Cover Page

Student Name	**Candidate Number**	**Weighting**
Jack Hales	221808	50%
Gregory Korchagin	202859	50%

Development Log

Date	**Time**	**Duration (h)**	**221808**	**202859**
8/11/23	15:30	2	Driver	Observer
9/11/23	14:00	2.5	Observer	Driver
10/11/23	12:00	2.5	Observer	Driver
11/11/23	11:00	3	Driver	Observer
16/11/23	11:00	2	Driver	Observer
22/11/23	15:30	0.75	Observer	Driver
23/11/23	13:00	2.5	Observer	Driver
26/11/23	12:00	2.5	Driver	Observer
27/11/23	20:00	1	Driver	Observer

Design Choices – Production Code

Overview:

The Card Game is a multi-threaded program that simulates a game played between an arbitrary number of players, that consists of each player repeatedly drawing a card from the deck to the left, whilst also discarding one to the deck to the right. The aim of the game is for all four cards in your hand to have the same value. If a player hand consists of four cards with the same value the game ends, otherwise each player continues to draw and discard from the decks either side of them. At this time there are no known performance issues with the program. The game has been run with the IntelliJ IDEA Code Profiler open which shows no notable issues – for example, CPU utilisation remains at 0% which shows that the program is well optimised.

Before the game begins, the user must enter the number of players and a valid pack file for the given number of players, which consists of 8n cards, where n is the number of players. If the pack file doesn't contain 8n cards, or if it contains non-integers, the game will through an exception and prompt the user to re-enter the number of players and the pack file. It is possible for a player to be dealt a winning hand immediately, which is a valid win case. However, this can be prevented by changing the order of values in the pack file. Similarly, it may be impossible for certain players to ever win if there is at least one value of their preferred domination, but less than four. If this is the case for all players, no one will win, unless they draw a winning hand when the game starts. The specification describes a valid pack file as *"a plain text file, where each row contains a single non-negative integer value, and has 8n rows"*, thus we concluded that the game should not check that a pack file allows the game to be winnable, however, this is something that could be implemented if the requirements were extended to cover this.

The game is split into five separate classes, each of which are covered below, in alphabetical order.

Card:

The Card class is used to store value of each card from the pack, and thus there are 8n instantiated Card objects when the game is played. This is the most basic class of the five as it just has a single attribute to store the card value, as well as a constructor, a getter and a toString() method. At the start of the game, each Card object will be put into a Pack, before being dealt to a CardDeck or Player, after which the game begins. It is necessary to have each Card as its own object to ensure the core principles of object-oriented programming are enforced and maintained.

CardDeck:

The CardDeck class in the card game effectively manages a deck of cards, encapsulating core object-oriented principles. Each deck, uniquely identified by an integer deckNum, manages its cards using an ArrayList<Card>, allowing efficient operations like adding cards using addCardToDeck() and removing cards using removeCardFromDeck(). The class also handles file output for each deck's actions, with a method for writing using writeOutputFile() and converting the deck's contents to a string format using deckToString(). This design facilitates tracking of the game's progress and debugging, and also provides a clear interface for interacting with the deck. The constructor initialises an output file for each deck, and a specialised method playerHasWon() logs the final state of the deck when a player wins.

CardGame:

The CardGame class manages a multi-threaded card game by managing initialisation, gameplay, and the game's conclusion. In its main method, it prompts for user input to set the number of players and the card pack file, catching input and file-related exceptions. The runGame method initialises the game, creating Pack, Player, and CardDeck objects, and distributes cards to players and decks. Players, represented as threads, each start with four cards, and the game checks for an initial winner. If no immediate winner is found, a synchronised gameplay loop begins where players draw and discard cards from decks, with the game continuously checking for a winning condition. Upon identifying a winner, the game declares it and informs all player threads and decks to conclude and stop, using management of game state, concurrency, and thread synchronisation.

