Design Patterns and Software Development Process

Exercise 1 – Monopoly by PUJOL Corentin & PELET Quentin

GitHub link: ***https://github.com/corentin-pujol/Monopoly***

1. Introduction

The objective of the project is to simulate a simplified version of the Monopoly™ game. Indeed, the players will only have to make turns on the board without the system of buying properties and monopolies that can be found in the complete version.

The only constraints will be the "Jail" square, the "Go to jail" square, replay if you make a double and finally players will also go to jail when they make three doubles in a row. In addition, if a player is in jail and makes a double, then he/she goes out, moves on and his/her turn ends.

2. Design Hypotheses

Monopoly™ is a game played by 2 to 8 players, on a board with two dices and pieces moving from square to square in a clockwise direction. So, to design our game, we decided to create the following set of objects:

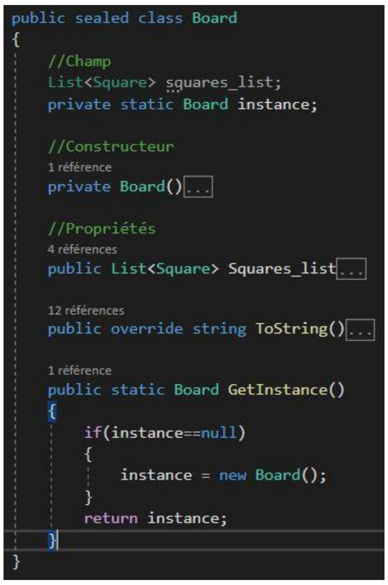
- The dices
- The board
- The game squares
- The players
- The players' pieces
- The game

a) Singleton pattern

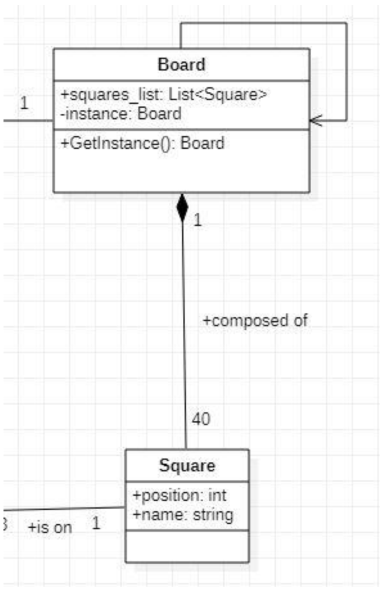
Let's recall the definition of a Singleton pattern: it is a design pattern that guarantees that the instance of a class exists in only one copy, while providing a global access point to this instance.

However, the game is played on a single board with two dices, and these are unique. It's therefore wise to use the Singleton pattern when designing these objects to ensure that they are unique during a game of Monopoly™.

Board design and development:



```
public sealed class Board
{
    //Champ
    List<Square> squares_list;
    private static Board instance;

    //Constructeur
    1 référence
    private Board()...

    //Propriétés
    4 références
    public List<Square> Squares_list...

    12 références
    public override string ToString()...

    1 référence
    public static Board GetInstance()
    {
        if(instance==null)
        {
            instance = new Board();
        }
        return instance;
    }
}
```

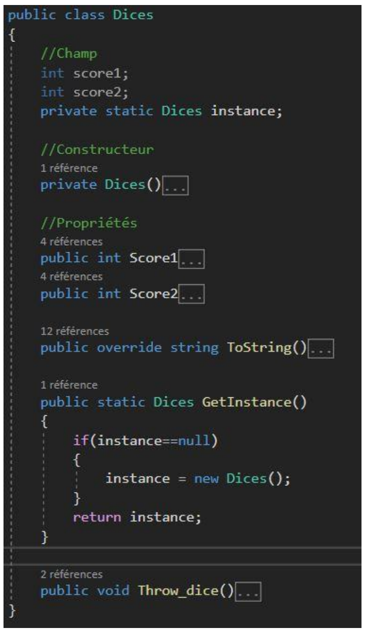


The board is made up of the list of Monopoly™ squares, as well as an instance of the Board class, which allows us to check in the GetInstance function whether the board object has already been created or not.

