Cassino

Name : Atreya Ray
Student Number : 787530
Degree : Bachelors in Data Science
Year of Study : 1st Year
Date : 27 April 2020

General Description :

The goal of the project was to make the Cassino (deck-cassino) card game with a fully functioning graphical user interface. The rules are as follows:

- The deck is shuffled at 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. The player next to the dealer starts the game. In the next round, he/she is the dealer.
- On their turn, A player can play out one of his/her cards: it can be used either for taking cards from the table or to just put 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. A player must draw a new card from the deck after using a card so that he/she always has 4 cards in his/her hand. (When the deck runs out, everyone plays until there are no cards left in any player's hand).
- A 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 the used card.
- 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.
- Sweep: If some player gets all the cards from the table at the same time, he/she gets a 'sweep' which is written down.
- 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 the table, Diamonds-10: 16 in hand, 10 on the table, Spades-2: 15 in hand, 2 on the table)

Scores are calculated when every player runs out of cards and the last player to take cards from the table takes the remaining cards from the table.

- Every sweep grants 1 point.
- Every Ace grants 1 point.
- The player with most cards gets 2 points.

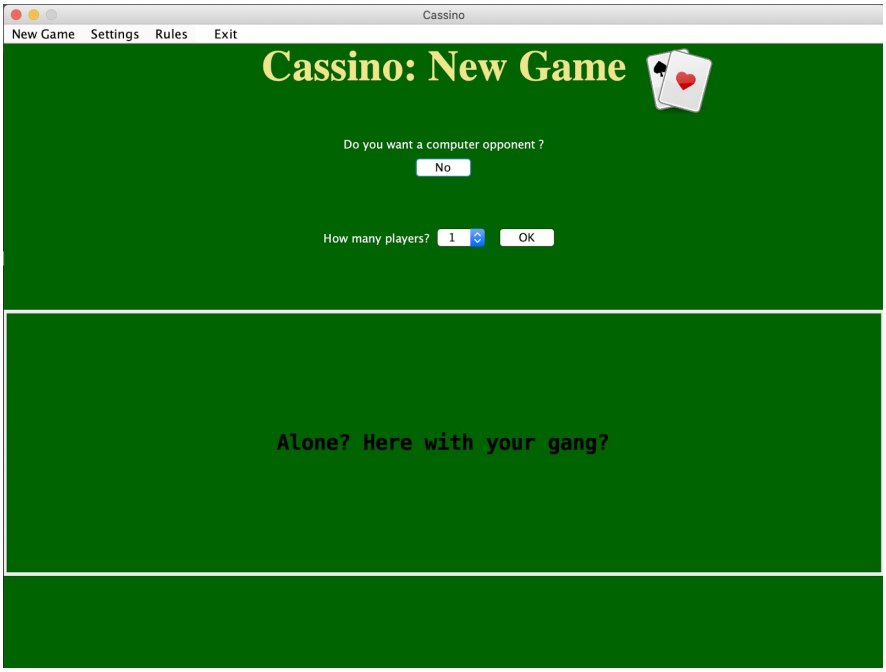
- The player with the most spades gets 1 point.
- The player with Diamonds-10 gets 2 points.
- The player with Spades-2 gets 1 point.

At any point in time, the game state can be saved onto a file and also retrieved from a file using a custom file format.

