ECM2414 Card Game Report

By 710048828 and 710033318

Final mark allocation: 50/50

Development log:

Date:	Duration (hours)	Driver:	Observer:	Worked on:
01.11.22	0.2	N/A	N/A	• Talked over whatsapp, deciding to create a UML design for the project
04.11.22	0.2	N/A	N/A	• Looked over both the UML designs and chose what we liked from each, and which one to go forward with
05.11.22	1	710048828	710033318	• Created all the classes from the UML design and the basic class methods
08.11.22	2	710033318	710048828	• Testing basic methods created last time • Swapped driver half way through because of JUnit
09.11.22	1	710048828	710033318	• Creating more advanced methods eg. Opening pack file, creating players, creating card decks
13.11.22	1	710033318	710048828	• Creating the player run method, and all the log methods

Design choice:

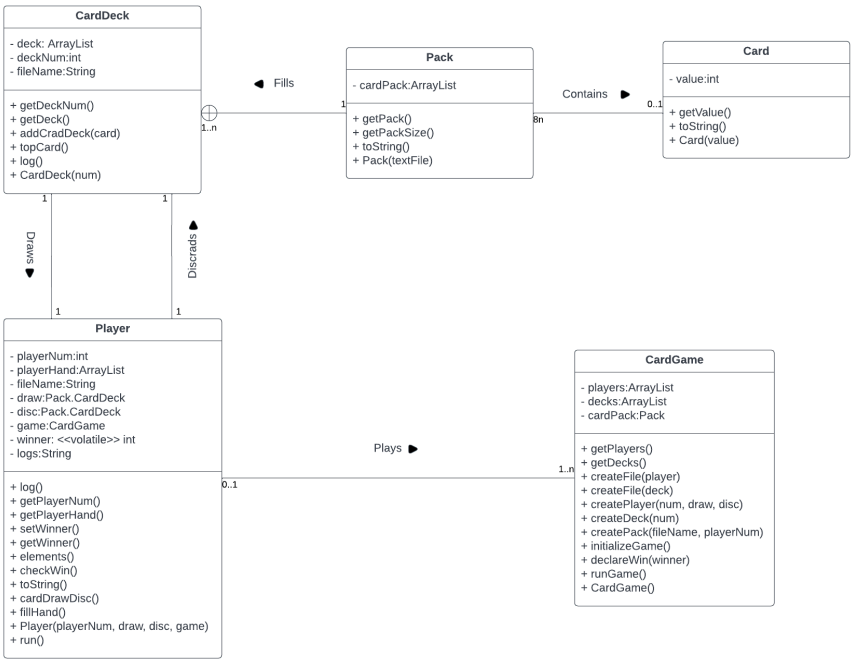

Figure 1: *UML design for the Card Game*

The specification required us to make a thread safe card game, containing at least a Card class, Player class, and an executable Card Game class. Firstly, to make sure that the CardGame class was executable, the public static method was written inside the CardGame class. Reading through the assessment specs to make sure the game runs properly we decided to add 2 more classes. First one being a Pack class, and a static nested CardDeck class within the Pack class. This can be seen in FIgure 1. At the start we thought of making the Pack and CardDeck 2 separate classes. However, since the cards from the Pack are used to fill the Card Decks, the 2 classes are related. To ensure the code is more readable and more maintainable we decided to make the Card Deck an inner nested class. Furthermore, the Card Deck is a static class. The Pack class is the one that fills the Card Deck. In other words, the Card Deck doesn't need to access and invoke methods on the instance of the Pack class.

To ensure that the cardinalities of the players, cards, and card decks conform to the game rules, from Figure 1 within the Card Game class we have a method called initializeGame(). When invoked, the method will ask the user to input the number of players participating in the game after which it will ask for the file containing the values of the cards for a pack. It will not create a pack until a file with the correct amount of values is loaded ($8 \times \textit{The number of Players}$). After the method successfully creates the pack it will go through a for loop and create the players for the game and an equal amount of Card Decks.

One thing we also had to guarantee was that the players and the card decks formed a ring topology. For this to happen, as could be observed in Figure 1 under player class, we decided to include the Card Deck the player would discard to and draw from as 2 arguments in the Player object constructor. This made sure that the players would only draw and discard to specific card decks. However, this isn't the end of it. We couldn't just say in our code player[i], i being the player number, draw from deck[i] and discard to deck[i+1],

because then player[n] would have to discard to deck[n+1] which does not exist. To avoid this in the initializeGame() method we create the card decks first, place them into an ArrayList and then when creating a player we assign them decks to draw and discard to. Within a code there is an if statement that checks for the last player to assign him to discard to deck 1.