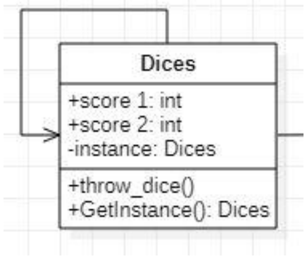
If it has been created, we return the previously created board, else we create a new one. Its character is therefore unique in our application.

Dices design and development:

Likewise for Monopoly dices, there is only one pair in a game of Monopoly™:



```
public class Dices
{
    //Champ
    int score1;
    int score2;
    private static Dices instance;

    //Constructeur
    1 référence
    private Dices()...

    //Propriétés
    4 références
    public int Score1...
    4 références
    public int Score2...

    12 références
    public override string ToString()...

    1 référence
    public static Dices GetInstance()
    {
        if(instance==null)
        {
            instance = new Dices();
        }
        return instance;
    }

    2 références
    public void Throw_dice()...
}
```



In the same way, the dices are instantiated once and only once with the same reasoning.

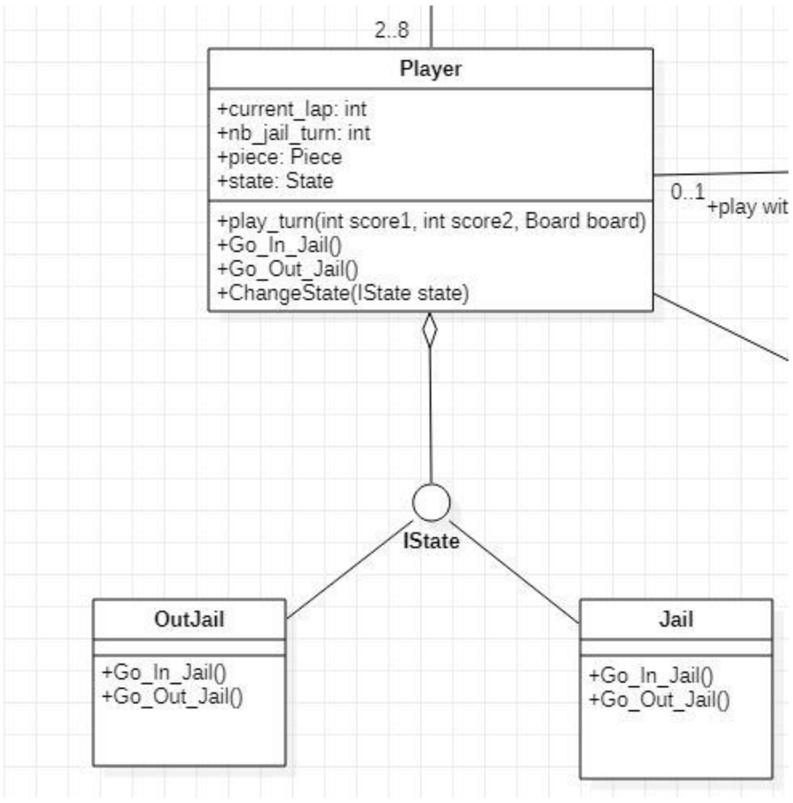
b) State pattern

In this version of Monopoly, players move from square to square and change state only when they fall on the "Go to jail" square, or when they make three doubles in a row. In order to model the changes in player states, we have chosen to use the State pattern to simplify and make the management of the different states more efficient and simpler.

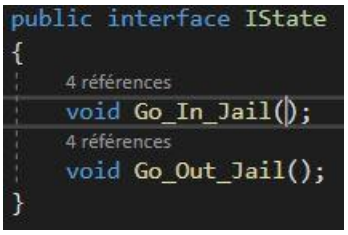
The State pattern proposes to create new classes for all possible states of an object and to extract the state-related behaviors in these classes. This allows the behavior of an object to be modified when its internal state changes.

Rather than implementing all the behaviors of itself, the original object stores a reference to one of the state objects that represents its current state. It delegates all state manipulation to this object.

