# ECM2414 Software Development

## CA - Report

Split: 50:50

Student Number: 690029623
Candidate Number: 025474

Student Number: 690062247
Candidate Number: 074486

| Date | Start | Stop | Time | 025474 | 074486 |
|---|---|---|---|---|---|
| 15/10/2020 | 12:45:00 PM | 2:48:00 PM | 2:03:00 | Created and filled out a to do list document and this development log. 025474 | Created and filled out a document summarizing the CA document. 074486 |
| 16/10/2020 | 1:07:00 PM | 2:41:00 PM | 1:34:00 | Created static method in cardGame for inputting and validating a pack of cards. Drove first half then navigated, 025474. | Created static method in cardGame for the user to input the number of player's playing in the game. Navigated first half then drove, 074486. |
| 21/10/2020 | 2:39:00 PM | 5:14:00 PM | 2:35:00 | Created and filled Card class, created CardPile Class which contains an Array of card objects. Both attempted to add JUnit tests but couldn't get it to work. Drove first half then navigated for player class, 025474. | Created Player class which includes a player id (1,2,3) and their current hand as a CardPile object. Both attempted to add JUnit tests but couldn't get it to work. Navigated first half then drove for the Player class, 074486. |
| 27/10/2020 | 1:30:00 PM | 4:27:00 PM | 2:57:00 | Changed all uses of Arrays to ArrayLists. Drove for these conversions and then navigated, 025474. | Created a method to distribute cards between players in a round-robin fashion using an ArrayList of card objects. Navigated then drove to create above method, 074486. |
| 28/10/2020 | 10:04:00 AM | 12:16:00 PM | 2:12:00 | Changed Player Class so it implements Runnable and each player can run on separate thread. Navigated then drove for this edit, 025474. | Edited the method created yesterday by allowing it to also be used to create the decks that are going to be drawn from. Drove to edit this method then navigated, 074486. |
| 30/10/2020 | 2:06:00 PM | 3:29:00 PM | 1:23:00 | Added methods in the Player class to draw and discard cards from the correct decks. Added JUnit tests for these methods. Navigated then drove after to make these methods, 025474. | Added a method in the class Player to check if the player's current set of cards is a winning hand or not. Added JUnit tests for these methods. Drove first to do this method then navigated, 074486. |
| 31/10/2020 | 10:39:00 AM | 12:26:00 PM | 1:47:00 | Edited draw and discard methods to write to their output files. Added method to delete all output text files. Navigated, realised we'd need this method then drove after, 025474. | Added method to create player's output file and write initial hand to it. Added a method that writes their current hand to the file. Drove first to add these then navigated, 074486. |
| 03/11/2020 | 1:01:00 PM | 3:34:00 PM | 2:33:00 | Fixed issue where the 'player x wins' would be displayed from each thread instead of once. Drove to fix bug then navigated, 025474. | Both added more comments to the code. Added a method for writing the cards of the decks that don't belong to a player to a text file. Navigated then drove to add method, 074486. |
| 05/11/2020 | 1:00:00 PM | 4:26:00 PM | 3:26:00 | Both refined, commented and added more JUnit tests to each of our classes. 025474. | Both helped fixing a bug where the number of cards left in a deck was not four. This was caused by a player having an extra go after another player had won, this is now fixed. 074486. |
| 06/11/2020 | 1:03:00 PM | 3:09:00 PM | 2:06:00 | Added more comments throughout the program and general tidy up. 025474 | Added comments and arranged the methods in each file to have a more logical layout and order. 074486 |

# Production Code

Firstly, we have our executable file ‘CardGame.java’ as detailed in the specification, this class is responsible for the main running of the Card Game program. This class has various methods and a main method found at the bottom of the file.

The first method that’s run in the main method is ‘deleteTextFiles()’, this method always runs once at the start of running the program. It is responsible for deleting all output text files generated from the last time the program was run. It does this by iterating through all files from the main folder and deleting those that end in ‘_output.txt’. While it would be possible to just overwrite these existing files each time the program was running. This could cause confusion when the program is run with 4 players for example, generating 8 text files. Then the program is run with 3 players, only overwriting the first 6 files. This would leave ‘player4_output.txt’ and ‘deck4_output.txt’ still in the main directory but it was not part of the last game.

