**Card Game**

*Student A Candidate Number: 014485*

*Student B Candidate Number: 054530*

*(Marks to be allocated 50:50)*

**Table of Contents**

[Development Log 3](#_Toc56003105)

[23/10/2020 – 12:45pm (5 hours) [*Signed: 014485, 054530]* 3](#_Toc56003106)

[25/10/2020 – 12:30pm (3 hours 30 minutes) [*Signed: 014485, 054530]* 3](#_Toc56003107)

[30/10/2020 – 12:30pm (3 hours 30 minutes) [*Signed: 014485, 054530]* 3](#_Toc56003108)

[5/11/2020 – 1:00pm (4 hours) [*Signed: 014485, 054530]* 3](#_Toc56003109)

[Design Analysis 4](#_Toc56003110)

[Test Design Analysis 6](#_Toc56003111)

# Development Log

## 23/10/2020 – 12:45pm (5 hours) [*Signed: 014485, 054530]*

Started with A as driver and B as navigator. Created *CardGameTest.java* to test methods *validateInput* and *importPack*, which will be used to handle input for number of players and input for the pack of cards respectively. Created *CardGame.java*, and added input for number of players and location of pack to load in the main method, with auxiliary methods *validateInput* and *importPack* to handle these inputs. Ensured code passed unit tests.

Swapped roles (B as driver, A as navigator), then added test methods to *CardGameTest.*java for *dealCards*, which will deal cards to players, and *countFrequencies*, which will check if pack of cards is guaranteed to be winnable. Developed methods *dealCards* and *countFrequencies*, and ensured code for these methods passed unit tests. Created *Card.java* with setter and getter methods for card values and holders. Created *CardDeck.java* with methods for deck transactions.

## 25/10/2020 – 12:30pm (3 hours 30 minutes) [*Signed: 014485, 054530]*

Started with B as driver and A as navigator. Created *PlayerTest.java* to test a threaded class which will handle actions of the player, with a method *play* which will continually draw and discard cards until a winner is found. Created threaded class *Player.java* with method *play*, and ensured the code passed the unit test.

Swapped roles (A as driver, B as navigator). Added a test method in *PlayerTest.java* for a new method *isWinner*, which will perform checks for a win condition and declare the winner of the game. Created a new method *isWinner* in *Player.java*, and ensured the code passed the unit test.

## 30/10/2020 – 12:30pm (3 hours 30 minutes) [*Signed: 014485, 054530]*

Started with B as driver and A as navigator. Added test methods in *PlayerTest.java* for *strategy*, which will help the player decide to do with a newly drawn card depending on cards in their hand, and auxiliary methods *keep*, *remove*, and *discard* to manipulate the cards depending on the scenario.Created the method *strategy* and its auxiliary methods *keep*, *remove*, and *discard*.

Swapped roles (A as driver, B as navigator). Added test methods in *PlayerTest.java* for *createFile*, which will create an output file for each player’s actions, *writeToFile*, which will write to these files when a game action has occurred, and *viewArray*, which will display the cards in the hand or deck of a player. Created methods *createFile*, *writeToFile*, and *viewArray* in *Player.java*, ensuring that the code passed unit tests.

## 5/11/2020 – 1:00pm (4 hours) [*Signed: 014485, 054530]*

# Design Analysis

At the start of the game, the program asks for the user to input the number of players in the *main* method. It uses the *validateInput* method to check if an integer greater than two is inputted, and then it iterates this number of times to create a player object and a deck object for that player. Creating deck objects for each player makes it easier to decompose problems later when it comes to drawing and discarding cards. In the main method, it also asks the user to input the file name for the pack of cards to be used. When loading the pack of cards in the *importPack* method, the program checks that the values in the text file are all positive integers, and that there are enough cards for the number of players. The pack of cards is implemented as an ArrayList, because this allows us to dynamically add and remove cards from the pack as cards are transferred to players’ decks.

In *dealCards*, the method keeps track of the size of players’ hands through the getter method *getHandSize* to ensure that it continues dealing directly to their hands until every player has four cards. After that, it sequentially adds cards to players’ decks with the *addToDeck* method.

The *genHashMap* method is used to generate a HashMap of key-value pairs of the cards in play, where the key represents the value of the card, and the value represents the number of these cards in the game. This is used in the *countFrequencies* method, which iterates through the players, and analyses their chances of winning based on the number of cards with their preferred value. If the player has fewer than four cards of their preferred value, then a winner isn’t guaranteed from the game, and it may stagnate.

We created a class *Player.java* to handle the decisions of each player in the game, as this helps to segregate decisions made by the game (like a referee managing the game), and the players.

To draw a card from the player’s deck, the method *draw* takes the last card from the top of the deck on their left, removes it from the deck, and then discards a card to the bottom of the deck on their right. After every draw, the player’s hand is printed using the method *viewArray*, which looks at the player object and iterates over the hand to check each value.

Immediately drawing a new card, the method *strategy* looks at the player’s hand of cards (excluding the card which has just been drawn), and randomly discards one of the cards which is not their preferred value.

Auxiliary functions *keep*, *remove*, and *discard* are used for the player to perform the appropriate actions according to the game scenario. The *keep* method takes the card as an argument, then adds it to the player object’s hand. The *remove* method takes the card as an argument, then removes it from the player object’s hand. The *discard* method takes the card as an argument, then discards it to the deck of the next player’s deck.

To track the activities of players as actions in the game occur, we made a method *createFile*, which creates an output .txt file for every player in the game in the format *player[NUMBER]\_output.txt*. The additional method *writeToFile* is used to write strings into the text file on new lines after an action in the game is made.

A method *isWinner* is used to check whether a player has the appropriate cards in their hand to win the game. As there may be multiple players with a complete hand at the same time due to the threading, the method checks if a winner has been found before overwriting the winning player’s integer. If a winner has been found, then it stops players from continuing to draw and discard cards in the *run* method.

# Test Design Analysis

For our testing, we decided to use JUnit 5.4.2, as this has several advantages over using JUnit 4.x. It enables you to import only what is necessary, which is more efficient on storage. In addition, it allows multiple test runners to work simultaneously, which means testing can be performed more quickly.

We took a test-driven development approach, and we sought to abide by best practices for testing, as outlined by JUnit themselves (<https://junit.org/junit4/faq.html#best_1>).

We created test classes for major classes, and created test methods to test each method in these classes where necessary. Tests were written before the code, which helped us to get an idea of how we should write the code to be concise and fit for purpose. Test-first programming was practised, only writing new code when an automated test was failing.

A pragmatic approach was taken in that tests were not created for everything; we simply tested everything which could reasonably break. Where possible, we adapted our design so that testing would be possible. For example, when seeking to take inputs from the user, we took the input from the main method, and then processed them in auxiliary methods which took the input as an argument. This ensured that we could test different inputs, and check whether they were being validated correctly in the auxiliary methods.

The benefit of our test-driven approach was that the code developed was less likely to include bugs, as they would have been exposed through the unit tests created beforehand. We had very few problems with the functionality of our code, which left us with more time to optimise our code for efficiency.