So, this is how we wanted to model and develop the Player class:



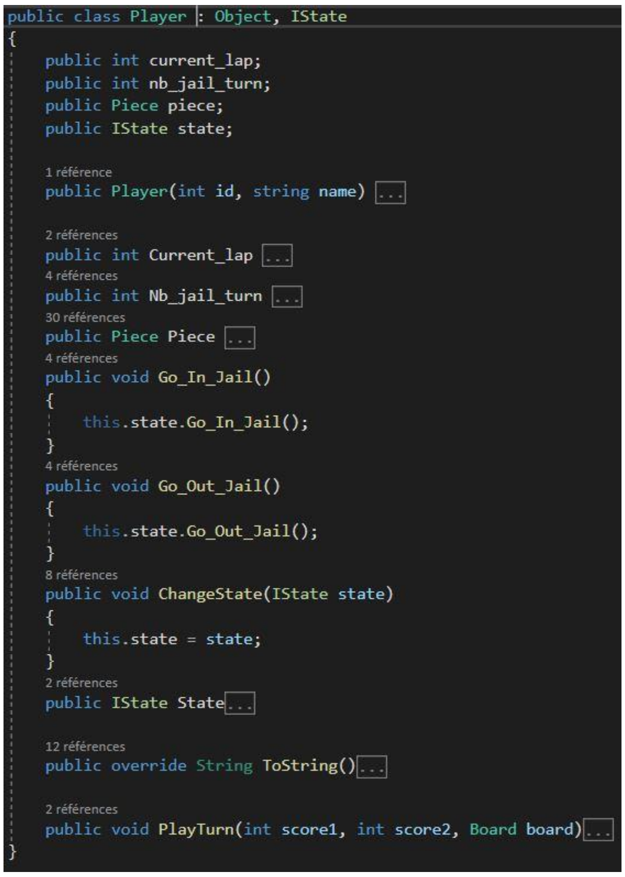
We have created a State interface, which allows us to define the functions that handle state changes:



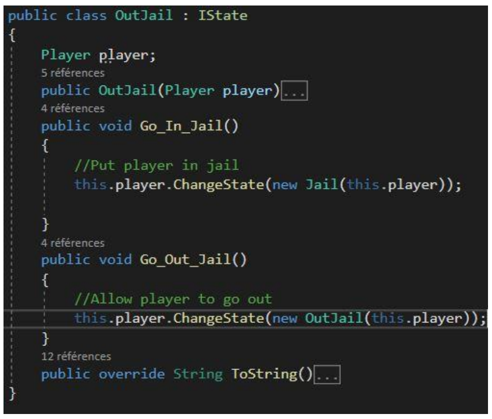
```
public interface IState
{
    4 références
    void Go_In_Jail();
    4 références
    void Go_Out_Jail();
}
```

Here, we only have the go and get out of jail options, but if we had wanted to implement a Monopoly with all these rules, we could have put in the other states such as: On start box, On property box, On community or lucky box, On taxes box, etc...

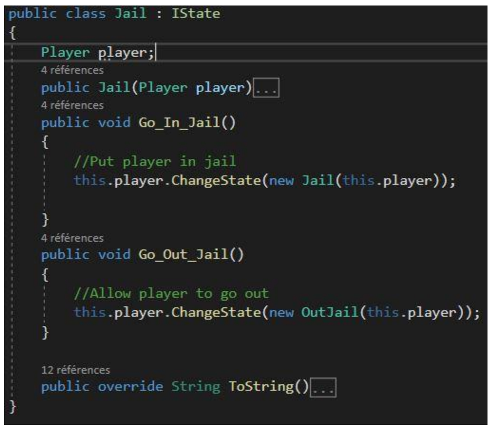
This interface is implemented in the player class which stores its current state, as well as in the state classes "Jail" which defines the state of a player in jail and "OutJail" when the player is not in jail.



```
public class Player : Object, IState
{
    public int current_lap;
    public int nb_jail_turn;
    public Piece piece;
    public IState state;

    1 référence
    public Player(int id, string name) ...

    2 références
    public int Current_lap ...
    4 références
    public int Nb_jail_turn ...
    30 références
    public Piece Piece ...
    4 références
    public void Go_In_Jail()
    {
        this.state.Go_In_Jail();
    }
    4 références
    public void Go_Out_Jail()
    {
        this.state.Go_Out_Jail();
    }
    8 références
    public void ChangeState(IState state)
    {
        this.state = state;
    }
    2 références
    public IState State...

    12 références
    public override String ToString()...

    2 références
    public void PlayTurn(int score1, int score2, Board board)...
}
```