Once the text output files are deleted, the user is prompted for the number of players to play in the next game via the ‘inputNumPlayers()’ method. Once a successful input has been received, the method returns the value and name of the pack file is asked using ‘inputPackFileName()’. We decided to check the validity of the contents of the file within this method allowing the user to possibly make changes to a pack file while the program is running rather than entering an invalid pack file and the program having to exit later down the line due to an invalid game setup. In addition, we thought it would be suitable to check that it is possible for a player to win with the pack file and reject a pack file if it is not. Saving the user from having to halt execution of the program manually during an infinite game.

The valid pack file is then read again using another method ‘fileToDeck()’, this generates a CardDeck object containing an ArrayList of all cards from the pack represented as a Card object. We decided to use ArrayLists throughout our solution as they were much easier and simpler to work with, especially when adding and removing elements from an ArrayList compared to an Array. We distributed the cards to the players then the decks using the same method ‘genderateDecks()’ ran twice. First the player’s hands are generated, then the decks are generated. We originally had separate methods to distribute cards to the players and the decks. However, these methods were very similar so we were able to use the same method for both by removing each card from the ArrayList of cards when it had been distributed.

We then create and start a new thread for each player and add the thread to an ArrayList of all player threads. This is done so we can halt execution until all player threads have finished and thus the game has finished. The final state of each deck is then written to their respective output files and lastly the winning player is printed to the terminal.

Secondly, ‘Player.java’ contains a Player class that represents an individual player playing the game. This class implements Runnable so a Runnable object can be created for each player thread created and started in ‘CardGame.java’. This class contains the methods used while the game is being played to allow each player thread to perform actions such as drawing a card simultaneously. At the start of the game, the player output files are created and the player’s initial hand is written to the first line using ‘writeInitialHand()’. A while loop is then entered which will only be excited when a player has won the game. The winning player notifies the other player threads by changing the state of the volatile static flag variable named ‘stop’ which then causes each thread to exit the while loop. We decided to check each player for a winning hand at the start of this loop so if a player starts with a winning hand, the program will prevent the player from taking a turn and the game will end. In addition,

we add the player's ID to ArrayList so in the event that more than one player has a winning hand in the same round, the program will automatically assign the player with the lowest ID as the winner.
Each player will first draw a card from the deck to their left. We represented the decks as a public ArrayList of CardDeck objects, this allowed for each player thread to be able to read and edit the decks and made it easier to select which deck a player needed to draw from and discard to. For example, the index of the deck within the ArrayList of decks the player needs to draw from is always the player's ID (1,2,3…) subtracted by 1. Once cards have been drawn, each player now has 5 cards and needs to discard one. Each player will then keep picking one of their cards randomly until one is picked that is not equal to their ID, this card is then discarded to the correct deck. Although it may have been a better game strategy to prefer to keep cards that are equal to other player's IDs, we chose to do it this way as it is stated in the specification that the game strategy is not optimal.

Due to the fact that the order of which threads run cannot easily be controlled, we needed a way of undoing some of the player's last go when a player had won. Without this, sometimes a player would have an additional go before a player had announced they had won resulting in decks that did not contain 4 cards and players that would draw and discard more times that the winning player had. The method 'undoLastRound()' will return the player's discard card back to their hand, put back their drawn card to the correct deck and delete the last 3 lines of their output file which detailed what the player did in the round to be deleted. This is only done if the player has not won the game and player has had an additional go. To finish the round, each player then appends their current hand to their output text files.
The file 'Card.java' contains a class that represents an individual card within the program. This is a very simple class with each Card object having an attribute 'number' which is the value of that card. There are setter and getter methods to get this number and finally a constructor to create a Card object with the one parameter being the number of the card.
Finally, the 'CardDeck.java' file contains a class used to represent a deck of cards, each card being it's own Card object. This CardDeck object is used throughout the program in various different kinds of decks. For example, the entire pack of cards is put into a single CardDeck object before the cards are distributed into small CardDeck objects for the players and the decks. This class contains various methods to perform actions and retrieve information about the card deck.
Firstly, we have methods to add and remove specific cards from the card deck which just adds and removes the card object from the ArrayList respectively. Next, there is a 'getCard()' method which will return the Card object at a given index and also remove it from the deck. This is used in actions such as when the cards are being distributed at the start of the game in a round-robin fashion, when a player draws a card from a deck and when a player picks a card to discard from their hand. We also have a 'viewCard()' method which is similar to 'getCard()' but it doesn't remove the card from the deck. This is used when a player is picking cards at random to be discarded but it can only actually remove it once a card is picked that is not equal to their ID.
The method 'numberOfCards()' returns the number of cards in the deck, this value can be a sign that a player needs to undo their last go when the deck they are drawing from contains 3 cards rather than 4 at the end of the game when a player has won. Lastly, we have a 'displayCards()' method which returns each card value with a space between them as a StringBuffer. This was heavily used during testing and debugging. In addition, it is used each time the values of each card within a deck needs to be written to an output file such as the 'deckx_output.txt'.

