**Software Development ECM2414 – The Report**

William See 710019037/ 234799

Jude Wallace 710016884/ 253672

**Cover Page**

Final Mark Allocation

William See **40:60** Jude Wallace

**Development Log**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Date | Time | Duration | Driver | Observer | Signed |
| 01/11/22 | **13:00** | **1H 10mins** | **n/a** | **n/a** | 253672 234799 |
| 6/11/22 | **2:30** | **1H** | **Jude** | **William** | 253672 234799 |
| 7/11/22 | **11:45** | **1H 45mins** | **Jude** | **William** | 253672 234799 |
| 8/11/22 | **13:00** | **2H** | **William** | **Jude** | 253672 234799 |
| 10/11/22 | **11:30** | **1H 30mins** | **William** | **Jude** | 253672 234799 |
| 11/11/22 | **10:30** | **1H 30mins** | **Jude** | **William** | 253672 234799 |
| 13/11/22 | **2:00** | **1H 30mins** | **Jude** | **William** | 253672 234799 |
| 14/11/22 | **12:00** | **45mins** | **Jude** | **William** | 253672 234799 |
| 15/11/22 | **2:00** | **1H 30mins** | **William** | **Jude** | 253672 234799 |
| 16/11/22 | **11:40** | **1H 30mins** | **William** | **Jude** | 253672 234799 |
| 18/11/22 | **12:30** | **2H** | **Jude** | **William** | 253672 234799 |
| 18/11/22 | **17:30** | **1H** | **Jude** | **William** | 253672 234799 |
| 19/11/22 | **10:30** | **0H 30mins** | **William** | **Jude** | 253672 234799 |
| 19/11/22 | **14:15** | **2H** | **Jude** | **William** | 253672 234799 |
| 20/11/22 | **12:30** | **1H 30mins** | **Jude** | **William** | 253672 234799 |
| 22/11/22 | **14:30** | **1H** | **Jude** | **William** | 253672 234799 |
| 23/11/22 | **10:00** | **3H** | **Jude** | **William** | 253672 234799 |

**Documentation detailing our design Choice with respect to our Production Code ( Max 2 page )**

**Java Version: Java SE 17 / JDK 17**

**Methodology Followed: Scrum Methodology**

As a pair, we decided to go with the Scrum methodology as we deemed it to be the most applicable. Our regular coding meetups allowed us to exercise this methodology to full effect; with regular sprint planning meetings to highlight short-term tasks.

**Code logical flow**

For our design we decided on 5 classes, one being a public inner class deck. Our CardGame class is where we have implemented the user inputs. Once the user has entered the correct inputs the main method creates an object of the CardDeck class, where an array list is set with all the cards read from the pack. After this is completed, each player has a player object created followed then by the disrtibuteCard() method. The distributeCard() method is responsible for making a corresponding Card and Deck object for each player. Once the method has initialized these for every player It proceeds to deal the cards stored in the CardDeck mainDeck array list to the player's hand in a round-robin fashion, using a for loop.

Once all the initial setup has happened a thread is created in the main method of CardGame for each player of the game. The threads then run the run() method in the player class. Within this method is where the game strategy is contained. The start of the logic within this method is, each thread creates a playerN\_output.txt file and writes to it the initial hand of the player. It then enters a while loop until a player has declared a win. The first statement in the while loop checks if a player has set a member value of the player class to true, if so, it'll skip to ending the thread and complete the exciting sequence for that thread. If not, the game enters the second statement which checks if their current hand is a winning hand, with the use of the checkWin() method. If it has a winning hand it'll proceed to change the member boolean playerWon to true and write the relevant ending messages into the correct files using appenedToFile() in the player class.

The final else block contains the atomic action of picking up a card, comparing it to the hand, changing a card within the hand and then discarding it to the deck to the right. Meaning no player can have no more than 4 cards nor any less than 4 at a given time. The picking of the card is implemented by a method takeCard() within the players' deck object. This class pops the head of the deck array list and returns the card it was. The card returned from takeCard() is parsed to comparePickedCard(), where the card comparison to the player's hand occurs. The card swapping is implemented by using the playersCards.set(X,Y) depending on the card case discovered within the comparePickedCard(). Using this also allows us to maintain the size of the array list which means it is an atomic action. The returned card from comparePickedCard() is the card to be dropped so is added to the deck array list of the player to the right of the player of the thread. Once this action is completed the relevant player files are updated with the messages describing the events that just happened. The loop then repeats from the start, until a winner can be established.

**Style of Threading:**

We have used implements runnable, overextends runnable, because using the runnable interface it allows our classes to extend other base classes if needed in our design.

**Notifying the other players/threads of a win:**

For this problem, we have used a member value in the player class. The member value is set by a synchronised method checkWin() once a player has won. The member boolean value playerWon will be set to true in this event and whoWon with the player's name and number of who won. Using the synchronised method means no 2 players can declare a win at the same time. We have implemented this by before every player picks a card they check if they have a winning hand and if the member value has been set and act accordingly if it is or not.

