Mikdaam BADAROU **Yunen SNACEL**

OBJECT ORIENTED PROGRAMMING IN JAVA

SPLENDOR

Developer's Guide

ESIPE INFO - Year 1
2021 - 2022

SUMMARY:

INTRODUCTION **3**

THE PROJECT'S ARCHITECTURE **3**

THE CARDS 3
THE COLORS 3
THE LEVELS 3
THE PLAYERS 4
THE GAME MODES 4
THE ACTIONS 4
THE DISPLAYERS 5
THE CARD DECKS 5
THE TOKEN PURSE 5
THE BOARD 5
THE ARGUMENTS 6
THE UTILS CLASSES 6

THE PROJECT'S REVIEW **6**

WHAT WORKS 6
WHAT DOESN'T WORK 6
USER GUIDE 7
CONTROLS 7
LAUNCHING COMMANDS 7

IMPROVEMENTS **8**

CONCLUSION **8**

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

THE CARDS

The cards have been designed with an **interface**, called **Card**. This interface represents a basic card and everything a card should "know". So basically, it should know its level, its color, its amount of prestige points and its price. This everything we can find in common between the different cards of the game.

Then we have two types, represented by two **records**: **DevelopmentCard** and **NobleCard**. These two types represent the development cards and the noble cards of the game respectively. We chose to use records because both types only represent data, not mechanisms.

THE COLORS

To represent the different colors (precious stones) a token and a card can have, we have created an **enum** called **Color** that lists all the colors possible in any game mode. So, in this enum, we can find values such as: **RUBY**, **ONYX**, **EMERALD**, etc.

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. 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 LEVELS

To represent the different cards levels, we have created an **enum** called **Level** that lists all the levels possible in any game mode.

THE PLAYERS

Since it was asked to have two different types of players at the end of the project, we have created an **interface** called **Player** to describe what a player should be. From this, we have created the **class HumanPlayer**. This class represents the data a human player has and all the mechanisms linked to a player. So basically, it contains methods describing what a player can do all by itself.

We expected to add another **class RobotPlayer** to represent an artificial intelligence for the fourth part of the project. However, we did not have the time to implement it. But everything was ready for it to be built, thanks to our architecture.

THE GAME MODES

Since we had to create, at least, two different game modes, we decided to create an **interface** called **Game** that describes the basic things a game should do. For instance, a game should initialize itself, run itself, choose an action, etc. From this, we have created two different **classes**, representing two different game modes: **SimpleGame** and **NormalGame**.

We managed to refactor a lot of code into the interface between the two classes, but each game mode initializes and runs itself in a different way.

To manage the data more conveniently, we have created a **record GameData** that contains every data a game should have (such as its board, its decks, its players, etc). It was necessary to have a good interaction with the actions.

THE ACTIONS

We knew we wanted something simple and universal for every action. The ideal was to have a same method for all of them and each action would adapt this method in their own way. This is why we have created the **GameAction interface** to represent an action. An Action is a **record** that has an **apply method** to apply the action it represents.

Then we have created several records, one for each action:

- **BuyCardBoardAction**
- **BuyReservedCardAction**
- **GiveBackTokensAction**
- **ReserveCardBoardAction**
- **ReserveCardDeckAction**
- **ThreeTokensActions**
- **TwoTokensAction**

Then, we have created one last **enum** called **ActionType** to describe each action. It's mostly used in the game loops when choosing the action.

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 **classes** of displayers, which are **ConsoleDisplayer** and **GraphicalDisplayer**.

Both classes have common methods of course, the most important one being **display**. But each one of them implements this method differently. We have created the **GraphicalDisplayer** class, but we did not have the time to implement it correctly…

THE CARD DECKS

To represent the card decks, we have created a single **class CardDeck**, which implements all the mechanisms linked to a card deck, such as adding and removing a card, counting its number of cards and number of prestige points, creating a summary of itself, etc.

THE TOKEN PURSE

To represent the token purses, which are present on the game board (this is the bank) and on each player, we have created a single **class TokenPurse**, which implements all the mechanisms linked to a token purse. For example, we can add and remove some tokens, tell the number of tokens associated to a color, etc.