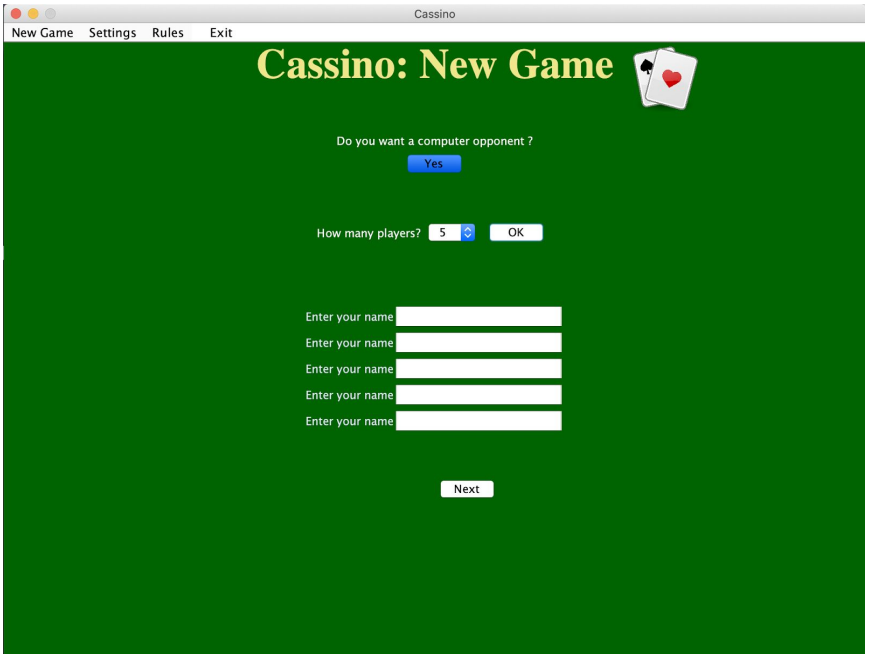
Difficulty Level: The project was completed on the demanding level with a complete GUI and a possibility to play against a simple computer-controlled opponent as well as human opponent which would use basic tactics and strategy.

User Interface :

The program can be run through a graphical user interface (GUI) which can be run through the GameGUI.scala file. Upon running, the GUI presents the first page where the user may choose a computer opponent and the number of human opponents.



After selecting whether or not to have a computer opponent with the help of the toggle button and choosing the number of human opponents with a drop down list, when the “OK” button is pressed, multiple text fields appear to allow inputs for the name of players.



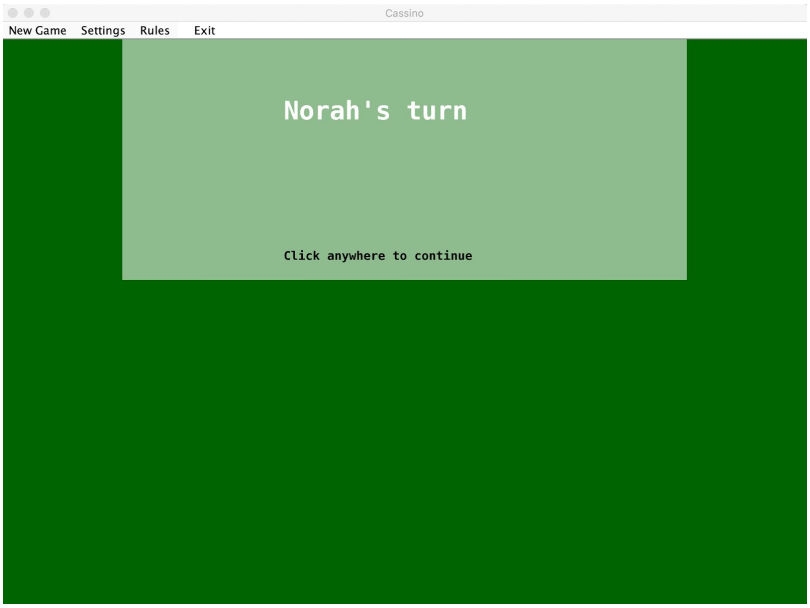
The names of all the opponents can then be entered and then the “Next” button may be pressed. After this a new page appears which contains the game window.