Pack:

The Pack class is used to read in a pack file at the start of the game and generate a pack of cards from it so the game can be played. Similarly to Cards, it is a relatively simple class, with just an array containing all the cards, a constructor to create the pack, and a getter to read the pack. The constructor method reads the pack file supplied by the user and attempts to create a pack from it. However, it may throw an IOException if there is an error reading the file, or an IllegalArgumentException if the pack contains non-integer values or has an incorrect number of values for the given number of players. These exceptions are then handled by CardGame and will allow the user to re-input the number of players and a pack file.

Player:

The Player class, extending Thread, encapsulates the behaviour and state of a player in the card game. Each player, identified by playerNum, maintains their hand of cards in ArrayList<Card>, and handles game-related outputs through outputPath. The constructor sets up initial attributes, creates an output file, and manages file I/O exceptions. Player actions are logged using writeOutputFile(), providing a record of each move. The addCardToHand() method adds cards to the player's hand, logging initial and subsequent card draws. Players make decisions using chooseCardToDiscard(), which employs a random strategy to discard non-preferred cards. The sameValueHands() method checks if all cards in a player's hand have the same value, determining if they've won. Upon game completion, playerHasWon() logs the game's outcome, indicating either victory or acknowledgment of another player's win. This class forms a core component of the game's functionality by integrating thread management, gameplay logic, and file output.

Design Choices – Tests

Overview:

The tests for this project were written using JUnit 5. Overall, they have 95% (21/22) method coverage and 77% (121/156) line coverage. The one method which isn't tested is the main() method in CardGame where the user enters the number of players and the corresponding pack file. These values are taken from the code coverage tool in IntelliJ IDEA.

There are five individual test classes, one for each class, and these are covered below in alphabetical order. Moreover, a test suite is included, CardGameTestSuite, that will run all five test classes without needing to run each class individually.

CardTest:

The CardTest class consists of a setup method, and two specific tests that target key aspects of the Card class.

In the setUp method, a new Card object is initialised before each test using a randomly generated value. This approach provides a diverse range of inputs for testing and also enhances the robustness of the tests by ensuring that the Card class behaves correctly across various scenarios.

The testCardConstructorAndGetValue() method focuses on validating the Card constructor and the getValue() method. It confirms that the card's value is correctly initialised by the constructor and accurately retrieved by the getter. The assertion checks if the value obtained from getValue() matches the initial random value, ensuring the constructor's functionality and the accuracy of the getValue() method. This test is crucial for verifying that the Card class correctly encapsulates and provides access to its core data.

The testToString() test is designed to assess the toString() method of the Card class. This test verifies that the method returns a string representation of the card's value that matches the expected format. By comparing the toString() output to the string conversion of the card's value the test ensures that the method accurately reflects the card's current state.

CardDeckTest:

The CardDeckTest class tests the functionality of CardDeck. Each test method uses a setup method that creates a new deck prior to the test methods running.

The testAddCardToDeck() method tests that a given card can be added to the deck. To this end, a Card object with a random value is generated and added to the deck, with an assertion checking it has been added successfully.

The opposite of testAddCardToDeck(), the testRemoveCardFromDeck() method tests that a card can be removed from a deck. The test method first generates a deck of four cards (to mimic the game), before asserting that the first card in the deck is removed when the removeCardFromDeck() method is called.

The testGetCards() method tests the getter method for the Deck object. Similarly to testRemoveCardFromDeck(), it first generates a deck of four cards to mimic the game. It then asserts that all four cards are returned with the correct values when the getCards() method is called. To test it works after removing a card, one card is removed from the deck, and another assertion is used to check the deck now has three cards with the correct values.

The testWriteOutputFile() method uses Reflection to test the private writeOutputFile() method. It generates a random alphanumeric string and invokes the private method, before asserting that the output file contains the correct alphanumeric string.

The testDeckToString() method tests the deckToString() method which returns the string representation of the current deck in the format required for output (n_1 n_2 n_3 n_4). It generates a deck of four cards, and then asserts that the returned value from the method has the correct values and is formatted correctly.

The testHasPlayerWon() method tests the playerHasWon() method which is called when a player has won the game. It causes the deck to write its contents to the output file as detailed in the specification. The test method first generates a deck of four cards, and then checks that the output file has been written correctly after playerHasWon() is called.

