# ECM2414 Cover Page

|  |  |  |
| --- | --- | --- |
| Student ID | 227775 | 250107 |
| Weight | 50% | 50% |

# Development Log

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Date | Time | Duration | 227775 | 250107 | Purpose | Signatures |
| 7/11/2022 | 11:30 | 2h | Driver | Observer | Discussion about implementation | 227775, 250107 |
| 9/11/2022 | 13:00 | 1.5h | Observer | Driver | Project file and initial class creation | 227775, 250107 |
| 14/11/2022 | 14:00 | 2h | Driver | Observer | Player & Deck class creation | 227775, 250107 |
| 15/11/2022 | 11:00 | 2h | Observer | Driver | Card & Pack class creation | 227775, 250107 |
| 16/11/2022 | 14:00 | 1.5h | Driver | Observer | Bug fixes | 227775, 250107 |
| 18/11/2022 | 15:00 | 2h | Observer | Driver | Testing classes made | 227775, 250107 |
| 19/11/2022 | 10:00 | 2.5h | Driver | Observer | CardGame class made, and main game completed | 227775, 250107 |
| 20/11/2022 | 14:00 | 2h | Observer | Driver | Testing classes finished & CardGame test | 227775, 250107 |
| 21/11/2022 | 11:00 | 2h | Driver | Observer | Threading bug fixes | 227775, 250107 |
| 22/11/2022 | 10:00 | 2h | Driver | Observer | Game testing complete | 227775, 250107 |
| 22/11/2022 | 18:00 | 3h | Observer | Driver | Reports written & jar file built | 227775, 250107 |
| 23/11/2022 | 14:00 | 1.5h | Driver | Observer | Final checks before submission | 227775, 250107 |

# Design Choices

Diagram

Description automatically generatedThe overall design for the program is shown in the UML diagram to the left. All classes are used in the main function of the CardGame class.

The CardGame class deals with creating the players and decks, dealing out the cards (validation the inputs given) and starting the players threads.

The CardGame() method takes inputs from the user specifying the pack file and also the number of players. This input is read using a BufferedReader (bReader), validating the number of player given by the user is done within this method, however by passing the bReader object into the Pack() method of the Pack class, we were able create a new Pack instance, while keeping the implementation of checking for the validity of the pack file within the Pack Class.

Once the CardGame() method has a valid number of players and an instance of a Pack, it them runs the dealCards() method – this method creates the players (when a player is created their left and right decks are passed in as variables to the creator method), decks, and deals the cards to specification (storing the players in the players : ArrayList<Player> attribute). Finally the CardGame methods checks whether any of the players already have a winning hand and the runs threads for each player.

When designing this program, we decided that a passive way to avoid stagnation (i.e. cards remaining in the same place indefinitely, and the game not ending) was to store the cards in queues. Expanding on this we decided that the Player and Pack classes should inherit from the CardDeck class, because all three entities have the same basic storing/drawing and discarding of cards.

In order to make a player’s go atomic, we implemented a static AtomicInteger `winner` that stored the winning player’s number and is checked before every player starts their go – if `winner` has been set then a player has won, and all player will stop their next go, if they are in the middle of a go then this finishes, and they are stopped when they next try to play. This also provides a simple way for dealing with two players winning at the same time – because setting the value of `winner` is an atomic action, when a player “wins” they try to set `winner` to their number, if this has already been set then they still lose (i.e. first to change `winner` wins the game).

When designing the program to be thread safe we initially make the Player.playGo() method static, this was to insure the atomic requirement of a player’s go. This however led to only one player’s thread playing at any one time, which we felt defeated the point of multithreading. So insure the thread safety of taking and discarding cards to the deck, we synchronized the takeCardFromTop() and placeCardOnBottom() methods – thus allowing only one player to access a deck at the same time, but multiple player to player at the same time.

We decided to implement Runnable in the Player class. Originally the Player.playGo() method was void, but we changed it to return a Boolean – `true` if the player should play another go, `false` if one of the players has won. This allowed a very simple while loop to implement the running of a player go in the player’s .run() method.

In the validation of the pack file, we initially had a counter that incremented when a new line was reached and invalidated the pack when to many lines where being read (according to the number of player). This led to clunk looking code and would only provide noticeable performance improvements in the case that a pack file is exceptionally larger. Thus, assuming the pack file only contain number (invalidation occurs while reading the file if a line doesn’t have a number in it) the whole pack file is parsed into the pack’s cards queue, and then the length of this queue is checked against the expected number of cards.