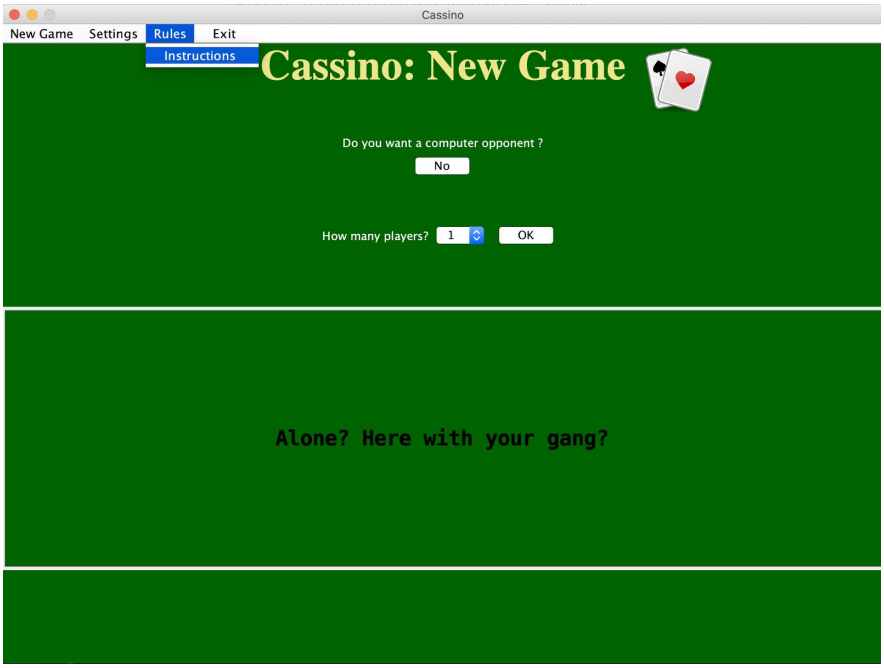
The game window is where the entirety of the game runs. There are “Capture” and “Trail” buttons on the bottom right which may be clicked to execute those respective moves. The player’s icon, name and hand are displayed on the bottom. In the centre the cards on the table, visible to all the players, are shown. The right side contains the Score panel where the score of each player is given besides their corresponding icon. On the right of the panel, there is a text box which gives some helpful tips about the gameplay.

During the computer’s turn, the GUI displays the move that the computer opponent makes. This involves having one card from the hand being visible, along with the selection to capture from the table (if any). The user may click anywhere on the window to proceed to the next turn.

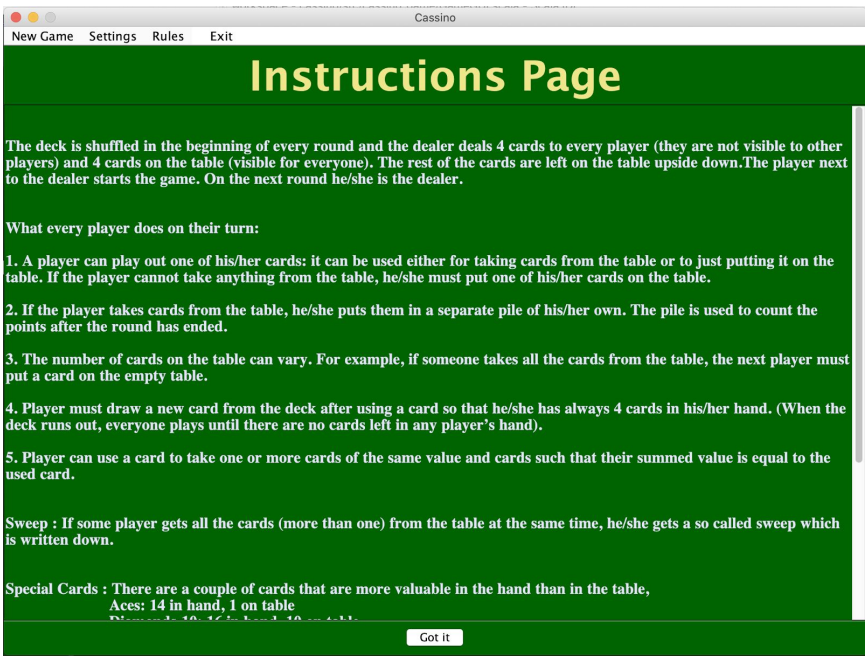
For a regular player’s turn, the player may click on any card from their own hand along with one or more cards from those available on the table to execute a capture. After selecting the required collection of cards, the “Capture” button must be clicked. An invalid capture move is not allowed and prompts a suggestion on the text box above

On the other hand a “Trail” move is always valid and can be executed by the player by first selecting the card in their hand that they want to trail and subsequently pressing the Trail button. Upon the successful execution of a “Capture” or “Trail” move, a new window pops up, informing the players of the change of turn, thereby allowing privacy and hiding the hand of individual players.