# Testing

We decided to use the JUnit 4.12 framework along with Hamcrest Core 1.3 to test all our classes used in our solution. We found our solution would use on average 6.5MB of memory when running the cards.jar file inside the terminal; these program runs involved a winning valid test pack that took on average 15 rounds for a player to win and the program to exit.

## Card Class

As the Card class is a very short and simple class, the testing for this class is limited with only one test being done. The values of each card have already been verified prior to a Card object being created therefore this does not need to be checked in either the Card class or the tests for this class. For this test, a Card object is created in the setup with a value of 1. The test asserts that the value that this Card object is indeed equal to 1.

## CardDeck Class

The first test in this class is testing that the method of creating a new deck and adding a card to it is working. A new CardDeck object is created and a new card is added to it with the value 1. It is then tested by asserting that the length of the CardDeck object is equal to 1, and by asserting that the value of the card in the first index position has a value of 1. This test provides confirmation that the addCard method used to add cards to players hands and to draw/discard piles is functioning as is intended.
The second test is testing the removeCard method, used to remove cards from player's hands and the draw/discard piles. A new CardDeck object is created that contains 4 card objects of values 1, 2, 3 and 4, the card of value 1 is then removed from the CardDeck object using the removeCard function. The length of the CardDeck object is then asserted to be equal to 3 as there should now only be 3 card objects within it, the remaining cards in the CardDeck are then converted into a string, before being then compared to a string of what the results are intended to be of "2 3 4 " using assertEquals as the card of value 1 should have been removed..
The third test is testing the getCard method, used to return the card object from the deck at the index given and remove that card object from the deck ArrayList. The test is done by creating a new CardDeck object and populating it with 4 card objects of values 1-4 before using the getCard method on each of their index positions. This should result in an empty CardDeck object which is tested by using assertEquals on the length of the object with the expected length of 0.
The fourth test is testing the viewCard method, used to view a single card in a player's hand or a deck. The test is done by creating a new CardDeck object with values 1-4 and then creating another CardDeck used to check. The checking card deck is then populated using the addCard method in conjunction with the viewCard method to retrieve the values of the first deck and add them to the checking deck. This should result in 2 identical CardDeck objects both of length 4, containing only card objects of values 1, 2, 3 and 4 which is then checked by using assertEquals statements on both decks for their lengths and values within.
The fifth test is testing the displayCard method, used to display the cards that a player or deck has in a readable format. The test is done by creating a new CardDeck object and adding card objects of value 1-4 into it. The displayCard method is then used with toString to convert the contents of the CardDeck into a string which is then compared to the expected result of "1 2 3 4 " using assertEquals.
The final test is testing the numberOfCards method, used to return the number of cards in a CardDeck. The test is done by creating a new CardDeck with 4 card objects in it and then using

numberOfCards to find how many cards are in the CardDeck before using an assertEquals to compare it to the expected value of 4.

