then 1 to 2,… and finally 39 to 0.

To implement this board, we have hesitated to use the **Prototype Design Pattern**, to clone every position around the board. But we decided not to use it as it did not fit well in our final structure

Nonetheless, if we wanted to go further, we could have been able to implement other behavior patterns to get closer to the real monopoly, for example to attribute a price to each case.

Then, the **two major challenges** we faced during this exercise were the following:

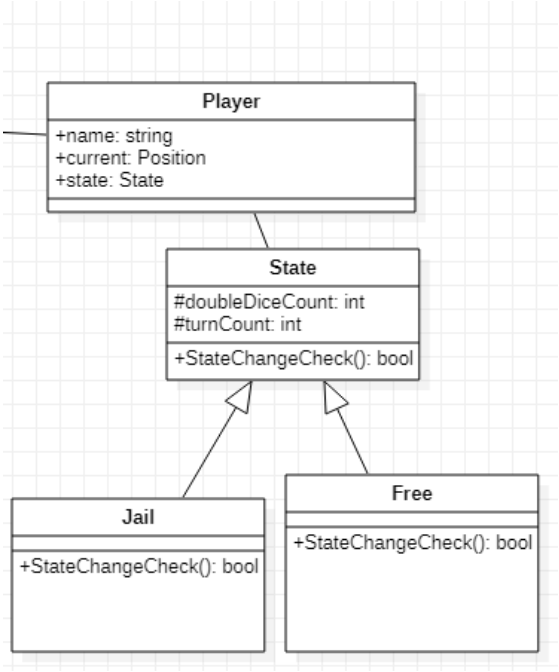
- How to link the influence of the state on the positions, knowing that the information was stocked in different classes?
- How to implement every position separately, knowing that some specific positions do not have the same effect, while they are instance of the same class?

We struggle with these questions for a while, but we realize that positions have two kinds of effect: when a player **stops** on the position, and when he **moves** from the position. These effects are what makes positions different, and therefore we decided to use the **Strategy Pattern** for the movement (move and stop) behavior, and a **State Pattern** in order to check the state of the player.

State Pattern

Our **State Pattern**, the **Check State Behaviour**, allows us to define elements depending on the state of the player through the Boolean method StateChangeCheck() contains inside the mother abstract class State. The State class also contains two protected attributes, doubleDiceCount, which counts the number of double dices made in a row and turnCount, which counts the number of turns played by each player.

To implement this pattern, we need to create two different states that will define the player: a **Free State** and a **Jail State**. Then the Check State Behaviour will depend on the state of the player: if he is free, then he moves normally, if he is in jail, he needs to do a double. Every double increments an integer (doubleDiceCount) and will check the state of the player, and change it if needed, every 3 doubles.



Strategy Pattern

Concerning the **Strategy Pattern**, the behaviors which can shift according to the positions are due to our **interface MovingBehaviour**. This interface implements two capital methods, **onStop(Player player)** and **onMove(Player player)** which are inherited by the **Position class, the abstract parent class** of the 3 classes defining the behaviors according to the case: **JailPosition, SimplePosition** and **GoToJailPosition.**

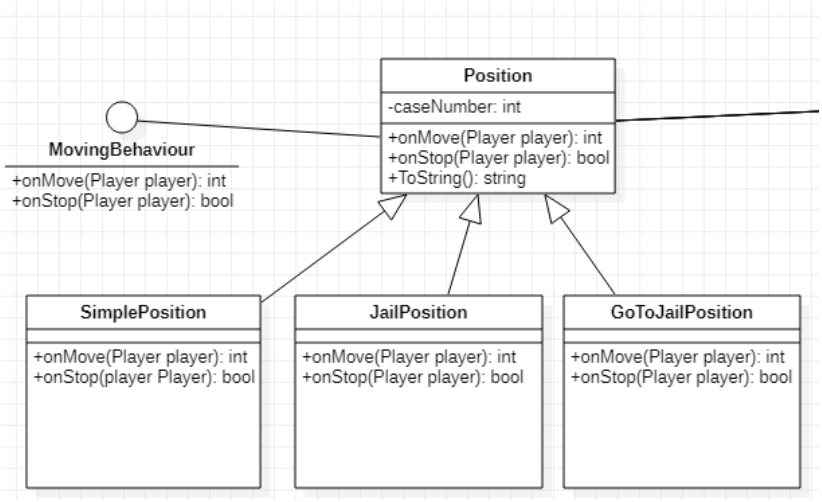
The onMove(Player player) and onStop(Player player) methods are thus **overriden** as needed in each child class. As these methods have the class Player passed into a parameter, we can **have access to our State Pattern** and especially the method StateChangeCheck() which **influences the Moving Behaviour**.

