**ECM2414**

***Cover Page***

|  |  |  |
| --- | --- | --- |
| **Name** | **SID Number** | **Allocation** |
| Sangeeth Sohan | 1807071 | 100 |

***Development Log***

|  |  |  |
| --- | --- | --- |
| Date | Time | Duration |
| 11/11/22 | 13.30 | 2hr |
| 14/11/22 | 15.00 | 3hr 30min |
| 15/11/22 | 20.00 | 2hr |
| 17/11/22 | 9.25 | 4hr |
| 19/11/22 | 11.35 | 2hr 15 min |
| 20/11/22 | 18.45 | 2hr 45min |
| 22/11/22 | 20.15 | 1hr 15min |
| 25/11/22 | 12.05 | 1hr 35min |
| 28/11/22 | 8.55 | 3hr |
| 30/11/22 | 11.20 | 2hr 25min |
| 03/12/22 | 12.15 | 3hr 15min |
| 05/12/22 | 17.25 | 3hr 40min |
| 06/12/22 | 11.30 | 1hr 55min |
| 07/12/22 | 13.30 | 1hr |

***Design Choices of Production Code***

*Classes*

Card

The card class does not have that much functionality as the only thing needed for this class is the card value. The reason for this is that is the classes such as CardDeck and Player as the functionality of the deck is given by the CardDeck and in the same way the functionality of the hand of cards a player has given by the player class.

CardDeck

For the decks used in the game, it was decided that a card deck class was needed to represent the functionality of an actual deck. It includes all the methods that shows the functionality of the card deck e.g. player getting card at the top of the deck, giving each deck a number, adding the cards to the deck etc. We also have the method for writing an output file of the deck contents for the number of decks that is used in the game.

Player

The player class handles all the actions that the player does and also the results of which player has won the card game. This class extends Threads which can be used to handle various number threads coming from different methods.

There were two different accesses **lDeck**(left deck)and **rDeck**(right deck)which was used for the players to pick up a card from the left deck and add the card that they do not want to the right deck. These are include in the methods ‘drawCard()’ and ‘removeCard()’. There was an access for the **CardGame** called *gameState*which is used to present what situation of the game the player is in such as player winning the game.

The player class has a constructor which has parameters of the player number, game state, left deck and right deck. These are initialized in the CardGame class for the method for creating the players.

For the players to select the cards what cards they want to discard, we have added the method ‘randomCardDiscard()’. For this we have used the Random class which can generate various amounts of numbers. We added an if statement inside a for loop stating if the card value in the player hand is not equal to the player number then that will be added to the arraylist of discarded cards. Then we created a variable ‘randomDiscard’ to get an int value from the random generator sequence of discardedCard list. Afterwards we add the int value into discardCardList then the card is removed from the hand. Finally the method returns the discarded card.

There is a method ‘playerMove()’ which is used to writes the card values that the player has drawn from a deck and the deck the player has discarded the card to.

There are synchronized methods ‘checkWinner()’ and ‘setGameWinner()’ which prevents the interference of other threads. The ‘checkWinner()’ method has an if statement and for loop stating if one of the card values is not equal to the values in other cards then it returns false else it is true. The method ‘setGameWinner()’ just simply sets up the confirmation that there is winner and including their player id.

There are four methods which involves writing the hand that the player has got in different stages of the card game into the player output file. These methods are:

* writeInitialHandFile()- writes the initial hand of player into the output file.
* writeMove()- writes the current hand of the player after they draw and discard card from and to decks.
* writeLoserFinalHand()- writes to the output file that the winner has informed other player that they have won the game then writes the final hand of the players who have not won the game.
* writeFinalHand()- writes the final hand of the player that has won the game.

Finally we have the run() method which states when the game has yet not been won check if there is a winner and when there is call the setGameWinner() method. Else, the run() method calls the playerMove() and writeMove() method to continue the game. When the game is won then the methods for writing the final hand of the winning player and others are called including ‘writeCardDeck()’ for the right deck contents.

CardGame

CardGame is the executable class which is mentioned in the coursework specification. This is the main class which represents the actual game that is being set up in its final stage then starting to play the game.

This class has the method ‘userInput()’ which is for the user to enter the non-negative number of players in the game and enter the location that the pack file is in which has all the list of cards.

There is an additional method called ‘validPack()’ which checks the validity of this pack to see errors such as a pack file having a card with a negative number or not having an integer. It also checks that the number of cards is equal to 8 times the number of players which is as mentioned in the specification.