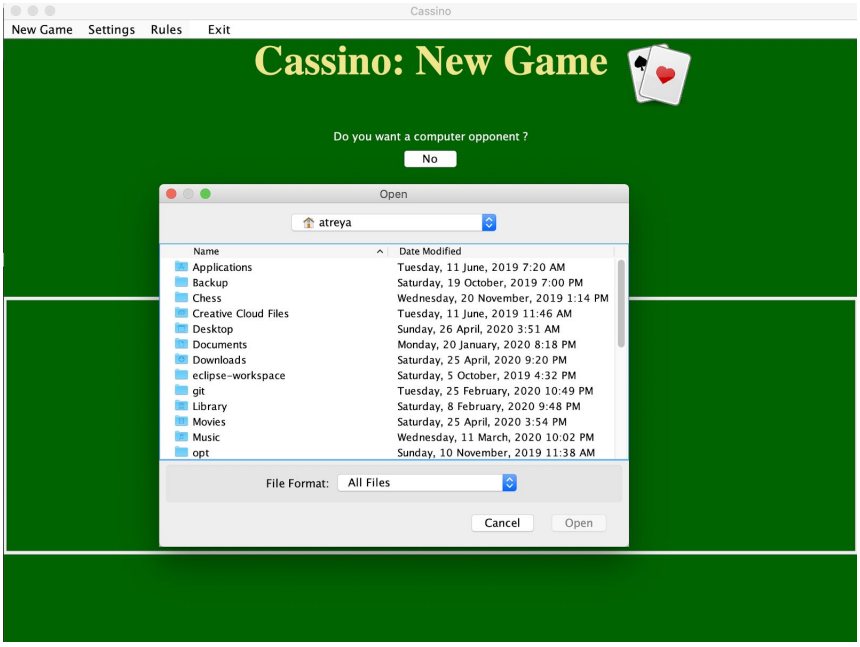




The instructions page can be accessed through a drop down in the menu bar. Clicking on it takes the user to a separate page.



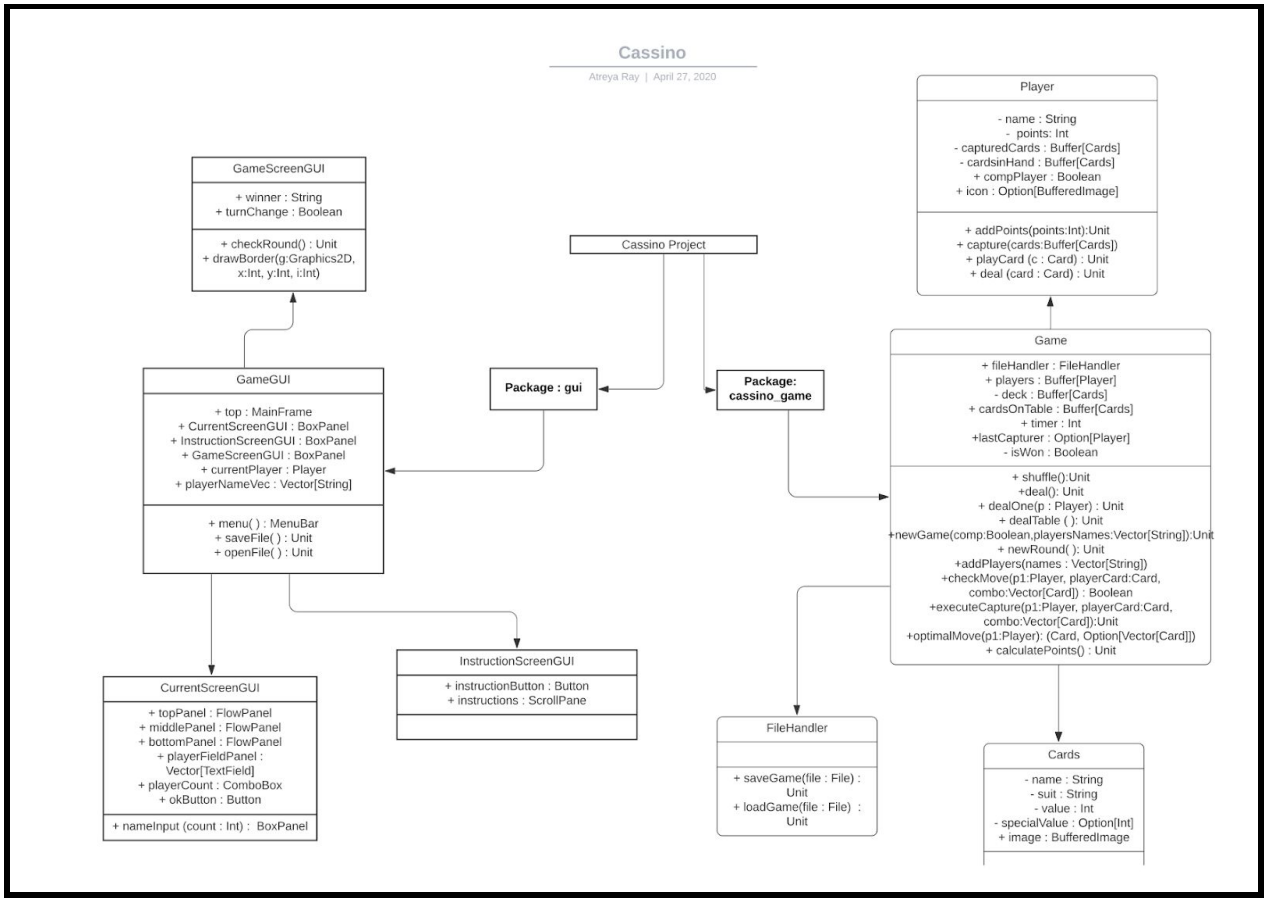
The instructions are given in text format and the user may scroll down to read the entirety of the text. After reading, the instructions, clicking on the “Got It” button will take the user back.