On the one hand, the onStop(Player player) is a boolean function which returns True if the player stays on the same position or False if he has to change his position. This is particularly the case if the player lands on the GoToJail (case 30), he goes to jail (case 10).

On the other hand, the onMove(Player player) is the method which returns the next position of the player according to his current position, after having him rolling the dices. For example, for a “Simple position”, it would only return the new position but for a “Go to jail” position, it will bring the player to Jail. This methods also deals with special cases linked to double dices.

This is particularly the case in the SimplePosition class where it is necessary to ensure that the player has not made more than 3 doubles in a row, or in the JailPosition class where the player must make a double to get out of prison. As our implementation framework allows us to have access to the State Check Behaviour, we could check after checking the number written on dices if the state of the player has change, in order to update the position of the player.

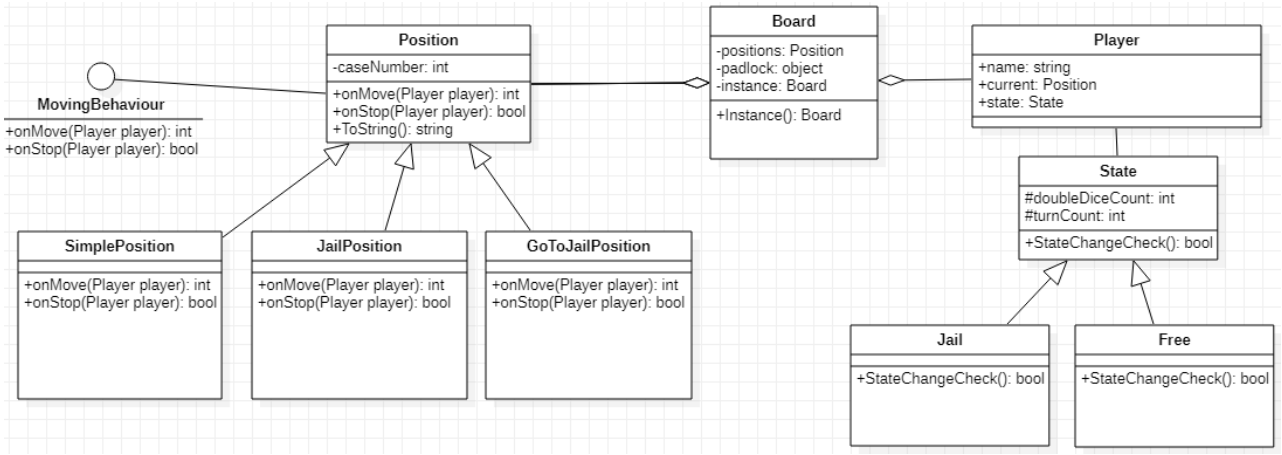
Using this, the "Position" class will delegate its StopBehavior and MoveBehavior instead of creating many classes and methods according to different behaviors.



Finally, concerning the global workflow of the project, each player rolls the dice depending on his turn, his state, moves or not, and finally stops on a position to trigger a behavior.