After the validity of the pack file checking is finished then we move onto methods for creating the pack where it can add the data of the card values.

Furthermore, we use ‘createDecks()’ for making the total number of decks equal to the number of players that are participating in the game.

‘createPlayers()’ method uses the constructor from the player class to set up the attributes of each players such as giving them an id and the game state that they are in.

Additionally a method ‘moveCards()’ was added for the cards to be moved in a round robin fashion to the players and the deck during the game.

The ‘ startGame()’ method was included in this class as it uses the object inside a ‘for’ loop to check if one of the players won the game when the cards were dealt out. If that did not happen then we move onto starting the player class’s run method through ‘.start()’.

Finally all the methods mentioned above are called in the main method which basically is actual functionality of how the game should run.

**Testing Choices**

Unit testing is one of the most important part when it comes to designing projects like these as it shows whether the code of the Card Game works as intended. The type of Unit Testing that was used in this project was Junit4 as it was covered in the lectures and got familiar of using it more often.

Junit4 testing was used as there were testing methods that was needed such as

The Junit testing covers all the classes in the program that needs to be tested separately in their own classes(CardDeckTest, PlayerTest and CardGameTest).

*Testing Classes*

CardDeckTest

In this class we imported the JUnit annotations ‘@Before’ and ‘@After’. ‘@Before’ annotation was used to execute the method ‘settingTest()’ before each test methods that were in this class. So in this case before running all the test methods it was important to set tests that was important for creating the game which was included in this method such as setting number of players and pack location then creating pack, players, decks and testing for the cards to be moved in a round robin fashion. ‘@After’ annotation is normally executed after each test as it is used to clear up the testing environment. It was used to the delete the player files and deck files that were created when the testing was going on.

There were a list of testing methods that was need for the CardDeck class. One of the testing method was ’getDeckTest()’. This was used to test whether there is the right amount of cards in the deck. We have a method ‘writeDeckTest()’ which includes reading one of the deck files and using it to test whether the deck file that was created gives out the correct deck content output.

PlayerTest

Similar to the CardDeck test class, the player test class also includes the JUnit annotations ‘@Before’ and ‘@After’. Their methods have the same function as the CardDeckTest class but in the ‘settingTest()’ method included to see whether the player wins or not.

The method ‘randomDiscardTest’ is included to test whether the player wants to discard the card that he wants. So normally the player would discard the card that is not their preferred number which needs to match with their player id. For this we included ‘assertNotSame’ method which is used to assert two objects that do not refer to the same object. So based on this we want this method to make sure the player discards a card value that is not their preferred number.

Another method was ‘writeInitialHandTest’ which has a role of reading the player output file and checking whether the player has the correct output of their initial hand.

The method ‘writeFinalHandTest’ is used to test the final hand in the player output file and to do this we had to set one of the players as the winner for testing then read through the player output file. Afterwards, we used assertEquals to test whether the output file gives a wrong final hand.

In the method ‘setGameWinnerTest’ it calls ‘setGameWinner()’ for the winning player. An assertEquals method is included to declaring one winner in the game. With this an assertTrue method is included declaring that the game is won by a player is true.

The runTest() method is responsible for testing the way the cards are drawn from a deck and discarded to a deck. In this we run any one of the player’s threads and we created objects for getting the decks that the player has to draw and discard to. Then we use assertEquals methods to test whether the card is drawn and discarded to the expected deck.

CardGameTest

Similar to the other two test classes this too has the ‘@Before’ and ‘@After’ annotations for their methods which are used for the similar purpose as well.

This class includes the ‘UserInputsTest()’ method which is used to test the validity of the pack. It includes the testing for whether the file type exist and the number of players that have been inputted. AssertEquals has been used to test whether the expected pack location and number of players is correct.

There are many test methods in this class which is used to check the validity of each attributes that the CardGame needs such as:

* ‘createPackTest()’ - used to test the validity of the length of the pack.
* ‘validPackTest()’ - used to test whether pack location is valid.
* ‘createDeckTest()’ - used to test whether the number of decks is valid.
* ‘createPlayerTest()’- used to check the testing of whether input of the number of players is valid.

The method ‘moveCardsTest()’ is used to test the movement of the cards in the game from the deck to the player’s hand. For this testing we have added two objects which includes the player and deck of the same number. It also includes getting the objects for the number of cards that should be in the player hand and number of cards then creating variable to get the values of the cards in those section. Afterwards, we moved onto checking the testing was correct for each of these such as testing number of cards in hand, size of player hand, number of cards in deck and card value in the deck.