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

10. 29/11/2023, 12:00-3:30

Driver: 720041178

Observer: 710036960

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, JUnit4 was used.

Our code achieved:

* 100% of method coverage and 80% of line coverage in CardGame…
* …and 87% of method coverage and 91% of line coverage in Player.

In order to calculate this, we used IntelliJ’s built-in code coverage tool.

Testing was divided into 7 classes and 1 interface (used as a category holder) to improve readability, with 2 of said classes being categorized into ThreadTests for ease of navigation.

All testing is facilitated through a single TestSuite class that includes all to-be-mentioned testing classes.

**TestSuite**

This testing suite was created in order to facilitate the simultaneous testing of all methods and classes in our program. Due to the fact that there is not a large quantity of classes in our program, we didn’t see it necessary to create multiple test suites.

The TestSuite tests every single class, and includes the ThreadTest category to indicate Player tests.

**ThreadTest**

This category was created to connect the two different Player testing classes. Since we used JUnit 4, we didn’t have access to the @Nested annotation, meaning we couldn’t simply nest the tests within one another. Therefore, we created a category.

**CardTest**

To setup CardTest, the class uses a Before method to setup a Card object with a value of 1.

To test whether the value is correctly passed on, we assert that calling getValue on our Card object is equal to 1.

To test whether our roundCount is correctly set, we assert that calling getRoundCount on our Card object is equal to 0.

To test whether incrementing the round count works properly, we assert that after incrementing the round count, calling getRoundCount gives us 1.

To test whether resetting the round count works, we increment the round count, then reset it, making sure the round count is back to 0.

After each test is complete, we return the card to being null.

**CardDeckTest**

To set up CardDeckTest, the class uses a Before method to setup a cardDeck object using a set of Cards.

To test whether our file is created correctly, we manually delete the file, then instantiate a new CardDeck, and check if the file has been recreated.

To test whether toString is working, we assert that calling toString on our object created with the Before method yields the correct response, both with and without data.

To test that printing the log file works properly, we call the printLogFile function, then read the contents, asserting that the output is equal to our expected value.

After each test is complete, we set all used variables to null.

**PlayerTest**

To set up PlayerTest, the class uses a Before method to both setup a dummy ByteArrayInputStream object – which is used simulate user input to the CLI - and start the CardGame main() method.

To test whether we discard the correct card, we first let the CardGame run its course. After this has finished, we read “player1\_output.txt”, and read the first 3 lines.

Because our input pack is always the same – reading from “input\_testing.txt” – input.txt should always try and discard from the same pool of cards every time, leaving it down to its choice, which should be 2. Due to threading complications, however, we were regrettably unable to test whether it would correctly choose a higher round count over a closer non-one card.

To test whether our game finishes correctly, we once again let the CardGame run its course. After this is done, we read each file, and look for the keyword “wins”, which should only be in one of the files. If it is in multiple files, the test has failed, and if it is in none of them, the test has also failed. The desired outcome is that the “wins” keyword only appears in one file.

After each test is done, we tear down the CardGame class’s static variables to start anew.

**PlayerWinsInstantlyTest**

To set up PlayerWinsInstantlyTest, we instantiate a set of Cards, each with the face value of 1. We use this set of Cards to instantiate a Player with hand {1, 1, 1, 1} and two CardDecks with decks {1, 1, 1, 1} (though this is mostly just so that the Player runs at least once). We then call the player’s run() method, and locate the file that should have been created by it, assigning a BufferedReader to it.

To test whether the winning thread prints properly, we read from the file, and check that the result is not empty or null.

To test whether the game ends as expected, we create two variables – a StringBuilder builder and a String expected. We read 4 lines from the winning player’s logfile and check if the result is equal to the expected String.

To test whether toString works as expected, we assert that the first line of the file reads the same as our expected string.

After each test is completed, we close the reader and null it, removing it from existence.