# Text Description automatically generated with medium confidenceTesting Choices

Tests for all the main classes have been created. These are:

* CardDeckTest.java
* CardTest.java
* PackTest.java
* PlayerTest.java

The tests were written using JUnit version 4.13.1. The tests were made to ensure that each of the classes function correctly.

## CardDeckTest.java

The CardDeck class is used for storing the card objects in a queue data structure, this is used throughout the program and therefore testing of its methods is essential. The main methods that are used in the CardDeck class that need to be tested are the drawing of cards and the placing of cards from and onto a deck.

### void testToString()

This testing method was made to ensure that the ToString override method in the CardDeck class is outputting valid information.

### void testTakeCardFromTop()

This class was made to test that the correct values were being taken from the top of the queue. This was done by:

1. Initialising a new deck
2. Recording the values in the deck
3. Run the method takeCardFromTop
4. Assert the values inside of the deck are now as expected and not the same as the value before the method was executed

### void testPlaceCardOnBottom()

Like the previous method, this test is used to ensure that the cards are properly being places on the deck. This was done by:

1. Initialising a new deck
2. Recording the values in the deck
3. Run the method placeCardOnBottom with a new card as the parameter
4. Assert that the values in the deck are now as expected and not the same as the value before the method was executed

## CardTest.java

The Card class is used to store the value of the card. In this class there is just a simple getter and setter functions.

### void testGetValue()

This method is to test that the getter function inside of the Card class is working as intended

### void testToString()

This method is to test that the value of the ToString override is outputting the correct values when executed

## PackTest.java

This testing class is used to ensure that a text file can be read into the program correctly and that the correct value is output when requested.

### void testToString()

This method is used to test the override ToString method. To initialize the pack object that I used for testing a method called initFile() is used to create a new text file and populate it with numbers, simulating a pack.txt file. This is then read into the program using the constructor in the pack class.

## PlayerTest.java

This class is used to test all the methods within the Player class, this is very important as these methods are used throughout the program. For this testing class a new player object is initialized at the beginning of each method, this is then used to test the functions with respect to the deck they were given when initialized.

### Player createPlayer()

This is a method that is used in many of the test methods to construct a new player object and then populate the players decks with the required cards for testing

### public void testForceDiscard()

This method works by giving the player four different starting cards, with one of them being the preferred card. Then trying to force the player to discard the four cards in their hand. This checks that the player doesn’t discard the preferred card.

### public void testThreadSafety()

This method has been created to check that deck accesses are thread safe, this is done by creating 4 player objects and then creating new processes for their deck accesses so that they can happen simultaneously. The decks at the end are then checked to have an equal number of cards.

### void testDrawCardFromDeck()

This test is used to ensure that the correct card is being drawn from the players left deck and then placed into their hand. This is done by:

1. Recording the value before any changes have been made
2. Executing the drawCardFromDeck() method
3. Recording the expected and actual output
4. Assertion test to check these outputs

### void testDiscardCardToDeck()

This test is used to ensure that the correct card is discarded from the players deck and is then placed into their right deck, this is done by:

1. Initialising a new player object with the queue of cards
2. Recording the value before any changes have been made
3. Executing the discardCardToDeck() method
4. Recording the expected and actual output
5. Assertion test to check these outputs

### void testPlayGo()

This test is to simulate the players’ hand after they have played a go and ensures that the output is as expected.

1. Simulate a round of play by executing the playGo() method
2. Record the expected output of the deck
3. Assertion test this against the actual output

### void testToString()

This method is used to test the ToString override in the Player class.

### void testAllSameCards()

This method is used to check the getPlayerWon method in the Player class is working correctly, this is done by:

1. Initialising a new player object with the queue of cards that are all the same
2. Checking the value of the getPlayerWon method which will return true if all the players card values are the same

### void testLogOutput() & void testLoseOutput() & void testWinOutput()

This method is used to check that the correct messages are being outputted to the text file, this is done by

* Initialising a new player object
* Executing the drawCardFromDeck method, which will output to the text file what happens
* Reading the text file and checking that value against what is expected

### void testCurrentHandOutput()

This method is used to output the current hand, this I used throughout the player class when outputting to the log file, this is done by

1. Initialising a new player object
2. Testing the output of the getOrderedCards method with an assertion test

### void testSetPlayerWon()

This method is used to check that the playerWon method within the player class works when the deck satisfies the necessary requirements, this is done by:

1. The players deck will contain four 1s and one 2, this is to ensure that the player hasn’t won the game on initialization
2. Execute the discardCardToDeck method to discard the single 2 card in the deck preventing it from being a winning hand
3. Execute the drawCardFromDeck method to make the players deck contain one 1s
4. Execute the setPlayerWon() method
5. Use an assertion test to check that the two values are different