**Inner class 'Deck':**

We decided to create an inner class named 'Deck' within the class CardDeck as we use methods from the outer, such as dealCard(), to set up the decks, which is not possible with a static inner class. We also make it a public inner as each player makes their own inner class 'Deck' object which has their specific deck stored in a volatile array list.

**Use of array List for player hands and decks**

We have used several array lists as it allows for them to grow and shrink throughout card drawing and discarding – something an array would not allow.

**Changing of cards:**

For the card-changing logic, It is implemented in comparePickedCard() with the aid of changeRandomCard() which uses using Math.Random(). Using this allows us to change cards in random positions in all game cases. The card changing follows the logic in comparePickedCard() specified by the specification and holds onto preferred denominations and uses the changeRandomCard() to change the non-preferred denominations randomly depending on the current cards the player has. However, once a player gets a value of their preferred denomination the game will only drop this card in the case [X,X,X,playernumber] where the drawn card is X, so the player number would get discarded to the deck on the right in this case only.

**Keeping inner class 'deck' thread-safe**

Within our design, the only thread-safe class is the deck class. This is because this is the class that contains the specific deck of the players which in runtime is accessed by multiple threads to add and remove elements from the deck array list. To create the array list thread-safe we have used the keyword volatile, which means the JVM and the compiler store the deck array list in the main memory. Every time the JVM reads or writes the array list it will do so from main memory instead of the CPU cache would potentially not be up to date at a time of a read and write. Using it ensures the array list is visible to the thread and will read and write to main memory. Furthermore, we have coupled this up with synchronised methods takeCard() and addCardToDeck(). Within these blocks takeCard(), removes a card from the head of the list and addCardToDeck() adding a card to the tail. So only one thread can call these methods at a given time.

**Empty player deck**

Our solution for this comes within the run() method within the player class. Before any card is drawn by a thread an if statement checks if the deck related to the thread is empty or has cards to draw. If the deck has a card to draw the game will continue with the specified logic. However, if the deck is empty the thread will keep checking the deck until a card can be picked up.

**Assumptions:** The files created after the game are deleted before a new game is started.

**Known Performance issues:**

**Documentation Detailing our design choice with respect to our Tests ( Max 3 Pages )**

**Junit framework used: Junit 4.13.2**

For our tests, we have defined 80+% to be good coverage. At this range, we can confirm all the key methods work as intended and return expected outputs, without going overkill on the testing.

**CardDeck = 91.3% Coverage**

**Card = 86.7% Coverage**

**Player = 27.6% Coverage**

**CardGame = 0% (Explained in report reasoning)**

Due to the nature of the run() and main() methods we have not created tests specifically for these methods. This is because the run() and main() method is a series of method calls, which have been thoroughly tested in their respective test classes. This causes the coverage for the Player and CardGame class to be significantly lower than our desired 80%, but we are confident through manual testing and unit testing the called methods within the respective classes they work as required.

In our design, the implementation of the validity of the user inputs is dealt with mainly within the CardDeck class. Within the getPackFileAndLoadIt () method in the CardGame class it calls the setCardDeck() method. This method has been tested within the CardDeckTest class, to check for invalid inputs, explained at the end of this report. Within the getPackFileAndLoadIt () there is a while loop which does a simple comparison for if the array created by the setCardDeck() is correct for the game by asserting its length. Due to the simple implementation in the getPackFileAndLoadIt() method, we have decided not to test it directly as with manual testing and the use of other unit tests we can prove the robustness of the code without having more unit tests repeat the same tests at different levels in the code. The same reasoning applies to the getNumberOfPlayers() within the CardGame class. This method also takes in a user's input in the form of an integer. When the user enters a value, we get it using the built-in method Integer.valueOf(). This throws a NumberFormatException if anything other than an Integer is included. This is the only computation within this method and with the certainty of how the Integer.valueOf() method works and confidence of its application, from the Javadoc's description, we have decided to not unit test this class however have done extensive tests when compiling and running the code that nothing, but an integer will allow the user to proceed to the game.

