Final report

Cassino (Demanding difficulty)

I. Personal Information:

Date: May 11th 2021

Name: Duong Le

Student number: 894834

Degree program: Data Science

Year of studies: 1st year

II. General description:

The game is a simple card game revolving around capturing cards in the table using those in your hand. One deck of card has 52 cards, each has a suit and a value. There are overall 4 suits and 13 possible values. The game can be played with other human or computer player. However, for the sake of simplicity of the game, the maximum number of human players is 6 and computer players is 3. More information on the rule:

2.1 Rounds:

The deck is shuffled in the beginning of every round and the dealer deals 4 cards to every player (they are not visible to other players) and 4 cards on the table (visible for everyone). The rest of the cards are left on the table upside down.

2.2 Actions in each round:

A player can play out one of his/her cards: it can be used either for taking cards from the table or to just putting it on the table. If the player cannot take anything from the table, he/she must put one of his/her cards on the table.

If the player takes cards from the table, he/she puts them in a separate pile of his/her own. The pile is used to count the points after the round has ended.

The number of cards on the table can vary. For example, if someone takes all the cards from the table, the next player must put a card on the empty table.

Player must draw a new card from the deck after using a card so that he/she has always 4 cards in his/her hand. (When the deck runs out, everyone plays until there are no cards left in any player's hand).

Player can use a card to take one or more cards of the same value and cards such that their summed value is equal to i * the used card for some integer i.

2.3 Sweep:

If some player gets all the cards from the table at the same time, he/she gets a so- called sweep which is written down.

2.4 Special cards:

There are a couple of cards that are more valuable in the hand than in the table,

- Aces: 14 in hand, 1 on table
- Diamonds-10: 16 in hand, 10 on table
- Spades-2: 15 in hand, 2 on table

2.5 Scoring:

When every player runs out of cards, the last player to take cards from the table gets the rest of the cards from the table. After this the points are calculated and added to the existing scores.

The following things grant points:

- Each sweep grants 1 point.
- Each Ace grants 1 point.
- The player with most cards gets 1 point.
- The player with most spades gets 2 points.
- The player with Diamonds-10 gets 2 points.
- The player with Spades-2 gets 1 point.

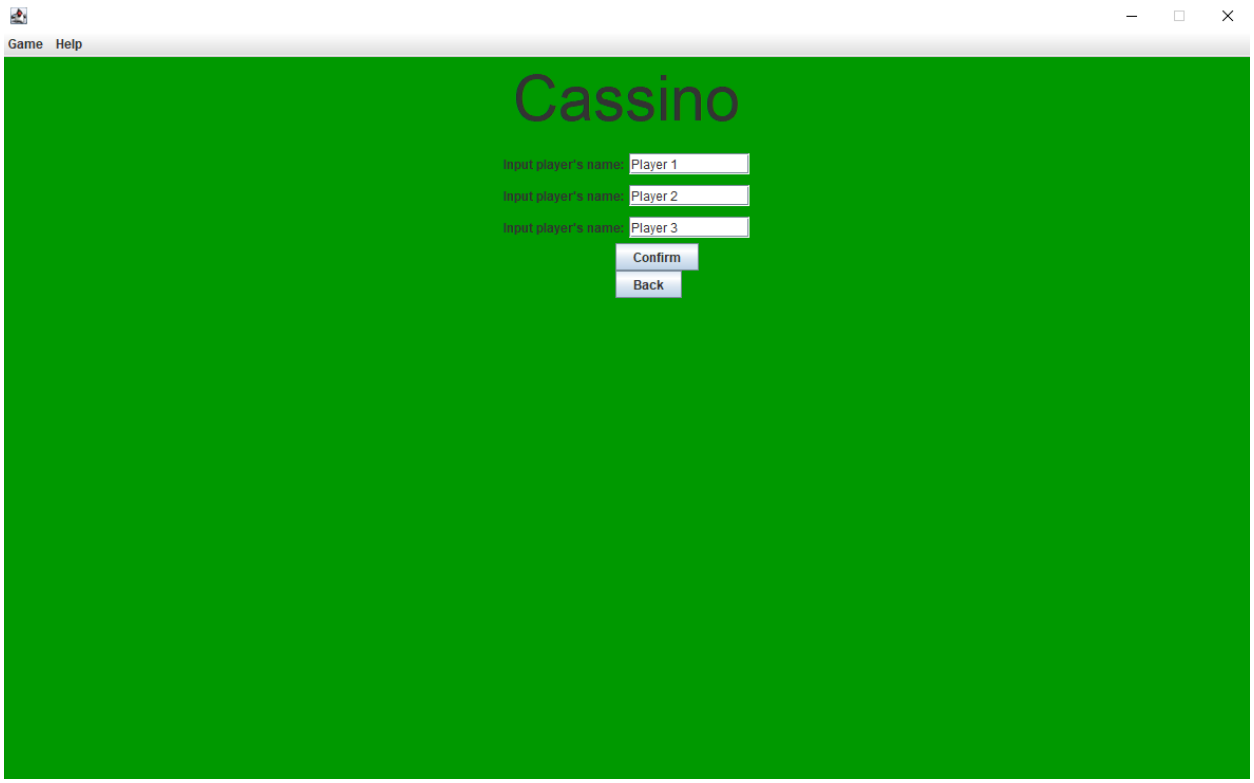
Substantial changes compare to the general plan:

- The maximum number of players is 9 instead of 12
- Players can take cards as long as their sum equals to i * the used card for some i instead of equals to i. This is the result of a bug in the validCheck method.

III. User interface:

When open the program, the user can choose the number of human and computer players for the game and then click “Ok”.

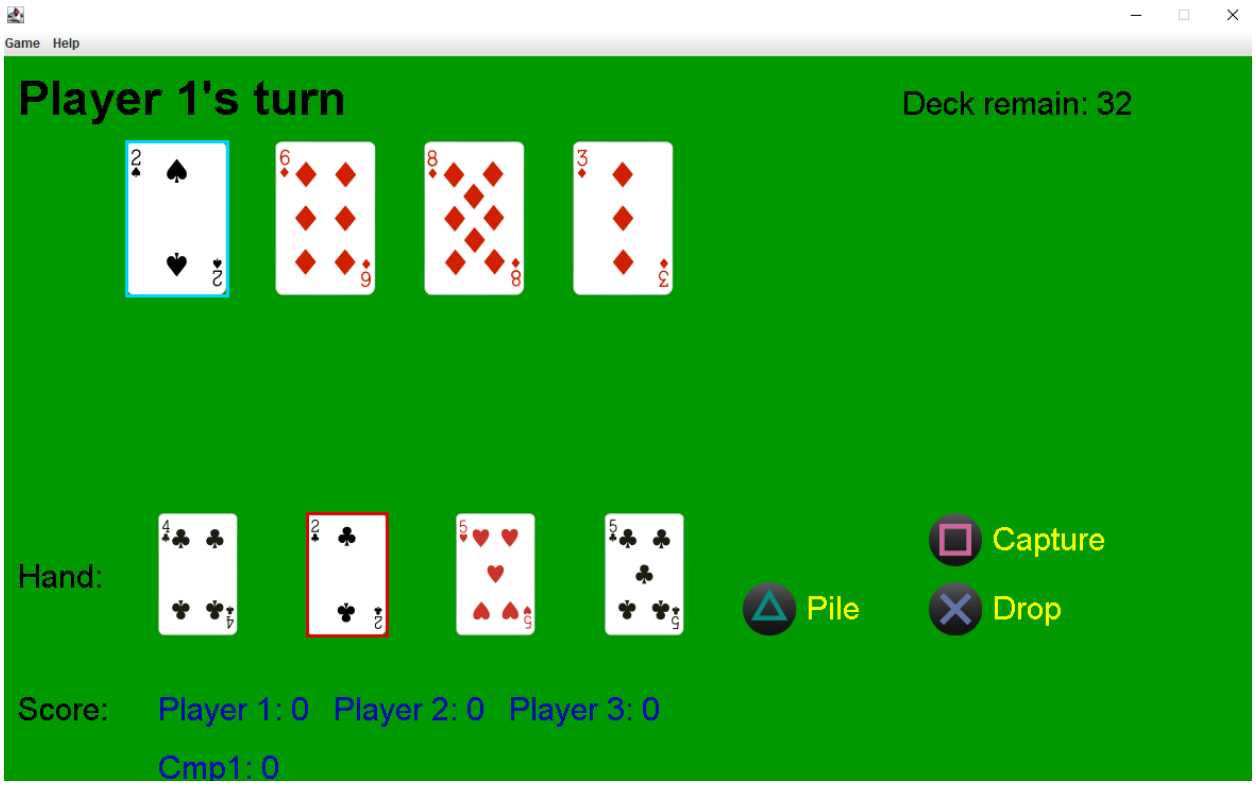
The program will then take the user to the fields to input the player's names. The computer player's name will be Cmp1, Cmp2, and Cmp3. If the user wants to change the number of players, they can click "Back". When everything is set, the user can hit "Confirm" to start the game.



