Final Project- A Monopoly™ Game

DIA2 | Chloé Coursimault | Héloïse de Castelnau

Table des matières

1. Introduction

To conclude the semester and the "Design Patterns and Software Development Process" course, we had to design a simplified version of the Monopoly™ game in C#. Here are some required features:

- Takes a set of given players.
- The boardgame should be composed of 40 squares and permits circular turns.
- Must include 2 specific squares: Jail and Go To Jail.
- Permits to roll dices.
- The position of the player must change after the throw of 2 dices.
- If the player gets 3 doubles during his turn, he goes to jail.
- A player turn ends after the position of the player changed.
- If a player is in jail, he needs to get a double or to skip his turn three times to get out. After getting out of jail, the player can't roll the dices another time, even if he got a double.

To implement our version of the Monopoly™ game, we did not have any other mandatory requirement but as this course was mainly about design patterns and tests, we included some in our code.

2. Design Hypothesis

The instructions do not mention many features to be implemented so we decided to add some in order to create something that would look like an actual game, even if it would be a really simple interface at the end of the project. Therefore, we decided to add a Start square so that the boardgame will be composed of 4 different types of squares: “normal”, “jail”, “go to jail” and “start” squares. We also added a way to finish the game properly: every 4 rounds, the players will be offered to end the game or to continue it. Then, if the players decide to end the game, a resume of their statistics will be displayed so that they could decide of a winner according to their rules and conditions. Moreover, we decided to display the boardgame at the beginning of the game and the dices after each time the players rolled them.

Before beginning to code, we established a first list of the different classes we needed to implement a simplified version of the Monopoly™ game:

- A player class: it permits to implement different players.
- A square class: it is useful to create the different type of squares needed to build the boardgame.
- A boardgame class: it generates a boardgame constituted of different squares.
- A dices class: it simulates 2 dices.
- The program class, where the main is located.

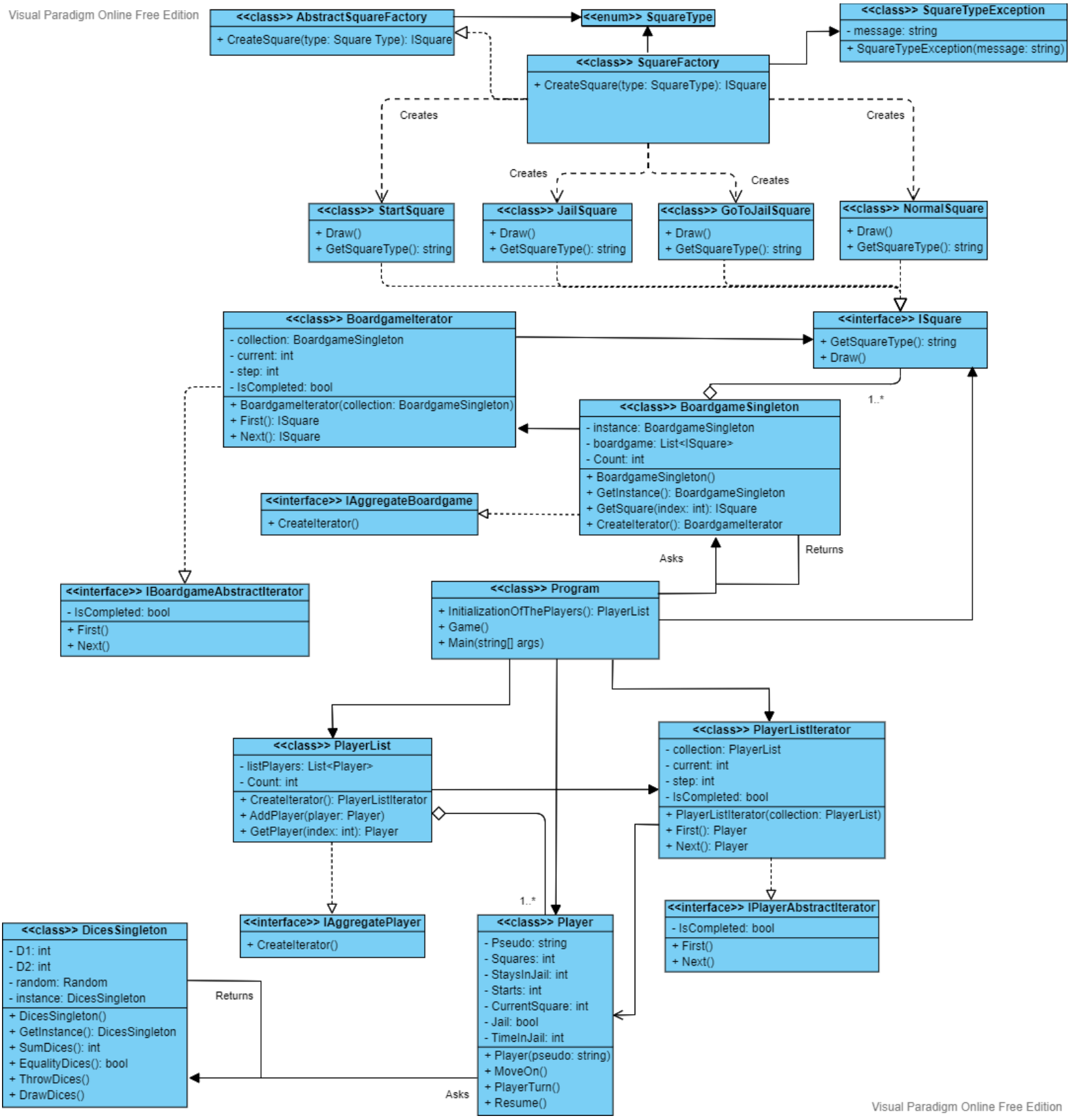
As we mentioned it earlier in the *Introduction*, we were meant to use one or more design patterns in our code to apply what we have learned during the semester. Consequently, we tried to find useful ways to implement some of them in our code. We finally opted for three different design patterns: the factory (creational), singleton (creational) and iterator (behavioural) patterns. We implemented them as follows:

- A singleton pattern concerning the pair of dices: this way, only one instance will be created during the game and those dices will be callable from any class.
- A factory pattern regarding the squares: as we need different types of squares to be implemented in the boardgame, we will use this pattern to create them efficiently.
- A singleton pattern about the boardgame: only one instance will also be created at the beginning of the game, and it will be used to get the corresponding square at a certain position.
- An iterator pattern regarding the boardgame: it will be used to display each square it is composed of at the beginning of the game.
- An iterator pattern about the players: it will permit to display the statistics of each player at the end of the game and to manage the roll of players’ turns.

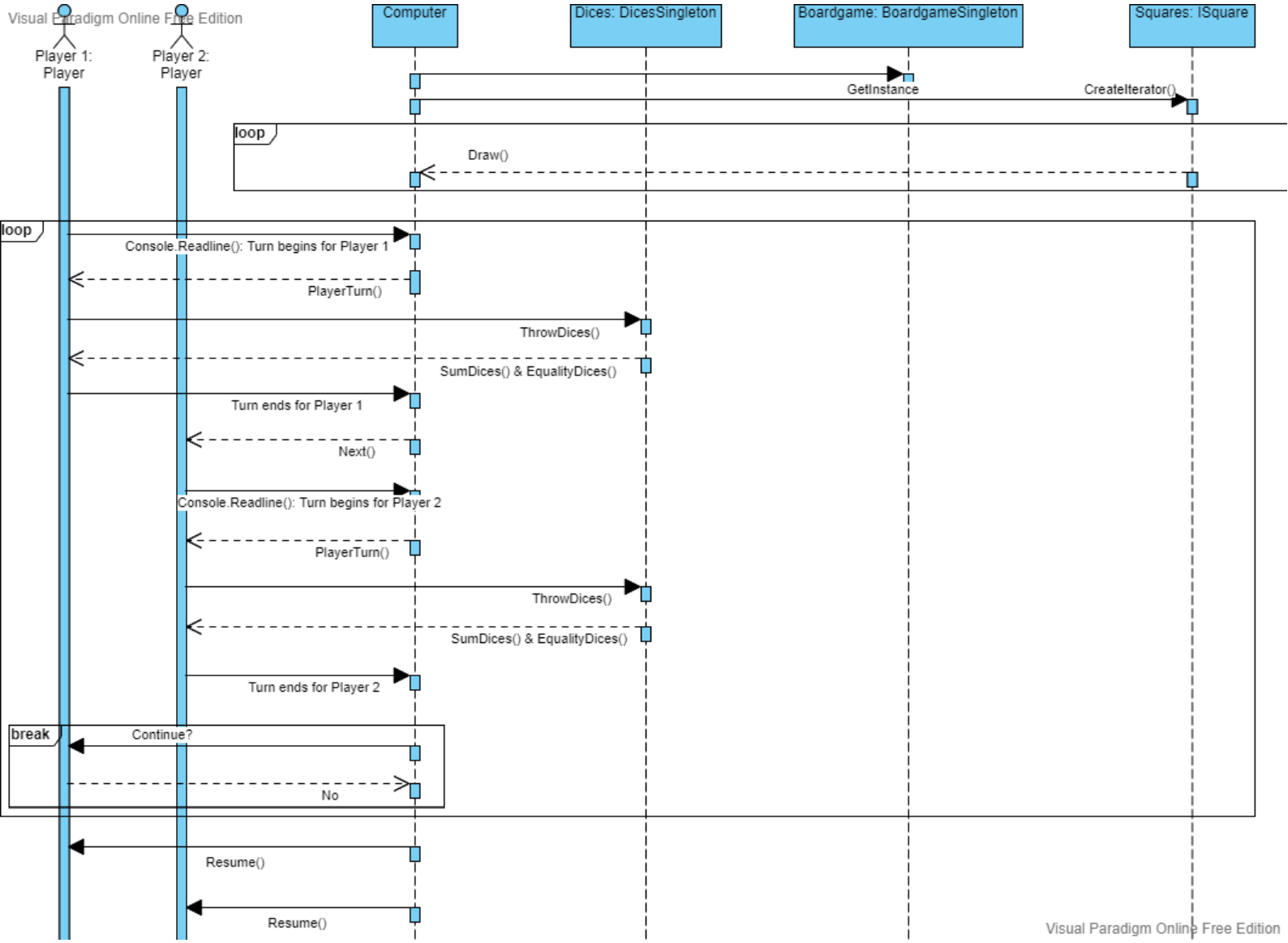
For more information on the different implementations of the needed classes to set the game up efficiently, please refer to the following section.