CardGameTest:

The CardGameTest class tests that CardGame runs without any exceptions being thrown. As CardGame only has one method, runGame(), other than the main() method where the user enters the number of players and the corresponding pack file, there is only one test method.

The testRunGame() method checks that runGame() completes successfully using three different packs, one for two players, one for three players and one for four players. These are all valid packs and as such the game should run without any exceptions being thrown which is the purpose of this test method. It is difficult to test specific functionality within this method as it is quite large and consists of the majority of the game logic. It also calls methods from the other four classes which are all tested in their own test classes.

PackTest:

The PackTest class features setup, parameterised, and standard test methods, each targeting specific functionalities of the Pack class.

In the setUp() method, PackTest initialises variables for the number of players and the expected number of cards. It uses a random number generator to simulate different game sizes, ranging from 2 to 4 players.

The test suite includes a parameterised test, testPackConstructorWithValidFile(), which is executed multiple times with varying numbers of players. Each iteration validates the Pack constructor with a corresponding valid file, ensuring the non-nullity of the card array and verifying its length matches the expected number of cards. This test is useful for confirming the Pack class's ability to process valid input files accurately.

The class contains tests for edge cases, including testPackConstructorWithTooManyCards(), testPackConstructorWithTooFewCards(), and testPackConstructorWithInvalidData(). The first two tests verify the Pack class's response to files with an incorrect number of cards, expecting an IllegalArgumentException in scenarios of both excessive and insufficient cards. This ensures the class effectively handles scenarios where the card count deviates from the expected range.

The final test, testPackConstructorWithInvalidData(), assesses the class's ability to manage files containing non-integer values, anticipating an IllegalArgumentException. It checks the class's capability to handle invalid data types within the pack file, which ensures game integrity.

PlayerTest:

The PlayerTest class tests the functionality of Player. Each test method uses a setup method that creates a new player prior to the test methods running.

The testGetPlayerNum() method tests that the correct player number is returned when the method is called.

The testWriteOutputFile() method uses Reflection to test the private writeOutputFile() method. It generates a random alphanumeric string and invokes the private method, before asserting that the output file contains the correct alphanumeric string.

The testHandToString() method tests the handToString() method which returns the string representation of the current hand in the format required for output (n_1 n_2 n_3 n_4). It generates a hand of four cards, and then asserts that the returned value from the method has the correct values and is formatted correctly.

The testAddCardToHand() method tests that a given card can be added to the player hand. Two cases are tested in this test method. Firstly, four cards are added to the hand to simulate four cards being dealt at the start of the game. An assertion then checks that the first line in the player output file contains the initial hand. The second test case checks that when a card is added to a hand after the initial dealing, the correct line is written in the player output file to say which card they have drawn and from which deck.

The testChooseCardToDiscard() method tests that a card is removed from the player hand, and that it isn't the player's preferred card denomination. To this end, it first generates a full hand as though the player has just drawn from a deck, and then calls the chooseCardToDiscard() method to discard a card to a deck. An assertion then checks that the discarded card does not have a value matching the preferred card denomination. It also asserts that there are two lines in the player output file saying which card was discarded, and what their current hand is.

The testSameValueHands() method tests whether a player hand contains four of the same card. Several test cases are included to consider all possible cases. Firstly, two assertFalse's are used to check that false is returned if there are only three or five cards in the player hand, even if they are all equal. Next, an assertTrue is used to check that true is returned if all four cards match. Lastly, an assertFalse is used to check that false is returned if there are four cards in the player hand that aren't all equal. Thus, this test method covers all valid, invalid and boundary cases.

The testPlayerHasWonSelf() method tests that the correct text is written to the player output file when they have won the game. To this end, a winning hand is generated before playerHasWon() is called. Three assertions are then used to check that the correct three lines have been added to the player output file saying that they've won, exited, and what their final hand is.

The testPlayerHasWonAnotherPlayer() method tests that the correct text is written to the player output file if another player has won the game, ie. this player has lost the game. To this end, a non-winning hand is generated before playerHasWon() is called with the player number of another player (ie. the player who has won). Three assertions are then used to check that the correct three lines have been added to the player output file saying that another player has won, the player exists, and what their losing hand is.