The game screen has 4 sections. The first section is the info section, locates on top of the screen. It has the name of the player of that turn and the number of cards left in the deck.

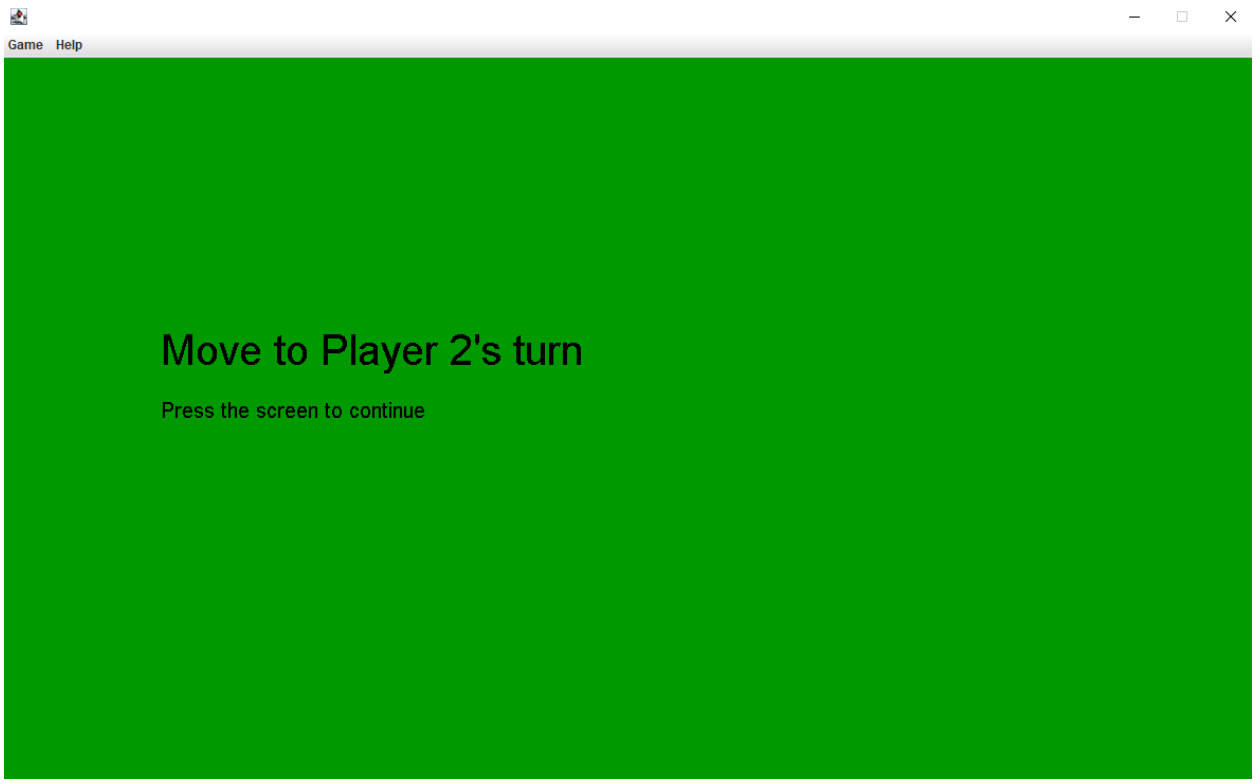
The second part is the table section, where the images of the table cards are displayed. Maximum of 14 cards can be displayed. If the number of cards exceed 15, the excessive will not be displayed. But after running a few test plays, I did not encounter such problem.