3. UML diagrams

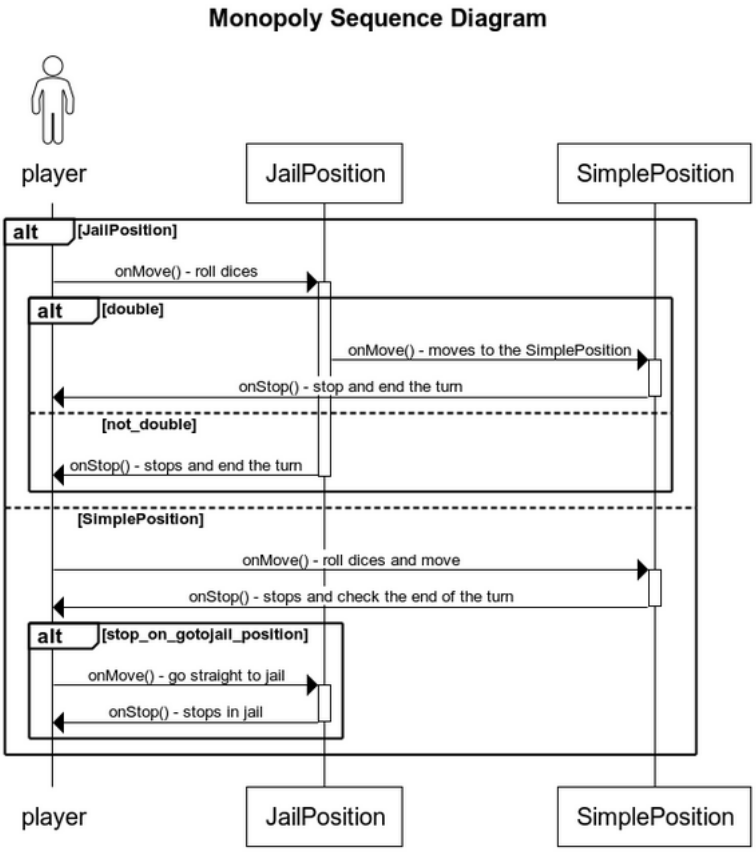
a. Class diagram of the solution



This class diagram was made on StarUML

b. Sequence diagrams

For the sequence diagram, we chose to represent what is happening during **the turn of a player**. A player either starts its turn on a JailPosition or a SimplePosition. We detailed a few cases depending on the value of the dice and the stopping position.



To make this diagram, we used the tool on this link : https://www.websequencediagrams.com/

We 'coded' the diagram, using 'alt' and 'else', arrows from one actor to another, and + and – for the vertical blocks. Here is the code snippet corresponding to the previous diagram:

```
title Monopoly Sequence Diagram

actor player

    alt JailPosition
        player ->+JailPosition : onMove() - roll dices

        alt double
            JailPosition ->+SimplePosition: onMove() - moves to the SimplePosition
            SimplePosition->- player:onStop() - stop and end the turn

        else not_double
            JailPosition ->-player: onStop() - stops and end the turn
        end

    else SimplePosition
        player->+ SimplePosition: onMove() - roll dices and move
        SimplePosition->-player : onStop() - stops and check the end of the turn

        alt stop_on_gotojail_position
            player->+ JailPosition:onMove() - go straight to jail
            JailPosition ->-player:onStop() - stops in jail
        end
    end
```

We could have done many different diagrams. For example, we could have represented both players, added a loop for the 50 turns, represented the State or the Board, or used the same actors but in a different scenario, etc.

4. Test cases

Once we had implemented all our patterns, we ran the program to see if the game was working correctly.

We then decided to test more precisely the 4 major functionalities of our project:

- The creation of a unique Board thanks to our Singleton pattern
- The “3 doubles dice in row” case leading to Jail case
- The “double dice” case that allows the player to get out of jail
- The case “Go to jail”

In order to do these tests, we created another **Unit Test Project** on Visual Studio.

Singleton Test case

In order to trigger the exception set up in the Singleton Pattern, we tried to create another instance of a board.

We obtained the expected result, only one board was created thanks to our Singleton Pattern protecting the uniqueness existence of a board.

Here is the code snippet:

```
[TestMethod]
0 références
public void TestMethod1() //test the singleton pattern
{
    Board board1 = Board.Instance();
    Assert.ThrowsException<Exception>(() => Board.Instance());
    //we try to create a second board, but we shoud get an exception as we can only have one board and set up a singleton pattern

}
```

Test case “Go to jail” after 3 doubles dices in a row

We created two different cases to test our State Pattern. The first one with a player having done 2 double dices to see if the Boolean method StateChangeCheck() was returning False, and the second one with the same player having done 3 double dices in a row, to check if the method was returning True.

We obtained the expected results for both tests, for the first one nothing happened and the state of player was still “Free” and for the second test, the player’s state changed from Free to Jail.

Here is the code snippet:

```
[TestMethod]
0 références
public void TestMethod2() //case go in jail after 3 doubles
{
    Player player1 = new Player("Player_test");
    player1.State.DoubleDiceCount = 2;
    Assert.IsFalse(player1.State.StateChangeCheck());
    //We test if the player stays free if only has done 2 doubles, meaning StateChangeCheck should return false

    player1.State.DoubleDiceCount = 3;
    Assert.IsTrue(player1.State.StateChangeCheck());
}
```

Test case “Go to jail” case

In the game, if a player stops on the position 30, which is the Go-to-jail position, he does not stay here until the next turn like other basic positions. Instead, the player has to move immediately to jail, and end its turn there even if he did a double. In order to test if that event was going as expected, we created a player and an instance of this position. We made the player stop on that instance by setting it as its current position, then called the onStop() method. This method, implemented on all types of positions, is the one designed to return true if the player’s turn is over or false if not. We indeed expected it to return false as a player can not stop on the go-to-jail case, hence tested it using Assert.IsFalse.