The data of a particular game may be saved in a .txt file and the same file may be loaded at a later time to allow continuity of the game. The “Open” and “Save” buttons inside the “Settings” dropdown open FileChoosers which allow users to do the same.

Apart from that, the “New Game” option in the menu bar creates a new game whenever clicked.

Program Structure :



Link: https://www.lucidchart.com/invitations/accept/f14256bf-79f3-4bcf-81af-8b781b7333eb

The entire program is split into two packages: ‘cassino_game’ and ‘gui’ based on the logic implemented inside each. Package ‘cassino_game’ for the most part implements the logic of the game and the necessary tools required for playing the game. On the other hand, the ‘gui’ package handles the graphical user interface and other code necessary for running the same. Due to the size of the code for the GUI, the code was allotted to a separate package in the realised class structure as compared to the preliminary plan. This slight change makes the program more readable.

Within ‘cassino_game’ the main method Game implements most of the methods required for playing the game such as shuffling cards in ‘shuffle’, dealing cards in ‘deal’ and creating new games for the GUI in ‘newGame’. There are also classes ‘Player’, ‘Card’ and ‘FileHandler’ which allow different players and cards to be a part of the game as well as allow the program to read and write to a text file. The game object contains objects of the

Player, Card and FileHandler class. However, the game contains only one FileHandler object but multiple players and cards.

The ‘gui’ package contains the main object ‘GameGUI’ which is run to play the game. It initialises and runs the game with the help of different screens which display different pages inside the game. ‘CurrentScreenGUI’ implements the page where the number of players and the player data is inputted. ‘InstructionScreenGUI’ creates the page where users may read instructions for the game. ‘GameScreenGUI’ is the page where most of the game is played and the points for different players are shown.

Algorithms :

The main algorithms in this program were in implementing the ‘checkCapture’ method which returns a boolean to signify if a given choice of capture is acceptable or not, and the ‘optimalMove’ method which generated a good move for the computer opponent to play.

For the checkCapture method, the problem was to check if it was possible to partition a set of cards into subsets which had its sum of values equal to another given value. Initial research indicated that this was a NP-hard problem and would be difficult to implement on a general level. Due to the relatively small size of the problem, I developed a workaround to this problem. The general observation were as follows :

A 4-membered set of cards may be partitioned into (order does not matter)-

1. Two 1-membered sets and one 2-membered set
2. Two 2-membered sets
3. (Four 1-membered sets are not possible in playing cards as 5 cards with the same value would be required)

Similarly, a 3-membered set of cards may be partitioned into

1. One 1-membered sets and one 2-membered set
2. One 3-membered set

A 2-membered set of cards may be partitioned into

3. Two 1-membered sets

4. One 2-membered set

Finally a 1-membered set would be a trivial problem.

Filtering out the chosen move combo by all 1-membered sets with value equal to the required value reduced all of the problems to trivial problems. For example, in the case of a set of cards {h5, d6, sj} to be matched with {dj}, filtering the set by cards with value not equal to required value 11 would reduce the problem. The new simplified set would be {h5,d6} and all that remains would be to check if the sum of values of this subset is equal to 11.