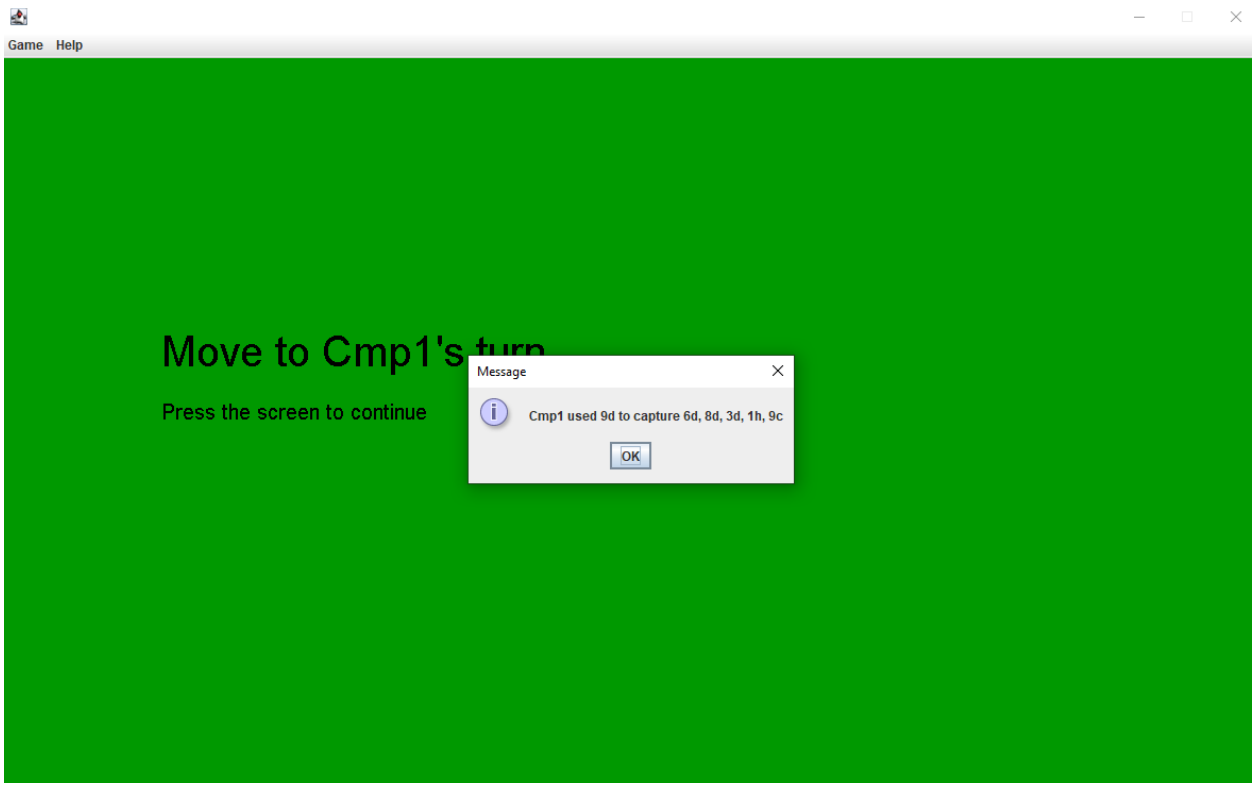
The third section is the hand section, where the cards on the hand are displayed. There are three buttons: Pile, Capture, and Drop. The Pile button used to check the pile cards of the player. The player can choose cards on hand and table and use the buttons Capture or Drop to perform action in a turn. The final section of the game screen is the score section. The human player will be displayed on the first line, and the computer players are on the second line.



In case the player attempts to perform an invalid action, a dialog will pop up to notice the player.