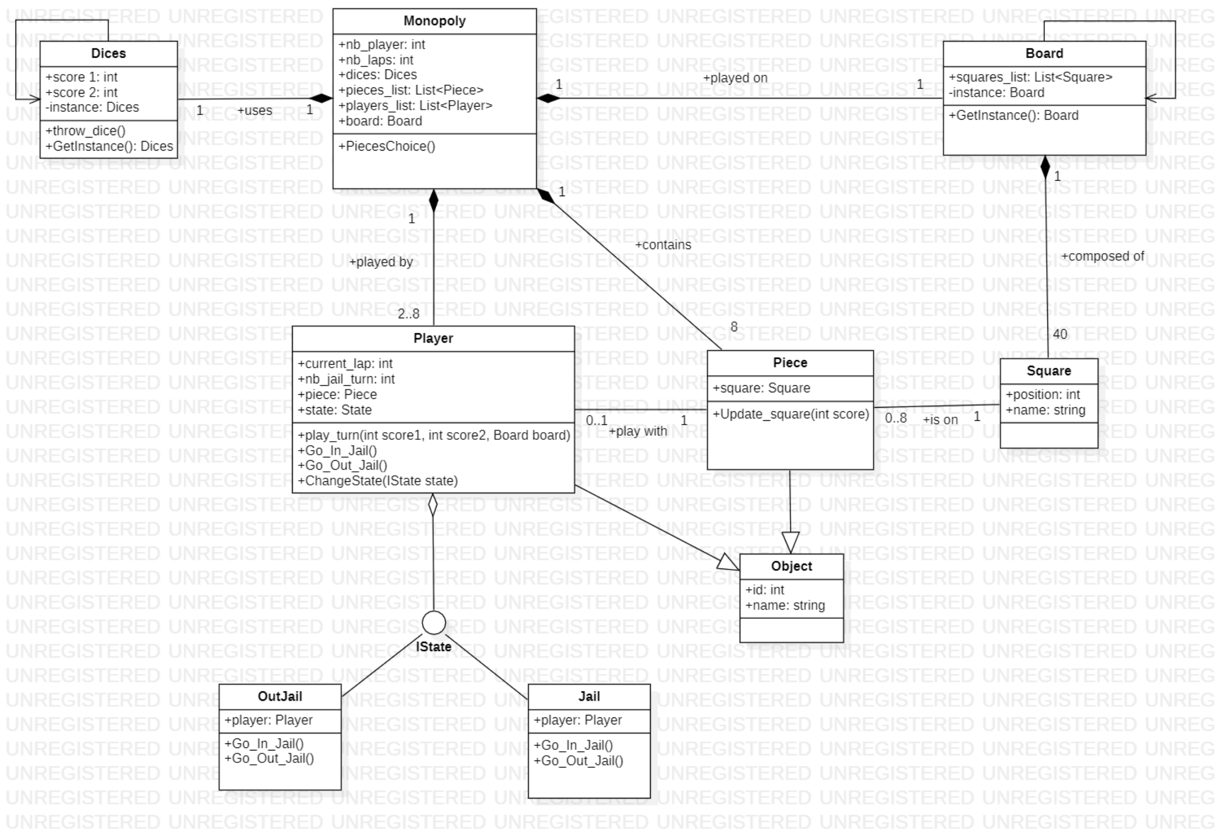
```
public class OutJail : IState
{
    Player player;
    5 références
    public OutJail(Player player)...
    4 références
    public void Go_In_Jail()
    {
        //Put player in jail
        this.player.ChangeState(new Jail(this.player));

    }
    4 références
    public void Go_Out_Jail()
    {
        //Allow player to go out
        this.player.ChangeState(new OutJail(this.player));
    }
    12 références
    public override String ToString()...
}
```



