Vrije Universiteit Brussel

Final Project Report

Programming in JAVA

Siyan Luo

Jan 2, 2023

User manual

Players are required to enter their choice when it comes to play or discard a card. The choice is valid only if they enter an integer between 0 and X, where X is the total number of cards that can be chosen. If the players want to know about current situation to help them make wiser choices, they can input show keepers, which shows their displayed keepers, show goals, which shows all the played goals, show rules, which shows all played rules, show hand, which shows the cards in hand. They can also enter help, which shows all the options they are allowed to input. When the deck is empty, players should play cards directly instead of drawing cards first. When the players run out of the cards in hand before reaching the play limit, their turn ends in advance. The goal will be checked after every played card. At the end of someone's turn, if there is a winner, the current player does not need to discard cards according to current rules, the game ends directly with the result.

Design choices and special elements included

In order to distinguish cards quickly, we set an instance variable of id to Card Class and assigned an id value to every instance of card. In that case, when the player plays a card, we can tell its type according to the id and make corresponding reactions. For example, the player plays a card with the id of 111, which falls within the predefined range of hand limit rule, then we renew current hand limit rule.

The usage of id also makes it easier when we check if there is anyone reaches the goal. We found that for most of the goals, their contents contain two keepers. In other word, if we want to check the winner, we have to traverse every player's displayed keepers to see if there are the targeted two. To make the check process more efficient, we set an instance variable of goalKeepers to Goal Class. It is an array list of integers where the integers indicate the id of targeted keepers.

Since there are special goals such as the player with more than 5 displayed keepers wins or the player with more than 10 cards in hand wins, we check its goalKeepers' size first. For instance, for the 5 keepers goal, when we instantiate this goal, we set its goalKeeper as an array list of five 0s. If the goalKeeper's size is 5, we can tell it's the

5 keepers goal, then we just check every player's displayed keepers' size.

While the idea of id has simplified the programming process, there does exist a problem regarding the extensibility of the code. How do we adapt the code if we want to add new cards or a new type of card? To tackle this issue, we set a wider range of id for each type of card. When adding a card, we assign it an id which is not used yet within the corresponding range. But after adding a bunch of cards, the predefined range may be filled up. When it comes to adding a new type of card, we can also set a new range of id available for it and add corresponding code without the necessary of making much modifications to the exist code.

For the design of instance variables in Game Class, as we noticed there are four types of rule cards, Hand Limit, Keeper Limit, Play Limit, Draw Limit, which can coexist in the game, we set four instance variables each represents the current one. We did not design them as the subclass of Rule Class, we tell them by its id. When we want to get the limitation value, we just call the getRuleValue method since we created an instance variable of ruleValue in Rule Class, which stores the limitation.

In our game, every discard card is added to the discards list, so is the replaced rules and goals. Then when the deck is empty, we recycle the discard cards as new deck.

In the play method, at the beginning, we let the player to play continuously to reach the play limit. Then we found it incorrect since every played card may introduce new rule or goal. For this issue, we created a parameter counter to count the number of cards played in a turn. The player has to play card one by one, only if there's no one achieves the goal and the player has not reached the play limit, he can continue playing. We make play method recursive with updated counter.

About the error handling, we set a Boolean value to check the validity of user input and used it as the condition of while loop. Besides, we used try catch statement to catch the invalid data type error.

To avoid the usage of magic numbers, we used constants (static final) to initialize instance variables in constructors of Game Class.

Division of workload and teamwork experience

For the initial design of UML, both of us participated in and made lots of modifications to decide on the final version. I completed the UML drawing. For the code implementation, we discussed together to get the blueprint of the whole structure. Then my teammate first created all the classes and their basic instance variables along with getters and setters. I coded mainly the methods following behind: initializePlayer, discardHand, discardKeeper, draw, play and userInput. I also finished the main method Class the run the game. For the testing and debugging, we worked together. For the documentation, we wrote the javaDoc of methods which are designed by another to get more familiar with the whole code. All in all, it's good teamwork experience with everyone involved equally and actively.