ECM2414 Report

Cover Page

Student ID	710020413	700037354
Weight	50%	50%

ECM2414 Report

Development Log

Date	Time	Duration (Hr:Min)	Roles	Signed
31/10/2022	12:00	5:41	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502
01/11/2022	14:00	4:30	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502
02/11/2022	16:00	3:00	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502
04/11/2022	13:00	2:00	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502
07/11/2022	13:00	3:00	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502
09/11/2022	17:25	2:35	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502
11/11/2022	13:37	2:22	710020413 - Testing 700037354 - Testing	208330 204502
14/11/2022	21:00	3:30	710020413 - Observer 700037354 - Driver Testing	208330 204502
15/11/2022	16:00	5:00	710020413 - Observer 700037354 - Driver Testing	208330 204502
16/11/2022	15:50	3:10	710020413 - Driver Testing 700037354 - Observer	208330 204502
21/11/2022	11:45	1:30	710020413 - Driver + Observer 700037354 - Driver + Observer	208330 204502

ECM2414 Report

Design Choice and Reasons with respect to both production code and performance issues

Component In Solution	*Design Choice*	*Reason*
CardGame	Decomposing game start-up into methods	By Breaking down the setup of the game into functions, we can visualise the problem more easily. Each method required at start-up has a descriptive name hinting at its purpose and ensuring that any outside party will have a clear understanding of what is going on.
Players	Player contains deck attributes	We have assigned the attributes of 'LDeck' and 'RDeck' to each player instance. These are references to the deck theoretically to the left and right of the player. Containing these objects' references within a player ensures that we are not performing a computation to locate the correct decks on each player's action, thus, making performance better.
CardDeck	Representing decks as Queues	We used this data structure to represent the card decks as it provided unique functionality useful for manipulating the cards such as dequeue and enqueue. In contrast, using a stack would prevent us from placing a card on the bottom.
CardDeck	CardDeck contains ID attribute	By having a deck ID this allows us to identify the deck from all other decks. Thus, enqueueing and dequeuing cards from the deck is faster as we can identify the deck needed with a direct unique identifier.
Players	Players contains ID attribute	When we instantiate new players, they are given an ID, this ID is used to both assigning what Card Deck's will be to the left and right of them; and identifying the player from all other players.
Players	Player contains hand attribute	This attribute is the cards that the player currently has in their hand. We choose to use an ArrayList data structure as it provided the functionality required for an ambiguously chosen card to be removed from the deck.
FileOutput	Abstract class for player output.	By having each player class extend the FileOutput abstract class, it separates the functionality of outputting to a file from the functionality of a player. Thus, increasing encapsulation and performance as there is no repeated code, coupled with files being opened and written to in each instance.
FileOutput	Polymorphism with method overloading	We have used polymorphism with method overloading to create multiple methods that return a hand to string, each method depends on the arguments that are passed in it. These methods are used for getting the string output for a player's hand and a decks hand.
FileOutput	Abstract class for deck output	By having each deck class extend the FileOutput abstract class, it separates the functionality of outputting to a file from the functionality of a deck. Thus, increasing encapsulation and performance similar to the player class.
CardDeck	Observer thread	Our observer thread starts the player threads using a thread group and then waits to be interrupted when a player has won. The observer thread will then write the final state of the decks to output files. We chose to implement our design in this

		way as we wanted to increase encapsulation and thus separate this functionality of the decks from the players. Using this implementation also solved the issue of having multiple players writing the output of the deck.
Players	Boolean win condition	Within each player thread, there is a condition to check if the player has won when the game has ended. If the player has not won, the player outputs which other player has informed them of their loss. Having this encapsulated within each player instance allows a player to handle its own loss, once it has identified a player has won. Being in the threaded run method increased performance as it allows it to occur simultaneously between the players and without further method calls after the game is over.
Packs, CardGame	Packs validation in stages	To reduce errors and save computation in the program, we have decided to split the validation of a given pack into stages. Firstly, we check if the file provided exists before any internal checks can be completed. If the pack doesn't exist, we do not perform any more checks. Saving time and computation. This principle is used again when checking the contents of a pack. We have chosen to check if the pack meets the rule of 8n first, so that we are not wasting computation on the process of checking if the pack contains all positive numbers.
CardGame	Thread group interruption	When a player has won, all other players need to cease all actions simultaneously; to achieve the most realistic simulation. Rather than sequentially stopping each player thread, we have grouped the threads together so that one singular command can be called to stop all threads at the same time.
CardGame	Atomic and volatile variable for gameOver	In this simulation it is important to ensure that data is always fresh. We have decided to have an atomic volatile variable 'gameOver' that defines if the game has finished. We choose to implement our design in this way because as multiple threads could finish at the same time, we have it so that this value can only be updated once and if another thread attempts to update it then they are put in a waiting state until all other threads have been notified that the game is over. This ensures that multiple threads will not assume that they have one and compromise the game.
CardGame	Class to simulate game	CardGame is a static class, thus providing static global variables that allow us to store game information.
Player	Player class implements runnable instead of extending thread	We designed our player class in this way to increase encapsulation. As if we were to extend the thread class, we would be unable to also extend the playerOutput abstract class that has functionality for writing to a file. Therefore, we choose to implement Runnable to maintain this feature.
CardDeck	A class to represent the Deck of cards next to a player	The CardDeck class represents deck objects that are to the left and right of players. These hold cards that are waiting to enter a hand and allow the movement of cards between players. Without these deck objects the simulation will not run.