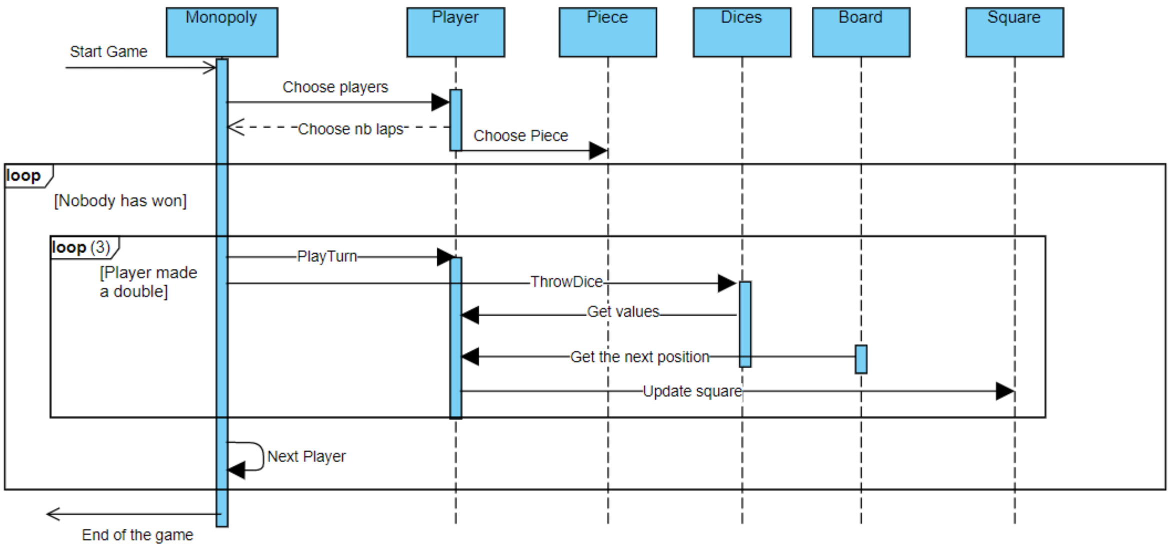
```
public class Jail : IState
{
    Player player;
    4 références
    public Jail(Player player)...
    4 références
    public void Go_In_Jail()
    {
        //Put player in jail
        this.player.ChangeState(new Jail(this.player));

    }
    4 références
    public void Go_Out_Jail()
    {
        //Allow player to go out
        this.player.ChangeState(new OutJail(this.player));
    }

    12 références
    public override String ToString()...
}
```

3. UML diagrams

a) Class diagram of the solution



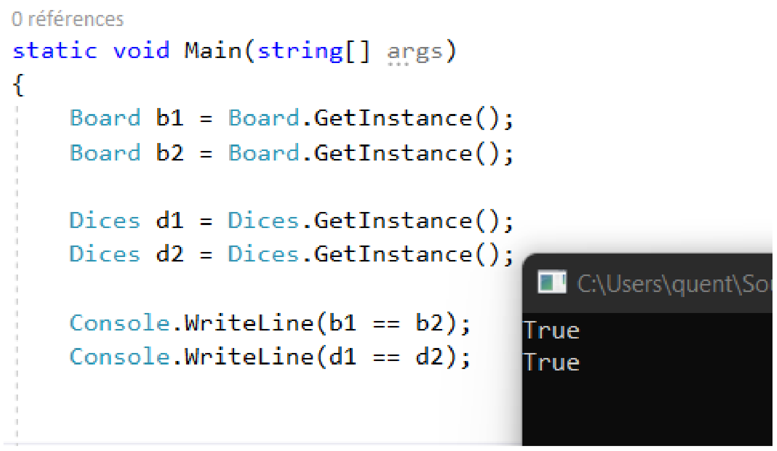
b) Sequence diagrams



4. Test cases

First, we are going to test the singleton pattern on the Board class and on the Dices class. We have created two instances of each class and we test if they are equals.

If the singleton pattern works, we will obtain that the two instances of each class are the same.