After a turn is complete, a screen will show up to announce turn change. User can click on the screen to move on. In case the next player is a computer player, a dialog will pop up to announce the player's action (e.g what card the player used)





When the game is complete, a scoreboard will show up with players who has highest scores colored in red.

If the user wants to know the rules, they can click the Help menu and then Instruction button, a new window will open with the instructions.

The user can save or load file using the Save and Load buttons in the Game menu. When clicked, they will open a file chooser, where the user can specify the file that they want to use.

IV. Program structure:

The program has two main packages: Logic and GUI.

4.1. Logic:

This package contains 5 classes and 1 object.

4.1.1. Game:

An object simulates a game round. The most significant variables of the Game object are the *players* buffer that store the players of the game, the *deck* variable represents a deck of cards, the *turn* variable used to keep track of the turns. The most important methods that the Game object has are *validCapture*, used to check if a capture is valid or not, *calculateScore* to calculate the score of each player, the *newGame* method to create a new game using the provided number of computer players and a list of players for that game, and the *reset* method to reset the game state to initial states.

4.1.2. Player:

A class represents players of the game. Each player has variables to store the hand cards, pile cards, score, and the sweep count. In addition, each player also has the methods *capture* and *drop* to perform actions in a round.

ComputerPlayer is an inherit class of Player, and it has in addition the method *optimalMove* to perform actions in each round, along with some helper methods for *optimalMove*.

4.1.3. Card:

This class represents the cards used in a game. A card has its *name*, *value*, *suit*. A card also has a *handValue* which is the value of the card when it is in the hand of the player, and an *image* variable used to store the path to the image of the corresponding card.

4.1.4. Table:

Table class simulates a table used in a game. Table has a private variables *cardsOnTable* to store the cards that are currently on the table, and the methods *allCard*, *addCard*, and *removeCard* used to access and modify *cardsOnTable*.

4.1.5. IOHandler:

This class is used to handle the save and load function of the game. It has only two methods: *saveGame* to save the progress to a file and *loadGame* to load from a file.

4.2. GUI:

The GUI package contains 4 objects.

4.2.1. CassinoApp:

This is the object used to manage other screens of the game, and the window that the user sees when using the app is the MainFrame of this object.

4.2.2. FirstScreen:

This object is the first screen and the name screen of the game. Player selection and naming players will be done on this screen.

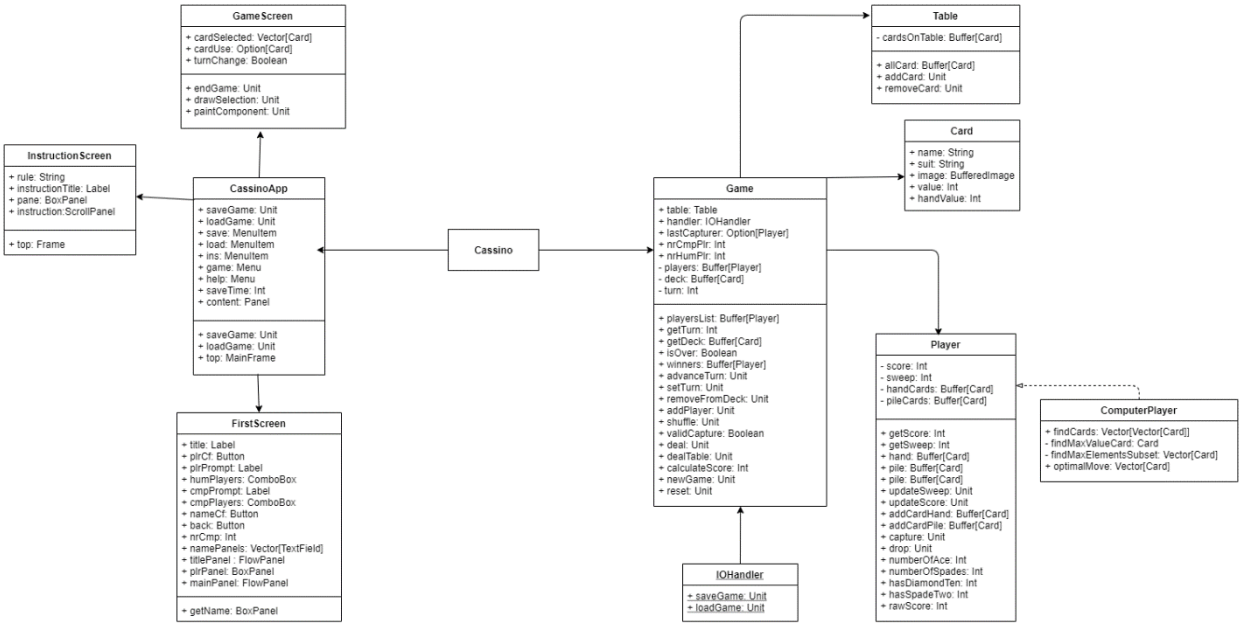
4.2.3. GameScreen:

This is an object that displays the game using the players information that the user provides in the previous screen. The game progress and user's action during the game are displayed in this screen.

4.2.4. InstructionScreen:

This object is a separate window used to display the instruction for the game. By making it into a separate window, user can play the game and check the rules if they are still confused simultaneously.

Below is an UML model to visualize the class structure of the program.



V. Algorithms:

The most important algorithms in the program are the algorithm used to validify a capture attempt and the algorithm to determine the optimal move for a computer player.

5.1. validCapture algorithms:

The algorithms will first check for overlapping by turning the cards Vector that the user provide into a Set. This way, any overlapping card will be removed. Then, the values of the cards are summed together. If the summed value is equal to i * card used hand value, then there are no overlaps, and the capture is valid. Otherwise, the capture is invalid.

However, during the tests, I noticed that there are some combinations that still passed the test without meeting the criteria of the game. For that reason, I need to modify the game rule a bit to work with my algorithm.

5.2. optimalMove algorithms:

The basic idea of the algorithm for the optimal move is to maximize the number of cards captured and at the same time, keeping the high value cards on the hand. Of course, the more card the player can capture, the more advantage that player can be. But at the same time, it is also important to keep cards with high value on hand since they are easier to capture other cards. So as a solution, I decided that if the player can sweep the table, then he will sweep. Otherwise, the player will get for each card the maximum number of cards that can be captured, and then sum it up with the hand value of the card used. The card that has the lowest value of the sum will be choose. The algorithms also include some specification for the case if there are too many cards on the table, but it seems during the test plays, the number of cards on table never reach that border.

If there are no possible card to used, the player will drop the card with the highest value. Since the higher card value are harder to capture, this can create an advantage for the player.

Although the algorithm is just my estimation, but during the two test plays that I perform, the results seem optimistic. The computer player can sometimes hinder other players attempt to capture cards and perform clever captures. Of course, the number of test run is too small to fully assets the algorithms, but it still poses a positive sign.

VI. Data structure:

The most common data structure of the program is Vector and Buffer since they are the basic structures and easy to use. The default structure for a variable is Vector, and if I need to change the state of the variable frequently, I use Buffer for easier modification.

In addition, I also use Map and Set. Map can be useful in some cases. For example, in the helper method *findMaxElementsSubsets*, I need to keep track of both the card and the number of elements in its subsets. So mapping the card with its subsets would be a reasonable choice. Set is useful when I need to check for overlapping. I just need to turn the Vector into a Set and compare its length to the original vector, then I can decide if there are elements that appear more than once.

VII. Files and Internet access:

The program uses text file to save and load game, and the format of the file is as following:

The game will save its data on a human readable text file. Each file starts with "Casino Save ver 1.0" and ends with #END. The file consists of chunks, each chunk starts with "#". The chunks in the save file are:

7.1. #Metadata:

This chunk will store the number of players in the game.

7.2. #Players:

This chunk includes smaller chunks to store player information, starting with *. Each player chunk will start by denoting if a player is a computer player or not, following by the player's name, hand cards, pile cards, and score.

7.3. #Turn:

This chunk contains the information about the turn. The chunk starts with #Turn, and has the turn count and the cards left on the table.

7.4. About card format:

Each card will be noted by the card name and the abbreviation of the card suit (S for spade, H for heart. C for club, and D for diamond). The name of the card will be a number from 1 to 9 for ace to nine and 0 for tens. Cards such as Jack, Queen, and King will have the name denoted as J, Q, and K.

