ECM2414 Cover Page

Student ID	710008606	710003076
Weight	50%	50%

Date	Time	Duration (h)	Role	Signed	Role	Signed
01/11/22	14:00	5	Driver	710003076	Navigator	710008606
05/11/22	11:00	4	Navigator	710003076	Driver	710008606
08/11/22	17:00	4	Driver	710003076	Navigator	710008606
12/11/22	11:00	4	Navigator	710003076	Driver	710008606
13/11/22	15:00	3	Driver	710003076	Navigator	710008606
15/11/22	17:00	5	Navigator	710003076	Driver	710008606

Design Choices

We decided to implement checking the validity of a pack file through a flag in each Pack object. This allowed us to contain the validation of the file within the Pack class and notify the user of all invalid cards.

Each player and deck needed to hold multiple cards. As the number of cards temporarily fluctuates during a player's turn, we use an ArrayList to store the cards which can handle variable length data.

Each sensitive variable in each class was made private and encapsulated so that the getting / setting of the variables could be controlled.

Additionally, the deck objects are accessed by multiple threads during the execution of the program, we synchronized the getter / setter methods to prevent unexpected exceptions and ensure the objects could only be accessed by one player at a time.

To allow the players to play simultaneously, we created a thread for each player and allowed them to run in unison. The players do not need to take turns independently, as each player writes to a unique deck and reads from a unique deck. This means that read-modify-write and check-then-act race conditions will not occur. Deadlock is also not possible as two synchronized blocks are not entered simultaneously.

To notify the other players of a victory, a static winning flag is changed for the Player class, which is checked at the beginning of each turn. We decided not to use interruptions to notify the other threads to prevent unexpected termination whilst writing to log files or another important event. This ensures that player actions are fully atomic and will never terminate whilst in progress. This design choice means that two players could rarely win simultaneously if both obtain a winning hand at the same time, however our program runs without error in this event.

Each player is responsible for logging their actions, as this is implemented in the Player class. Each logging event independently opens and closes the relevant log file to ensure that partial log files are still generated in the event of unexpected termination. Additionally, as there are the same number of decks as players, each player notifies the deck to their left at the end of the game by calling the deck's logging function.

To handle the case where a player is dealt a winning hand immediately, the pack object checks if any player has won before the player threads are launched. This implementation stops the players taking any turns if a player starts with a winning hand.

Design Choices for Testing

For testing we decided use JUnit 4.13.1 as the documentation was more mature.

In general, for each class, we test that you can create an instance and access/modify certain data through public functions. The complexity of the tests varies as some classes were very basic (i.e. Card) whilst others had far more functionality to test (i.e. Player). We test each function with standard inputs and their respective edge cases to ensure maximum coverage. Each class has its own Test class, and all of these can be run from the CardGameTestSuite, as shown in the README.

For testing the Card class, we verify that a card can be created and that the value of the new card is the expected value.

To test the CardDeck class, we first ensure that dealing a card increments the length of the deck, and then confirm the new card's value was expected using the appropriate getter method. CardDeck also handles logging a deck's contents to an output file when the game ends. We test that the logging method successfully creates/writes to the correct file and ensure the contents are correct. We test the logging for both empty and non-empty decks.

The Pack class handles the creation, validation, and dealing of the initial pack of cards, which is read from an input file. We test Pack's handling of invalid cards in the user input pack. We test for the wrong number of cards for the number of players, as well as negative numbers and non-numeric inputs. We also test that all players and decks are dealt four cards each from a valid input pack.

Testing the Player class involves invoking private methods with reflection and testing basic functionality like dealing cards to players. The strategy of each player is to ensure that cards with the same value as the player's ID are not disposed; we use the findDisposableCard method to find unwanted cards to remove. We test the private method findDisposableCard by giving a player three desired cards and one card of another value, and ensure the player successfully discards the unwanted card. We also make sure that once the player has four matching cards, they successfully inform the other players so that the game can finish. We also verify that a player can win with four matching cards with a value other than the player's ID.

We use reflection again for the CardGame class to test that an input pack file is read successfully using the private method getPackContent. As the program is multi-threaded and requires every player to interact with shared data in a thread-safe manner, we also simulate a full game with the fullGameTest method. This test provides an input pack for a random number of players and ensures the game terminates without throwing an error or hanging indefinitely.