Mikdaam BADAROU **Yunen SNACEL**

OBJECT ORIENTED PROGRAMMING IN JAVA

SPLENDOR

Developer's Guide

ESIPE INFO - Year 1
2021 - 2022

SUMMARY:

INTRODUCTION

Splendor is a board game. The goal is to buy several cards in order to get 15 prestige points. A good strategy is essential to win this game since you will have to choose your tokens and the cards you buy wisely in order to be the best precious stones merchant.

THE PROJECT'S ARCHITECTURE

Text.

IMPLEMENTATION CHOICES

THE CARDS

The solution we chose to implement the cards is pretty standard and not hard. We chose to create an interface **Card** that would represent a card, and then create two types of cards: the **DevelopmentCards** and the **NobleCards**. Since all the cards have a lot in common (prestige points, a color (precious stone), and a price), it was obvious for us to use an interface here. Then we only had to create two different types to represent these different cards. The main difference is that a NobleCard already knows its color and it cannot be changed (its color is **NOBLE**).

THE TOKENS

At first, we have created an interface to represent a **Token** and we have created a **BaseToken** type to represent a basic token. The idea behind that was to create another type for the gold token first, and have the possibility to create other types of tokens for the extensions in the ameliorations.

However, during the development, we eventually found out that all these types weren't necessary. Indeed, we had created an enum **Color** to represent the different precious stones, and the more we coded, the more we were using the colors instead of the tokens directly. Actually, the tokens became a problem because we had to access to its color and the type itself was no longer used.

We have then decided to remove the Token types and only keep the colors to represent the tokens in game. It became more convenient to code without the Token types.

THE COLORS

Just as the tokens and the cards, the colors were one of the first types we thought about. This type represents the different colors (mostly the types of precious stones) a token or a card can have. The idea was to have a single and immutable value for each one of these different colors and to be able to use it everywhere.

Fortunately, such a thing already exists in Java, which is why we have created the **Color** type as an enum. It suits our solution perfectly. It's useful for us since we don't have to remember the values of each type (actually, we don't even have to know them), but it's also better for the code for it to be as clear as possible. The type contains mostly the names of the precious stone, and a "NOBLE" color for the nobles.

THE GAME MODES

To create the game modes, we first realized that the two of them weren't asking for the same things in term of actions but also in terms of cards and tokens. In the simple game, nobles don't exist, as well as reserving a card and therefore the golden tokens. Moreover, the two modse do not have the same cards to play with.

To respect this, we have created an interface called **Game** from which we have created two different games, representing the two different modes: **SimpleGame** and **NormalGame**. It was the best way for use to separate those two different mechanisms.

THE ACTIONS

To create the actions, we first wanted to create a type for each one of them and create a method that would apply these actions. However, to create this method, we needed to pass all of the game's data as parameters, and we weren't sure if this was a good thing to do. Our idea with this implementation was to be able to create as many actions as we wanted and simply add them to the game modes, making them independent from them.

So, we first have developed our action into the game modes, eventually duplicating the code. Once we had the confirmation during the β oral, we have refactored the actions into their own types. We manage to implement the action as we wanted and we can even add more actions and add them easily to some new game modes.

THE DISPLAYERS

We thought a lot about this functionality. It was probably the hardest to design. To help use with that, we took inspiration from the MVC design pattern. And by taking inspiration from this, we eventually came out with the following solution: creating a **Displayer** interface and two different types of displayers, which are **ConsoleDisplayer** and **GraphicalDisplayer**.

This way, each displayer can manage its way of displaying the data independently from the other. And to make it work correctly, we decided to add the displayer as a data for the game, so it can call the displayer directly with one single method called **display**. This method then displays the game as its intended to.

THE ARGUMENTS

We were already thinking about parts 2, 3 and 4 of the project when designing it and we were thinking about the best way to ask the user about their game's parameters. We first decided that what we had to know was the game's mode first, then the number of players in case the normal mode was chosen, and finally the display mode.

The best way we found was to pass this information on the command line used to launch the game. The user must enter the mode and eventually the number of players on the command line, and our program treats this information in the main, directly and executes the right mode with the right parameters with the right displayer.

This was mostly to avoid anything asked on the console when launching the game and separate clearly the two different display modes.

THE β ORAL

During the β oral, we mainly discussed the refactoring we could do in our code. After our implementation was validated, it was suggested to us that we refactor several methods.

For example, in each game modes, we had a code for the action. We already had the idea to refactor it in several types (one for each action) and a method apply in each one of these types to apply the action. We did not have the time to do it before the oral and we wanted to be sure that it was not a problem to give all the game's data to the method. After confirmation, we successfully refactored our action codes in their own types.

It was also suggested to us to re-use the types we have created instead of using the default java ones. For example, for some methods, we were returning a **Map<Color, Integer>** where we could have simply returned a **TokenPurse** which encapsulates this type in our project. This also grants more control on our data. So, we have edited and changed all of this in our code, to re-use as much as we could our own types.

One major problem in our code was that were using a lot of arrays (for our board, our players, etc), which was a poor solution given the fact that arrays are not used commonly in Java and that they do not have many methods, unlike the **ArrayLists** for example. So, we have decided to change all our arrays to implement **ArrayLists**, which was more convenient for our project. Replacing the arrays also allowed us to delete a type we have created to represent an empty card. We were using this type to avoid manipulating nulls in our arrays but also for esthetic reasons. We have also created a brand-new type for our coordinates, a simple record, instead of using an array of two integers.

One thing we were doing in our code before the oral was editing our data directly (especially the tokens) instead of returning new objects. We have corrected that by making our types immutable.

Finally, some suggestions were given to correct and change some of our methods names, which we corrected after the oral.

PART 4 - AMELIORATIONS

Unfortunately, we did not have the time to implement any ameliorations that were proposed in the subject. We focused on the three first parts for them to work perfectly, and we ran out of time for these ameliorations

However, we believe that we could have implemented one extension easily with our architecture. We would just have added the necessary classes for the potential new objects, a new game mode and therefore probably new actions, and that would have been all.

CONCLUSION

This project allowed us to use all the knowledge we have seen in class. It allowed us to practice and see how everything works in a bigger and complete project. It was also challenging to have two different ways to display the game. It forced us to think wisely about our architecture, together, in order for it to work for all the project's part. We have spent some time to think about it but it eventually paid off. When coding the different parts, we noticed that the architecture was good enough since we did not have any problem to implement the different parts together.

It was also fun to play the real game all together first, with our classmates, to understand the mechanisms of this game. We also managed to play with some people to our own version of the game once it was functional, which was really fun too and gave us this feeling of pride to see our work functioning correctly and people having fun with it.