A key set of test methods within our tests are designed on picking up the card and comparing it with the current hand and deciding which card to drop. These tests depicted how are comparePickedCard() would handle all potential cases of the game strategy. Due to this, for this part of our solution, we used a test-driven development to make sure the game strategy was as specified. The tests written consist of:

**testChangeRandomCardNeverChangePrefferedDemominations():**

This tests a random card within the hand is changed at random, however, if it's the player's preferred denomination the card will never get changed. The test method includes a loop where it tests 100 times by changing a random card and asserts, the players preferred denomination is never changed but there is never a stagnant non-preferred card. To make this test run we have implemented the Math.randon() within the changeRandomCard() which picks a random index and checks if the card is a player denomination, if so, a new index is picked and changed.

testComparePickedCardThreeCardsNoChanges():

This tests the case where the player's hand is [X,X,X,N], Z the tests assert the result of the method is [X,X,X,N], Z. Our design for this and the following mentioned tests are as followed. Once the comparePickedCard() is called a HashMap of the player's cards is created, which contains the key as the card number and the value of how many times the card appears. Once the HashMap is set the logic to solve the problem can then continue. The logic for this test is to check the HashMap contains a value greater or equal to three which we used the built In .contains(int n) method and then check if the drawn card Z matches the most common card. If it's not it checks if the card with count one matches. In this case, the card is the player number, so comparePickedCard() returns the drawn card as it doesn't advance the game, nor can it be changed for the player number as is a preferred denomination.

**testComparePickedCardThreeCardsAndDrawnCardPlayerNumber():**

For this test the case [X,X,X,N], N it tested to assert the result [X,X,N,N], X. The design for this situation is like the above problem. However, within comparePickedCard() we check for the case [X,X,X,N] if the card drawn is N it then calls the changeRandomCard() to replace an X as this is overwritten by N, the player number, being drawn. This test also checks our logical design that players preferred cards are always kept and added regardless of the grouping of non-preferred denominations.

The following tests are designed to assert the strategy is correct when there is a pair of cards in the player's hand

**testComparePickedCardTwoPairsAndDrawnNoMatches():**

This tests for when there is a pair of pairs of cards [X,X,Y,Y]. This test asserts that if a non-preferred denomination is drawn as Z, the result of the comparePickedCard() returns Z as keeping these cards allows the player to progress in the game. The use of the HashMap described above is also used for the design to return the correct card. We have designed the for loop to check over the player's cards and check the count. If the count is two in the HashMap the next card is checked. If all the cards have a count of two and are of non-preferred denomination, we return the card Z.

**testComparePickedCardTwoPairs():**

This case is for when we have a hand with [X,X,Y,Y],Y. Where Y is a drawn card matching a value in the current hand. This test checks that an X is returned from the method as the card is to be dropped. This will also be implemented in the loop by checking the above test. However, if the count of the card is 2 and the drawn card matches the card at the index, changeRandomCard() will be called to change one of the cards which aren't of the drawn card denomination and return it as the card to be dropped.

**testComparePickedCardTwoPairsAndPlayerNumberDrawn():**

The last case that needs to be tested is when we have [X,X,Y,Y], N. Where the drawn card N is the player number. This test will assert X or Y is returned and replaced with the player number, due to it being the player's denomination. The design choice for this is to simply check if the player number is contained in the HashMap of the card and their counts, if it's not we can use the changeRandomCard() method to add the player number into a random index and have X or Y returned to be dropped to the deck on the right.

With all the above tests passing, we can confirm our design choice for creating these tests allowed us to meet the game strategy criteria of the specification.

Furthering testing the cards for the strategy of the game another key aspect is creating output files in the correct location. For our design, we have decided to create the files within the root folder of the project allowing it to be run without any changes tothe source code on any device. We have two tests to assert the creation and location of the files

**testFinalDeckToFileCreated():**

The design for this is to use the FileWriter and BufferedWriter libraries to create a deck file for each deck. Using this we can simply create a file with the correct text inside to be written and stored in the root folder. The test itself asserts that the file has been created within the right directory using the .exists().

**testCreateFile():**

This test is the same as the above however is for asserting an output file for the player is created in the correct location. The design choices for creating the file are the same as the above test and the implementation is the same apart from the content and name of the file.

Tests that also have depicted our design are related to the loading of the pack inputted by the player. The potential issues within the pack are an invalid type being in the pack, the pack not being of the correct length and the path to the pack being incorrect. These potential errors that check these cases are within the CardDeck class and asserted by CardDeckTest:

**testSetCardDeckPackContainsLetters():**

Our design based on this issue was handled within the reading of the file when a card is added to the temp array arr. When reading the file, we only allow the scanner to get the nextInt() so if the next element in the pack doesn't match the type of integer, the reading of the file stops causing the array containing the cards to be on an incorrect length which is then checked. Meaning our design is robust and can deal with any inconsistencies within the pack which don't match the required format.

**testSetCardDeckExceptionThrown():**

This test is for testing when a pack is inputted but the pack isn't at the path inputted by the user. The reason for this test is if the pack cannot be found the game cannot continue. If the pack is incorrect our design requests the user to enter the path again. To catch the exception in our design of the method setCardDeck which reads the file and throws FileNotFoundException. This is then caught in the method getPackFileAndLoadIt() within the CardGame class where the setCardDeck() method is called.

Our design choices for our solution were also determined by several other tests, which can be found within the test.java files with Javadoc detailing what they test. However, in the report, we have outlined a few key tests which check the code meets the specification and ones that were key in our design choices of the source code.