We decided to use ArrayList as our main data structure because we felt it was the most effective and easiest to implement within the game. The players and card decks are stored in an ArrayLIst within the game, and the cards in the player hand are stored in an ArrayList within the player. There are 2 main reasons we decided to go with ArrayLists. First one being direct access to elements. There are no search methods within the game and most of the methods used in the game only require to use the first or last element within the list. Hence, there is no performance loss of using ArrayList since we will never need to search for anything. Second reason, the ArrayList will not contain null values. If instead of an ArrayList we used an Array there could have been the danger of a thread drawing a null value into its hand.

In the initializeGame() method, the players are given 4 cards in a round robin fashion, and then each card deck is given 4 cards from the pack also in a round robin fashion. To check if any player has been given a winning hand at the beginning of the game we use 3 methods. From Figure 1 in the player class the checkWin(), declareWin() and the runGame() in the card Game class. Once the runGame() is invoked the system checks every players hand with the checkWin() method, if any player has a winning hand checkWin() will set the volatile variable winner to the player number and evoke a method declareWin() which will notify every player who has won and then log all their actions. It was very important to make the winner a volatile variable. If not there is a risk of the threads not noticing that the variable has changed and would result in it continuing to run when it shouldn't.

If none of the players have a winning hand at the start the system will go through every player and invoke the run() method. Once the run() method has been evoked every player will draw and discard cards and then check if they have a winning hand with the checkWin() method. **IMPORTANT** we did notice that sometimes for some reason the player tries to draw an empty card which then crashes the game. If that happens we just ran the game again and it seems to work fine. Since multiple threads could access the same Card Deck and cause race condition, the player cardDrawDisc() method is synchronised. In addition, at the beginning of the method it checks that the draw deck contains exactly 4 cards otherwise it returns from the method. Furthermore, the draw and discard method had to be 1 atomic action. That is why we made it 1 method. At first the player will check his current cards in hand to see if any card is of the same value as his player number. The other cards will be added to an ArrayList which is then shuffled, to ensure a random card being discarded, and the first card will be removed from the hand. The discarded card is then added to the discard Card Deck after which the player picks 1 card from his draw Card Deck.

Lastly, for the logs we have 2 methods log() in the player and card deck class. Both classes have private string logs which are updated every turn with the new actions. WHen coding we tried to make the player and card deck log every turn, but later realised the method would just keep creating new files every log action. To battle this we decided to add every action to a string and log it to a file at the end of the game. To ensure the files are created with appropriate names both card deck and player class have a string file name.

Testing:

For our tests we opted to create them by utilising JUnit 4.1 framework. All of the following tests are also encompassed within a test suite name TestSuite. To run the tests, we recommend using an IDE which supports Junit 4.* like Visual Studio. More information on this can be found within the ReadMe file.

The first class we designed tests for was the Card class. It is a very simple class therefore we only have the one test. testGetValue() ensures that the constructor can create a class as instructed and that the getValue() method is able to return the correct value from this class.

TestDeck provides tests for the Deck class which is an inner class within Pack. The reason we decided to give the Deck class a separate test file despite it being an inner class was to allow us to organise our tests better and make them easier to understand.

The setup for TestDeck involves creating two decks. One filled with four cards numbered 1 to 4 and the other filled with just 1 card. There is also an array which is a representation of what deck1's values should be.

testGetDeckNum() is simple and confirms the deck has been created correctly and checks that getDeckNum() is returning the correct value.

testGetDeck() is checking that getDeck() is returning both the correct number of cards from the deck and that they are returned in the correct order as this is imperative for the game to work logically. This is done by comparing the fetched values against a hardcoded array of values which are known to be correct.

testAddCardDeck() is testing that a card can be added to a deck and when it is added it is in the correct place because as mentioned previously the order of the cards must be correct.

Finally, the testTopCard() method verifies that when topCard() is called it does in fact return the card from the top of the pile by checking the value fetched against the known value of the top card. Whether this card is also removed from the deck is tested by checking that the size of the deck decreases by one.