THE BOARD

To represent the board, we have created a single **class Board**, which implements a basic ArrayList of ArrayLists of cards. This board implements mechanisms such as adding a card, removing a card, counting its number of cards, etc.

At first, we thought that we would only use the board for the main board where the players can buy cards. But we eventually realized it was also convenient to use it for the nobles and for the reserved cards.

THE ARGUMENTS

It was asked to be able to choose the game mode, the display mode and the number of players for each game. Each of these information was meant to be chose independently from the others (except the number of players for the simple mode game).

We chose to use the command line to pass these arguments. The user should enter different options (described next) on the command line to tell what they chose. We have decided to parse them directly in the **Main class** for more convenience. From there, we create the displayers and the games we need before initializing them and launching them.

THE UTILS CLASSES

We needed to create two more classes, which both contain methods that have nothing to do with the game's mechanisms. These **classes** are **Utils** and **FileLoader**. They contain methods to parse a file to get the cards, to align several strings on a single row (it was used to align the cards in the console display mode), etc.

THE PROJECT'S REVIEW

WHAT WORKS

In final, we managed to fully complete parts 1 and 2 of the project. Both parts work correctly, can be played until the end and with no apparent bugs. We have managed to create a fancy looking console mode for everyone to play. Since the β oral, we have corrected what we talked about with M. Carayol. For example, we have refactored the actions' codes, but also corrected the fact that we were opening a new scanner every time we asked something to the user. We have also added the action to give some tokens back when a player took more than ten, which made the games playable until the end.

WHAT DOESN'T WORK

We didn't manage to complete parts 3 and 4 of the project. We have started to do something for part 3 but we didn't have the time to finish it. However, we believe it would have been easy, with a bit more time, to complete these parts. Indeed, we have thought our architecture in order to implement parts 3 and 4 as easily as possible. The fact that we have separated the display mechanisms from the rest makes it easy to implement another display mode. Moreover, to implement another type of players and an extension, it would have been easy too thanks to our architecture.

USER GUIDE

CONTROLS

In console mode, the game can be played exclusively by typing numbers and words on the console. Here are the different controls you can use in this mode:

- **SELECTING AN ACTION**: You will see a list of actions at each player's turn. To select an action, you just have to enter the number associated with the action.
- **SELECTING A TOKEN**: To select a token, you have to write the color entirely on the console. (For example: EMERALD)
- **BUY A CARD:** To buy a card (from the board or from your reserved cards), you will have to enter its coordinates. The row corresponds to the card's level (the level is on the deck on the left), the zero for the columns starts at the first column on the left.
- **RESERVE A CARD (exclusive to the Normal mode):** Reserving a card can be made by using the same commands as buying a card.

LAUNCHING COMMANDS

A **build.xml** file is present in the game's root repertory. This file allows you to execute several actions, such as:

- **`ant compile`** compiles the game and stocks all the ***.class*** files in the **classes** repertory.
- **`ant jar`** generates the ***.jar*** executable at the game's root repertory.
- **`ant javadoc`** generates the *javadoc* in the **docs/doc** repertory.
- **`ant clean`** cleans the repertory (deletes all content in the **classes** repertory and the generated j*avadoc*).

To launch the game, you will have to enter several arguments:

```
java -jar splendor.jar [arguments]
```

- **`--mode simple/normal`** to select the game mode.
- **`--players N`** to select the number of players (exclusive to the normal mode)
- **`--display console/graphical`** to select the display mode

Examples:

```
java -jar splendor.jar --mode simple --display console
```

```
java -jar splendor.jar --mode normal --players 3 --display graphical
```

IMPROVEMENTS

We guess we could have improved our way to handle the GameData. We wanted to have every field as a **final** field, but it has become quickly difficult to handle this. And since it was a last change, we preferred to remove the **final** field where it caused some problems and handle as best as we could the GameData.

We could also improve our coding style. Sometimes, we do not declare some variables to gain some space and declare directly a value by calling some methods.

We clearly didn't manage our time correctly, thus preventing us from finishing at least the third part. We can clearly improve on this during the next two years and the other projects.

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.