ECM2414 Report

Design Choice and Reasons with respect to tests using JUnit4.13.2 framework

Component In Solution	*Design choice*	*reasons*
All components	We changed methods within many of the classes to be public	To create unit tests that could interact with each method separately we had to make them public.
All components	With respect to our design, we use the @Before notation to set up mock objects in all of our tests and the @After notation to tear down the mock object after a test is run.	We do this to maintain the clarity of our tests because repeatedly creating the same objects within our test methods reduces the overall readability of each individual test.
Players	Within our PlayerTest class, we have designed unit tests for methods within the Player class.	We have separated out the tests in this way so that each test method focuses on a particular feature of the player class. By doing this we aim to ensure nothing is overlooked in our testing.
Players	A 'testIsWinningHand' method which constructs a winning and losing hand and asserts that the method either passes or fails accordingly	We choose to create this test as if the isWinningHand function was performing incorrectly it would impact which player is being declared the winner of the simulation.
Players	'testAddToHand', 'testRemoveCard', and 'testCardsToArray' functions.	These tests were incorporated into our design because they provide checks on frequently called methods within 'Player'. If these methods were not performing as expected, such as removing multiple cards from a hand in a single call, this would impact the player's ability to win the simulation.
CardDeck	We have a test for returning the top card of the deck	This test is crucial because our simulation relies on players removing the card from the top of the pile to their left and placing it on the bottom of the pile to their right. If the remove card function is drawing any other card from the pile to their left then the simulation would be running incorrectly.
CardDeck	We have a test for checking the deck is empty and that a card is added to the back of the queue.	Our method for checking a deck is empty is tested. In the case that it's not functioning correctly, it will cause an exception in the program because a player was trying to access a non-existent element. Moreover, we also check that a card is added to the end of the queue when 'addToCardDeck' method is called so that it reflects a card being added to the bottom of a deck.
CardGame	We have a test for checking that our getPlayerCount function handles non-valid inputs.	We created this test to check that the function would return false for the inputs -5 to 0 and return true for the inputs 1 – 5. We also check that function returns false for any input that is not an integer.

ECM2414 Report

CardGame	Our test for distributing cards to the players and decks first checks that each player and card-deck has zero cards in their hand or deck before distribution. It then checks after the distribution each player has four cards and that these four cards have not been consecutively distributed to each player. It also checks that each deck has not had cards consecutively distributed to it.	We created this test to ensure that a player has an empty hand on their creation and after distribution, they have four cards. We also check that decks are created empty and that cards are not distributed consecutively to a player or a deck.
CardGame	We have a test that assigns mock players' decks and then asserts that each left and right deck of the player has the correctly assigned Id.	We choose to create this function to ensure there was no error when assigning players decks. If there was an error this could impact where players are placing and drawing cards from, subsequently ruining the simulation. These decks are crucial for the circulation of cards, and subsequently the ending of the game.
Pack	We have a test that generates a valid pack and runs a validity check (using the method "checkPack") on the pack to ensure it returns true	We created this test to ensure our "checkPack" method works correctly and a valid pack does not cause an exception.
Pack	We created a test that checks a pack only asserts true on a pack designed for 5 players. The test runs a validity check on the same pack with players ranging from 0 to 10	We created this test to ensure that only a pack designed for the right number of players is cleared as a valid pack.
Pack	We designed different tests that check how the program handles packs that contain either a sentence and, negative (or zero) values	We created this test to ensure we had reliably tested the "checkPack" method with erroneous inputs to see if it had the correct checks in place to deal with the input and not cause an exception. If an incorrectly formatted pack was to enter the simulation there would be a cascade of errors
FileOutput	We designed a test that would write a string to a designated test file using the writeToFile method located in FileOutput, then read the test file and assert the input matched the string	We made this test to ensure that any string being passed into the writeToFile function was being correctly outputted to the file.
Card	We create a new card in the @Before notation and then have a test that calls the returnFaceValue method and checks the return matches a predefined value	We wanted to ensure that the return of the method was the same as our expectation. In the case that it was not, this would reflect a major flaw in the Card class that would impact the simulation.
CardDeck	We have designed a test that checks if an output file for a deck has been created. This file simply	The specification describes how the simulation outputs the final decks to output files. Therefore, it is important that we ensure these files are being created and are

	uses the exists() built-in function found with the File module.	unique to each deck. This is ensured as each file is named after the unique object it is created for.
Player	We have designed a test that checks if a player's output file has been created. We again have used the inbuilt function exists().	The specification details how a player's actions must be recorded in a file. This file is crucial for examining the running of the simulation, and therefore must be validated for its existence. If this was to fail then a player would not be able to record their results.
CardGame	We have designed a test for the interrupting of our threads. The test creates 3 new threads and assigned them to the same thread group we use in the actual running of the simulation. When the threads start, they are made to immediately wait. We then call the endgame method which interrupts the thread group; we assert that all the threads were correctly interrupted.	We created this test to ensure that calling the endgame method stops all of the threads in the given thread group. During the simulation, we require the observer thread to be interrupted from its waiting state so it can write the final deck outputs, and thus if this test failed it would indicate that the threading interruption was not reliable enough to use and the final decks would not be written.