There is only one exception to this logic ie, a 4-membered set partitioned into two 2-membered sets. For this the iterator ‘combinations’ from the Vector class was used to then find distinct subsets with sum of values equal to the required value.

‘optimalMove’ class iterates over all the possible subsets of the set of cards on the table. It checks if the move is possible. If a given subset constitutes a valid move, then the combo is assigned scores according to the following criteria :

- Every Ace in the combo adds 1 point.
- Diamonds-10 gets 2 points.
- Spades-2 gets 1 point.
- Sweep adds 1 point

Based on the scores of the combos, the best scoring combo is then chosen from all the combos. If none of the combos give any points, the combo with the largest number of cards is then preferred. If no combo is possible in the context of the game, then the method moves on to trail a card. The card to be trailed is decided by choosing the card with the lowest value.

Data Structures :

Data structures/collections used in the project have been Vectors which were preferred over Buffers due to the immutability which makes it less prone to errors. Vectors have been used to store data like ‘cardsOnTable’, ‘cards’ in the Game object and ‘cardsInHand’ and ‘capturedCards’ in Player class. Other than this there are numerous instances where

collections have been used along with loops to reduce code. No user defined data structures have been used for the project.

Files :

The program primarily uses text files to read and write data from the game state. The data inside the '.txt' files are written in block beginning with '###' and ending with a blank line. The file is ended with the "EOF" string. All the cards are encoded with two letters with the first letter representing the first letter of the suit of the card and the second letter representing either the number or the letter value, ie A for Ace, J for Jack etc. All cards encoded as such are separated by a colon (:) wherever there is a list. By default, everything other than the player names and card data are in uppercase.

The first block specifies the name of the game and the time of saving for the file. This is for general information for the help of the programmer. The second block contains the 'METADATA', which contains the number of players, the cards on the table, the cards remaining, the last capturer and other essential details needed for the game to continue from a break.

The third block onwards specifies the individual players data, including the score, the hand and the cards captured. The number of these blocks depends on the number of players. Due to limitations of the GUI, the current player limit is 6.

```
###
CASSINO GAME
v1.0
Sun Apr 26 19:27:44 EEST 2020

###
METADATA
NUMBEROFPLAYERS 3
ROUNDNUMMBER 1
NOOFCARDSREMAINING 2
CARDSONTABLE d4:c6:
CARDSINDECK h9:dq:h2:cq:s4:h10:s7:c4:s9:dj:d3:sj:h8:c9:d9:c8:h3:s2:d8:ha:d7:h7:d10:hj:sa:ck:s3:c5:h4:
LASTCAPTURER Mom
CURRENTPLAYER Atreya

###
PLR
NAME Computer
SCORE 0
HAND dk:cj:s8:s10:
CAPTUREDCARDS d5:ca:s6:

###
PLR
NAME Atreya
SCORE 0
HAND d6:h6:da:c3:
CAPTUREDCARDS sk:hk:

###
PLR
NAME Alan
SCORE 0
HAND sq:hq:c7:c10:
CAPTUREDCARDS d2:c2:h5:s5:

###
EOF
```

Testing :

The program has unit testing for some methods implementing game logic. The GUI has been tested manually for alignment of different components and ease of use. There is a separate package for unit testing called ‘test’ which contains all the unit tests made. All tests were made using the scalatest class.

The ‘checkCaptureTest’ class contains several tests for different moves to be allowed or rejected. There has been extensive testing for different capture moves, with single member combos, multiple member combos, special value card combos, illegal captures along with trailing moves which are comparatively simpler in implementation. Similarly, the ‘optimalMoveTest’ contains a few tests to check if the move generated by the method is indeed legal, and a good move in the context of the game.

The gui was extensively tested manually to correct alignment of different components. Different use cases were tested to check smooth switching between windows and the working of buttons and other input fields. Color combinations were also tested to improve aesthetics of the program overall.

Known bugs and missing features :

There are some bugs and missing features which have not yet been implemented into the program. Adding corrupted files or files of different programs causes various errors in gameplay in later stages of the game. The program is not able to identify incomplete information in text files yet. Adding blank player names is possible which makes the players differentiable only by their icons in the game.

Due to screen space limitations, a maximum of only five cards can be trailed on the screen. However in most use cases this does not occur. Additionally, due to the same limitations, a maximum of six players along with a computer opponent can play the game at one time.

