ECM2414 Software Development

Cover Page

Agreed Split: 50/50

**Development Log**

1. 09/11/2023, 11:45-1:45

Driver: 720041178

Observer: 710036960

2. 10/11/2023, 10:30-1:30

Driver: 710036960

Observer: 720041178

3. 13/11/2023, 12:30-3:00

Driver: 720041178

Observer: 710036960

4. 17/11/2023, 9:45-10:00

Driver: 720041178

Observer: 710036960

5. 17/11/2023, 10:00-1:00

Driver: 710036960

Observer: 720041178

6. 21/11/2023, 6:00-8:00

Driver: 720041178

Observer: 710036960

7. 23/11/2023, 6:00-8:30

Driver: 710036960

Observer: 720041178

8. 27/11/2023, 1:30-3:30

Driver: 720041178

Observer: 710036960

9. 28/11/2023, 3:30-5:30

Driver: 710036960

Observer: 720041178

Design Choices

A screenshot of a computer program

Description automatically generated

The CardGame program is a multi-threaded program that simulates a game between 4 players. It consists of each player repeatedly discarding and drawing cards from their hand and a connected deck respectively. The aim of the game is for a Player to reach a Hand where all 4 cards have an equal face value.

Before the game begins, a player must input the number of players, and an input pack. The input pack is a text file consisting of 8n card values, where n is the number of players.

The code was divided into 5 files (including our custom Exception, and excluding our nested CardHand class), in efforts to improve readability and encapsulation.

**CardGame**

CardGame is our main class, and the entry point for the program. It contains the majority of game functionality. CardGame contains methods that set up the game and instantiate objects ready to be used in our game when the threads are finally run. For example, our initialiseAllGroupObjects() method instantiates our Players ThreadGroup - used to consecutively interrupt all of our threads once the game is won by a Player -, our Hands, and our Decks.

**Card**

Card is a data class used to store data about our cards. It includes data on the Card’s face value and its round count, which is a counter used to determine how long the card has been in the hand, put in place so that a card does not spend too long inside someone’s hand without being passed around.

**CardDeck**

CardDeck is a data class, used to store data about our card decks. It inherits the generic class LinkedBlockingQueue, as we thought that using a Queue data structure would be perfect for the task, as it allows us to quickly and efficiently remove cards from the head and place them at the tail.

Another useful functionality of LinkedBlockingQueue is the ability to block requests until the queue is not empty. This allows us to let our threads individually run as fast as they want without throwing a NullPointerException when one of the decks has no cards left, improving performance.

In addition to this, CardDeck has methods allowing it to print the “deckn\_output” files.

**Player**

Player is our primary class that runs our simulation, containing the majority of game functionality. It inherits from Thread, allowing it to run parallel to itself. It also contains the nested class CardHand, nested in order to provide encapsulation to the data that is in our hand.

We use a StringBuilder called gameUpdateStream to store the logs of the game. Doing this means that our thread does not write to the logfile every time we loop, improving performance and lowering the likelihood of IOExceptions.

As the Players class must be accessed by the concurrent threads, there is a very low likelihood, but a likelihood nonetheless, of two threads being ran on different cores, thereby using a separate cache. Because of this, our winningPlayerID variable is made volatile, so that we ensure the data is correctly obtained. No other variables are made volatile, since they are all constants.

In our run() method, we synchronize our win-checking section. This is because without doing so, two threads may end up trying to win at the same time, and this would lead to two player logs both saying they “won”, when they did not. Synchronising this section solves this issue.

**CardHand**

CardHand is a nested class stored within Player that stores data on the Player’s current hand state. It contains variables leftDeck and rightDeck, used as a very rudimentary form of doubly-linked list that allows us to connect our Players and CardDecks together using the Decks as a middleman.

It inherits the generic class ArrayList with type <Card>, specialised with new methods that allow for checking of hand equality, searching for optimal card removal and atomic drawing from and to the left and right decks respectively.

Test Design

Overview

For the purposes of testing, JUnit5 was used. Our code achieved a ??% of method and ??% of line coverage in CardGame, and a ??% of method and ??% of line in Player. To achieve this calculation, we used IntelliJ’s built-in code coverage tool.

Testing was divided into X classes to improve readability.