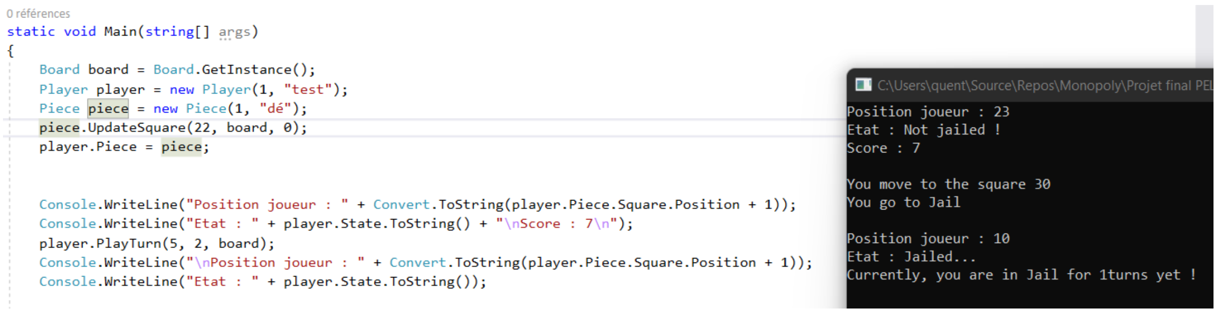
```
0 références
static void Main(string[] args)
{
    Board b1 = Board.GetInstance();
    Board b2 = Board.GetInstance();

    Dices d1 = Dices.GetInstance();
    Dices d2 = Dices.GetInstance();

    Console.WriteLine(b1 == b2);
    Console.WriteLine(d1 == d2);
```

```
C:\Users\quent\So
True
True
```

The two instances created are equals, so the singleton pattern works well.

After this, we have tested the behavior of an instance of the Player class and the State pattern too. We want to see if a player's state changes when he goes on the "Go to Jail" square.

His state should change from "Not Jailed" to "Jailed", and he should move from square 30 to square 10.



```
0 références
static void Main(string[] args)
{
    Board board = Board.GetInstance();
    Player player = new Player(1, "test");
    Piece piece = new Piece(1, "dé");
    piece.UpdateSquare(22, board, 0);
    player.Piece = piece;


    Console.WriteLine("Position joueur : " + Convert.ToString(player.Piece.Square.Position + 1));
    Console.WriteLine("Etat : " + player.State.ToString() + "\nScore : 7\n");
    player.PlayTurn(5, 2, board);
    Console.WriteLine("\nPosition joueur : " + Convert.ToString(player.Piece.Square.Position + 1));
    Console.WriteLine("Etat : " + player.State.ToString());
```

```
C:\Users\quent\Source\Repos\Monopoly\Projet final PEL
Position joueur : 23
Etat : Not jailed !
Score : 7

You move to the square 30
You go to Jail

Position joueur : 10
Etat : Jailed...
Currently, you are in Jail for 1turns yet !
```

The player's position and state have been updated, the Player class and the State pattern work well.

5. Additional / Final remarks

We thought of using two other patterns, but Monopoly being simplified, it was difficult to implement them. However, as an opener, we will list the things we would do if the project was a complete Monopoly:

Factory pattern

First, we thought of using a Factory pattern to design the different types of boxes that exist in Monopoly. Indeed, the Factory pattern is a design pattern that defines an interface to create objects in a parent class, but delegates the choice of the types of objects to be created in the subclasses.

This choice seems judicious because by creating a parent class Case, we can specify all the details of the other cases in different subclasses such as: the stations, the distribution companies, or the various other monopolies present in the game.

Strategy patterns

In addition, we also thought of the Strategy pattern to model the different choices that a player can make when they land on a square: buy the square, pay the rent if it’s already owned, draw a card if it’s a luck or community square, etc...

The Strategy pattern is particularly suitable because it allows to manage more simply the behavior of the player and the choices he can make when he arrives on a square, by modelling the different choices in subclasses to the player's one which stores an action to be performed as an attribute. This pattern is very similar to the state pattern we used above.