Missing features include the addition of a timer which was mentioned in the plan. Some other issues include a small lag between clicking and registering of the event on some computers. The program was developed on MacOS and some misalignments of the header were also noticed when run on Linux operating systems.

3 best sides and 3 weaknesses :

The best sides of the program are as follows :

1. Readability of Game Logic : For the most part, the game logic has simple class structures with intuitive variable and easy to understand variable naming. Most of the game code has been well commented and thus the more complex processes are

explained in more trivial steps. The initialisations of objects of Player and Card classes are especially succinct and cryptic.

2. Aesthetics of the GUI : A lot of time was spent on the aesthetics of the GUI to correctly place components and use appropriate fonts and color combinations. The use of player icons, other images along with pictures of cards make it quite a visual program. The intention of the program was also to increase ease of use and increase user experience quality.
3. Class Structure : Class structure in the project is based on logical processes. The compartmentalisation of objects and classes based on their role into different packages, makes the structure of the program logical and easy to follow. The distribution of code into smaller subsections has been done carefully.

The weaknesses of the project include :

1. Robustness of FileHandler Class : As mentioned above, the FileHandler class fails to handle corrupt files or files of different formats properly. Moreover incomplete game data cannot be detected by the program.
2. Magic Numbers : Even with the aesthetics of the GUI, placement of components are arbitrary in nature and have not been generalized according to some logic. As such, the positions are extremely specific and changing the design would require considerable effort.
3. Absence of Platform Invariant Design : Project runs into some design issues on other operating systems such as Windows and Linux. More testing is required on different operating systems.

Deviations from the plan, realized process and schedule :

The project was implemented in increasing intensity over each two week period. The first two week period saw the implementation of skeletal classes to form the preliminary structure of the classes for the game logic. The second two week period saw adding code to auxiliary classes such as Card and Player. The third period however coincided with the exam week and thus not much progress was made. The final time period contained the

most considerable effort with over 60 hours of work done in the final two weeks. In this time frame, the game logic was polished, the textUI was developed as a backup and then the GUI was created.

The planned schedule was a more balanced one (in terms of effort put in), but the realised schedule had an exponential increase in work during the end of the overall time period due to unforeseen difficulties in navigating through unfamiliar libraries . However the schedule of development of the game loosely followed the planned schedule. The planning schedule hugely underestimated the time taken to learn new libraries and coding styles inherited from Java but implemented in Scala.

I learnt a lot from this process. Overcoming difficulties with code without many good resources on the internet and learning how to create functioning GUIs were some of the major takeaways. Additionally, I realised I should have kept some additional time for optimization of the code and reducing redundancies.

Final Evaluation :

The overall output of the project was satisfactory and the existence of a working GUI with proper aesthetics makes it a commendable effort. The game logic can be extended easily to include more variations of the game without considerable effort. Nonetheless, it is nowhere in comparison to actual online card games and has to be made more robust in it’s file handling. Moreover smarter algorithms may be devised to make computer opponents smarter. The program can be extended to include timers for different players. The program could be improved by generalising the magic numbers used in the GUI.

If I were to start the project from the beginning, I would create a GUI before creating the textUI as a considerable effort was required to refactor the code to make it usable by a GUI. Additionally, I would be able to plan the structure of the GUI a little better to make it more logical. Finally, I would increase my efforts in the first month of the project to make time to address unforeseen difficulties.

References :

[1] : Stack OverFlow - Stack Overflow - Where Developers Learn, Share, & Build Careers

[2] : Scala Swing Documentation - Swing

[3] : Mark Lewis (YouTube Channel) - Mark Lewis

[4] : RapidTables (Color charts) - Web Colors

[5] : ColorHexa - Forest green / #228b22 hex color

[6] : American Contract Bridge League (Card images) - Back of Cards

[7] : Icons8 - Free Icons

[8] : IconFinder - https://www.iconfinder.com

[9] : Java Documentation - Overview (Java Platform SE 7)

[10] : Spigo (Sample game interface) - Spigo.co.uk - Games and Friends

References :

Files inside the zip archive include:

1. Source Code (Eclipse Project)
2. Project document
3. GUI use case screenshots
4. Examples of saved files
5. UML diagram of program structure