Example: 8c -> 8 of Club, 0d -> 10 of diamond, ks -> King of spade.

7.5. Example of a save file:

Casino save ver 1.0
#Metadata:
No. of players: 2

```
#Players:
*Cmp: false
Name: duong
Hand:2c2s7d0d
Pile:0s0h8s8h9d9c5d5s3d3h6c6h
Score:0
*Cmp: true
Name: Cmp1
Hand:2h1c7h8d
Pile:5h4h9hqdqh7s7c1hkh1s3sjs1d
Score:0
#Turn:
Turn: 24
Table: 8c0cjc
#END
```

VIII. Testing:

In the test package of the project, I include two objects that I used to test my program, one for the logic, and the other for the GUI. To test the program, I write an app and specify some methods that I want to test, then I print out the results and compare it with the expected results. This testing system is different from what I planned to use because I could not find a way to include the library for the unit test.

During the implementation progress, especially when I implement the GUI, I test the functions immediately after implementation. I also modify the objects frequently to suit my tests. And after I finished the basics of the project, I test directly on the app. However, I might not design the test carefully enough since there are some bugs that only found when I used the app. Most of them has been dealt with, although the biggest problem was the *validTrade* method is still left over.

IX. Known bugs and missing features:

Aside from the bug of the *validTrade* method, the program is known to have some other bugs. One of them is a small bug in displaying the turn of the computer player. If there are more than 1 computer player, the program will display a turn screen for the computer number 2 and 3 with the player's cards displayed upside down. However, this is a small bug and did not affect the game progress.

Another found bug is that when importing the project to other computer's IntelliJ, it sometimes automatically create an additional folder for the project, leading to the image file path become wrong.

X. Best sides and weakness:

One of the points that I consider good in the project is that I have created a good experience for the user. I believe the features that I include in the program such as to deselect a card or to save into multiple files can help the user enjoy the game better. Another good point is that I believe I have written my code in a good way to read. I used private variable and provided some convenient method to access and modify these variables, I also divide the code into different sections for more comprehensibility.

About the shortcomings of the project. First it is the validTrade method, I did not design it carefully enough, leading to the deviation from the project specification. Second is the optimalMove method, this method is

created based on my estimation. Even though it has yielded positive results in the test plays, there might be a better algorithm for this method. Another weakness of this program is that I did not dealt with the case the user attempt to provide identical name. The winner method only concerns about the name of the player, so it might yield negative results when provided similar names.

XI. Deviation from the plan:

During the first progress of the project, I matched with the planned schedule. But when I reach the optimalMove method, which I need a lot of time to figure it out, and I need to do other courses, so I slowed down a lot with the schedule. I had to request an extend, but the GUI part is really challenging. However, I came across an older project with similar topic as mine, and I had used it as an inspiration for my project. In the end, I roughly finish it by the extend deadline.

If there are anything that I can learn from this project, it is that I need to start researching sooner, since some part like the optimalMove and the GUI turns out to be even more time consuming than I expected. Another thing that I take after the project is the skill to deal with GUI. I learnt a lot about how to create a basic GUI for the user. For example, I though that I need different apps for each screen. But it turns out that I need only one app to manage all the screens, and the screens are extend from various classes.

XII. Final evaluation:

Based on the above sections, I can conclude the following about my project:

- The good parts: I believe that I have provided features to enhance the user' experiences, the code of the program is clear from my perspective.
- The downside: There are still bugs in my program, and I have deviate too much from my planned schedule.

As a final evaluation, my project is roughly at average quality. If I can start my project from the beginning, maybe I could perform more test plays to learn more about this game. I tried to design the algorithms in a general manner, but it seems that it is not always the case. One example is the number of cards on table. I design it only to store maximum of 14 cards, but it turns out that the actual number of cards does not reach that number. So maybe I could redesign methods such as *validCapture* based on assumptions like this.

XIII. Reference:

https://github.com/atreyaray/CassinoGame: This is a similar project that I used as an inspiration in the late progress of my project.