## CardGame Class

These tests follow the methods created in Card Game chronologically. Firstly, we test the method that takes in a user input from the terminal and verifies it as a valid number of players. A valid number of players for this game was decided to be an integer greater than 1. During setup, we simulated a user input via terminal using the SetIn method of System, this allowed us to control exactly what this method was going to check as a valid number of players. The output of this method is then assigned to a variable and this is checked within the test method against the expected value. After the test is complete, the original System state is restored.
The second test is similar to the first in the way that a user input is simulated and verified against the expected result. This time it is the method for prompting the user to enter the file name of the pack they intend to use for the next game. We decided to use a dedicated test pack for tests that required a pack file named ‘three.txt’. If the pack is valid, the method will return the name of the file and this is asserted to that same name within the test. Therefore, we know that the method is correctly detecting a valid pack.
The third test is checking the method that distributes the deck of cards to both the players and decks in two asynchronous calls of the same method. In order for this to work, the method must be removing the cards so the players and decks don’t get handed identical hands. To test this, a deck of 12 cards is created and the method is called to divide these cards into groups of 3 in a round robin fashion. To ensure this method is working, the number of decks created needs to be 3 and the number of cards in the original deck needs to be 0. In addition, the round-robin distribution is checked against a pre-calculated string that concatenates each deck created ensuring the cards were distributed as expected.
The last test for this class is testing the writeEndDeck method, this method outputs the card values within each deck that are not assigned to a player to their individual text files at the end of the game. To test this, we created a new deck of 4 cards and added it to an ArrayList of card decks. The method is then fetched using Java Reflection as it is a private method. The method is then invoked and the contents of the file that is created is checked against a string that we expect the first line of the file to be.

## File To Deck Method

This method belongs in the CardGame class however this particular test for this method did not work as expected when it was located in the same class as the other tests for methods in CardGame. Therefore, it has been moved to its own class and file. The method is responsible for converting the name of a pack text file to a deck of card objects consisting of cards from that pack. The method is called in the @Before method and the result once converted to a string is checked against an expected string with each card displayed with a space between them.

## Player Class

**stopGame:** Method for changing the flag to tell all player threads to stop running. Tested by using Player.stopGame() and then using an assertTrue statement on the value of Player.stop.

**getPlayerID:** Tested by creating a new player with the ID of 1 and then using getPlayerID to get this value and compare it to the expected result of 1 using an assertEquals statement.
**isWinningHandTrue:** Method for checking if a player's hand is a winning hand. Creates a player with a winning hand as their hand before invoking isWinningHand on the player which should return true and is confirmed using an assertTrue statement.
**isWinningHandFalse:** Method for checking if a player's hand is a winning hand. Creates a player with a non-winning hand as their hand before invoking isWinningHand on the player which should return false and is confirmed using an assertTrue statement.
**drawCard:** Method for drawing a card from the top of the deck to the player's left. Creates a player with a 4 card hand and a deck to draw from before invoking drawCard on this player. This player should now have a hand with 5 cards which is tested using an assertEquals statement with the value 5 and the player's hand length as parameters.
**discardCard:** Method for discarding a random card that is not of the player's preference to the bottom of the deck to the player's right. Creates a new player with 5 cards and a deck to discard to and then invokes the discardCard method on that player meaning that they should now have 4 preferred cards and the discard deck should have 1. This is tested with a series of assertEquals statements, the first to confirm the player now has only 4 cards, the second to confirm the player discarded a non preferred card and the third to confirm that the discard deck now has the discarded card.
**writeInitalHand:** Method for writing the player's current hand to their output text file. Creates a new player with 4 cards and then invokes the writeInitialHand method on it before reading in the output file created and using the first line as a String variable which is then compared to the expected first line of “player 1 initial hand 1 2 3 4 “ using an assertEquals statement.
**writeCurrentHand:** Method for writing the player's current hand to their output text file. Creates a new player with 4 cards and invokes the writeCurrentHand method on the player before reading in the output file created and using the first line as a String variable which is then compared to the expected first line of “player 1 current hand is 1 2 3 4 “ using an assertEquals statement.

## Test Suite and Runner

Our test suite consists of all of the above tests contained within 5 individual java files within the test folder. This file uses JUnit4’s runner and runners class to create a suite that can be all run at once. To run this test suite during development, we have a TestRunner class in the same folder. The main method of this class deletes all text files that end in ‘output.txt’. This is done so the text files created during the main running of the game don’t interfere or conflict with the text files generated within our tests. The same is done at the start of the main running of the program to clear either the output files of the testing or of the previous running of the game. After these deletions, the tests are then run and the details of any failed tests are displayed in the terminal. These tests can also be run via the TestSuite java file via the terminal command found in README.md.