3. UML diagrams

a. Class diagram of the solution



b. Sequence diagram



To give a quick explanation of the sequence diagram above from the player point of view, we can say that after the set of players has been defined, the game begins with the first player. To begin his turn, he must press the enter button to throw the dices. He is then automatically moved on the boardgame (his position changes) if he wasn't in jail or if he did a double. Eventually, if he was not in jail and did a double, he can throw the dices twice then three times. His turn ends when he finished throwing the dices and moving. The same process is repeated with all players, again an again, until they decide to end the game. When it is the case, the statistics of each player are displayed, and the game is over.

4. Test cases

What is unit testing ?

Unit testing is a sort of software testing that examines individual units or operations. Its main goal is to thoroughly test each unit or function. A unit is the smallest portion of an application that can be tested.

Why unit testing ?

Before any code is deployed, it is subjected to unit testing to ensure that it fulfils quality requirements. This promotes a dependable engineering environment that prioritizes quality. Unit testing **saves time and money** across the product development life cycle, and it helps developers design better, **more efficient code**.

How did we implement unit testing ?

Thoroughly our coding we tried to test all of our methods inside our classes. We created four groups of unit testing linked to the major four Design Patter Classes we implement : **Player, Board game, Square & Dices**.

We followed the basic unit test life cycle and adapt our code through the errors detected for each method.

```
#region Design Pattern Dices

[TestClass]
public class DicesTests
{
    [TestMethod]
    public void DiceSingletonTest()
    {
        DicesSingleton dices = DicesSingleton.GetInstance;
        Assert.AreEqual(dices.GetType(), typeof(DicesSingleton));

    }

    [TestMethod]
    public void DiceSingletonTest2()
    {
        DicesSingleton dices = DicesSingleton.GetInstance;
        DicesSingleton dices2 = DicesSingleton.GetInstance;
        Assert.AreEqual(dices, dices2);

    }
    [TestMethod]
    public void SumDiceTest()
    {
        DicesSingleton dices = DicesSingleton.GetInstance;
        dices.D1 = 5;
        dices.D2 = 6;
        Assert.AreEqual(11, dices.SumDices());
    }

    [TestMethod]
    public void EqualityDiceTest()
    {
        DicesSingleton dices = DicesSingleton.GetInstance;
        dices.D1 = 5;
        dices.D2 = 6;
        Assert.IsFalse(dices.EqualityDices());
    }

    [TestMethod]
    public void ThrowDiceTest()
    {
        DicesSingleton dices = DicesSingleton.GetInstance;
        dices.ThrowDices();
        Assert.IsTrue(dices.D1 != 0 & dices.D2 != 0 );
    }

}
#endregion
```

A basic example of Unit Testing is the **TestClass DicesTests.**

We tested the method linked to the Dice class such as the **SumDice** (which will return the sum of dices), the **EqualityDice** to check if two dices have the same value (to go out of jail for example), the **ThrowDice** of course to verify that the value of the dice change.

To do so, we called the **Assert Class** which is a collection of helper classes to test various conditions within unit tests. If the condition being tested is not met, an **exception** is thrown.

When an exception was thrown, we reviewed the original method, changed it and renewed the test.

5. Conclusion

To sum up, we found the project interesting and appropriate in order to test our knowledge on design patterns and testing.

In fact, the necessity to use different classes of diverse types was suitable to test various patterns. Therefore, we decided to use 2 creational patterns - Singleton and Factory - and 1 behavioural pattern - Iterator. Unfortunately, we did not find the opportunity to use any structural pattern this time.

Regarding the testing part, we decided to test all parts using unit testing. To develop our code efficiently, we proceeded with the test-driven development method so that we were ensured each part of the code was implemented as we wanted.