The test was successful. Here is the code snippet:

```
[TestMethod]
0 références
public void TestMethod3() //case go to jail
{
    Player player1 = new Player("Player_test");
    player1.current=new GoToJailPosition(1);
    Assert.IsFalse(player1.Current.onStop(player1));
    //We test if the player can stay at this position when he arrives on this position. It shoud return false meaning the game lauches the onMOve function immediatly which sends him to jail.

}
```

Test case “Go out of jail” after a double dice

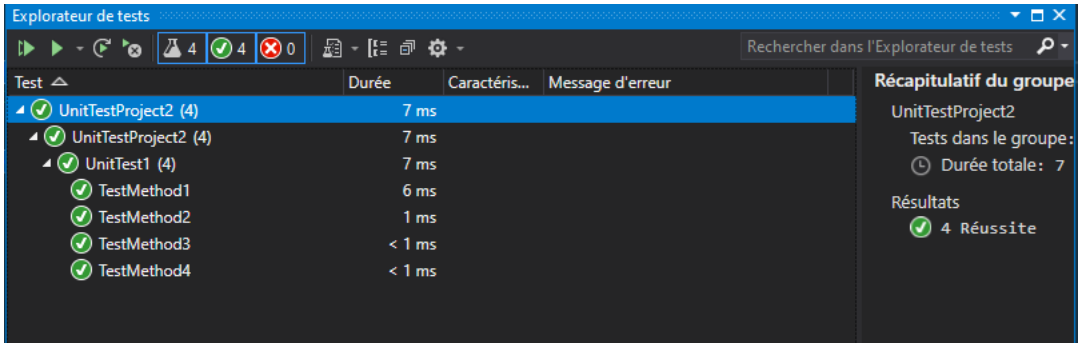
In Monopoly, players might end up in ‘jail’. There are two ways to get out: either make a double on the player’s turn, or wait 3 turns. We wanted to test that the first exit scenario was well implemented. To test that, we created a player and immediately set is State as Jail. We set the variable DoubleDiceCount to 1, imitating the case where the player just made a double. Then, we called the StateChangeCheck() method, designed to return true if the player needs to change state (jail to free or free to jail). In our case, the player is supposed to leave jail, so is state should go from Jail to Free, and the function needs to return true. That’s what we test using the Assert.IsTrue function.

The test was successful. Here is the code snippet:

```
[TestMethod]
0 références
public void TestMethod4() //case go out of jail after a double
{
    Player player1 = new Player("Player_test");
    player1.State=new Jail();
    player1.State.DoubleDiceCount = 1;
    Assert.IsTrue(player1.State.StateChangeCheck());
    //We test if the player that is in jail and did a double will be free, meaning StateChangeCheck should return true
}
```

Test cases overview

We then tested each Unit Test and as we can see below on the picture, we obtained the expected results as they were all successful.



5. Conclusion/ Final remarks

To put it in a nutshell, the project allowed to us to apply what we had seen along the course. We successfully **implemented 3 design patterns**: **Singleton, State, and Strategy patterns**. Our project meets all requirements and works well, allowing us to simulate a game of the simplified Monopoly without interfering.

The program prints some information along the game, like the value of the dices, players' positions after the turn:

```
Turn 2
Audrey's dices gave 6 and 2.
The player is on the position 21 of the board.

Lisa's dices gave 1 and 2.
The player is on the position 17 of the board.
```

The game stops on its own after 50 turns.

```
Turn 7
Audrey's dices gave 1 and 1.
Did a double. He can play again!
The player is on the position 14 of the board.
New turn for the player.
Audrey's dices gave 6 and 6.
Did a double. He can play again!
The player is on the position 26 of the board.
New turn for the player.
Audrey's dices gave 2 and 2.
Three doubles in a row! goes to jail!
The player remains in jail for the 1th turn.
```

On the example here, Audrey did 3 doubles in a row and went in jail.

```
Turn 10
Audrey's dices gave 3 and 2.
Goes out of jail after 3 turns
The player is on the position 15 of the board.
```

3 turns later, she didn't make a double but is going out anyway.

```
Turn 18
Audrey's dices gave 2 and 1.
Audrey is on case 30, this is the go to jail position.
He was free and now is in jail
```

Later on, Audrey stopped on the ‘go to jail’ position and went straight to jail.

```
Audrey's dices gave 1 and 1.
Goes out of jail after making a double
The player is on the position 12 of the board.
```

Luckily, she did a double and could leave jail quite fast!

On an improvement perspective, we could have added features to our project to make it more enjoyable: chose the number and names of players with the console, add a visual of the board throughout the game, etc.

We are quite satisfied by our project, both in terms of final results and the technical experience and improvement. We had a great time working as a team to make this project become real within a few weeks to meet the deadline and are pleased to submit our work.