The next tests we designed were for TestCardGame. The setup for the tests simply creates a player and deck containing nothing other than their number which are then added to a game object to allow the game to be tested.

Firstly, testGetPlayers() simply confirms that getPlayers() returns the correct information on the players by checking that the player fetched matches the correct player. The size of the array returned is also checked for extra certainty.

testCreatePlayer() first confirms that createPlayer() is able to add a player to the player list correctly and that the player added is constructed correctly by comparing the constructed player to a known correct player. It also confirms that the list is now just one bigger as it should be.

testGetDecks() is very similar to testGetPlayers(). It simply confirms getDecks() is able to return the correct list of decks.

testCreatePack() is also very similar to testCreatePlayer(). It uses the same logic of ensuring the deck is constructed as requested and that it is correctly appended to the list of decks.

There are a couple methods within CardGame that do not have specific unit tests written for them. These are initialiseGame() and runGame(). We decided against having unit tests for these methods for several reasons. Firstly, when designing these tests at the beginning of development, our concept of what these two methods would look like was limited because of their complexity so writing any tests would be challenging. Then, as development

progressed it became apparent that due to these methods' complexity they would be better suited to be tested during integration testing. This is because both methods essentially tie all other methods and classes together.

For the Player class tests we have created a set up function. It starts off by creating 5 different cards with values going from 1 to 5 respectively. We then create 2 decks, the first being a deck the players will draw from, with the first 4 cards. The second deck being the deck the player will discard too with the last card. We create 3 different players. First one has a player hand with no matching cards, the second player has a winning hand, and lastly the third player is only 1 card away from having a winning hand.
The first test we designed was to check that the Player getPlayerName() method was functioning correct. We are asserting that the function getPlayerNum() will return 1 from player1.
The next test designed was to test that the player hand was set up correctly. The method testGetPlayerHand() will be testing the player getPlayerHand.add() and getPlayerHand.size() method. First the test will assert that the player hand is of size 4. Then in the setup we created an Array named correct1 which contains the theoretical correct hand of player 1. The test will run and assert that player1 contains the cards with values 1, 2, 3, 4.
The test testGetWinner() checks that when the game is initialised there is no winner set. After the test is run it will assert that there is no player that has won the game by returning a value 0.
The test testElements() checks that a player hand is returned in string form with all the card values. The test will assert that player 1 elements are "1, 2, 3, 4".
The next method we are testing is the player checkWin(), and the getWIn() function. The first part of the testCheckWIn() checks that there are no winners after the players have been initiated. However, after setting everything up we didn't add the players to the game player ArrayList, so they have individual winner values. We did so to isolate every player and test the method checkWin() to see that it functions correctly. Second part of the test runs the checkWin() method on player1 and player2, we set it up so that player 2 has a winning hand. We then assert that the player1 winner value has not changed where the player2 winner value changed to its own player number.
The next test is designed to check the player cardDrawDisc() method. The test starts off by having player3 draw and discard a card. In the setup the draw deck contained 4 cards and the discard deck contained only 1. After the player draws the card we assert that the draw deck size has gone down to 3 and the size of the discard deck has gone up to 2. Following that we set it up that player3 has cards of value 3,3,3,5, we then assert that player3 discarded the card that isnt equal to his player number. Furthermore, for thread safety we try to invoke player1 to draw and discard cards. The system is designed that if the draw pack isnt of size 4 nothing should occur. We assert that after invoking the method nothing has changed in the decks and the player hasn't drawn or discarded any cards.
The last method we test for the player is the fillHand() method which adds a card to a player hand. We run the fillHand() method and try to add a card of value 3 to player 1. We then try to assert that the player's hand has increased by 1 and the new card is in his hand.

For the Pack class tests we had to create a mock object. We noticed that it was taking a very long time to create a pack, and JUnit had issues with opening files, so we decided to create a Mock object of a pack to test the methods. We created 5 different cards and added them all to the pack.

The first method was to test that the pack created had the correct size of 5. We are asserting that the getPackSize() method returns the correct value.
The last method tested for the pack was the getPack() method which returns every value of the cards in the pack. To do this we run a simple for loop to see that the values of the cards go from 1 to 5.

Throughout our unit tests we did not test much of our logging elements as it was not feasible to automate the testing of the logging. This is because the logging recorded all actions taken by the game which obviously implements threads. Therefore, due to the non-deterministic nature of threads in java we incorporated the testing of our log methods during integration and final testing.