ECM2414

50:50 split

Development Log:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Date | Start | Duration | Roles | Signature 1 | Signature 2 |
| 8th Nov, 2023 | 13:25 | 2hrs 20mins | Both drivers and observers | 720020735 | 720057498 |
| 15th Nov, 2023 | 12:50 | 1hr | Both drivers and observers | 720020735 | 720057498 |
| 19th Nov, 2023 | 11:00 | 1hr 15mins | Both drivers and observers | 720020735 | 720057498 |
| 24th Nov, 2023 | 19:00 | 2hrs | Both drivers and observers | 720020735 | 720057498 |
| 26th Nov, 2023 | 11:35 | 3hrs | Both drivers and observers | 720020735 | 720057498 |
| 27th Nov, 2023 | 19:20 | 2hrs 10mins | Both drivers and observers | 720020735 | 720057498 |
| 28th Nov, 2023 | 19:15 | 2hrs | Both drivers and observers | 720020735 | 720057498 |
| 29th Nov, 2023 | 14:00 | 1hr 30mins | Alex driver, Dan observer | 720020735 | 720057498 |
| 29th Nov, 2023 | 18:40 | 2hr 20mins | Dan driver, Alex observer | 720020735 | 720057498 |
| 29th Nov, 2023 | 22:00 |  | Documentation | 720020735 | 720057498 |

Code Design

**The Classes**

­To represent the situation, we used four main classes:

* The Card class, which represented each card in the game. By making these objects rather than integers we could tie the ‘staleness’ attribute to them, which we used to track how long a card had remained in a player’s hand.
* The Deck class, to represent the decks. To avoid deadlocks, we implemented these as three piles- this is explained in the *Avoiding Deadlock* section below. However, this detail was hidden from the Player objects through encapsulation- the Deck class provides methods for taking cards from and giving cards to the deck as a whole.
* The Player class, to represent players. Player objects can be run as threads, unlike the Deck objects, meaning they are responsible for driving card movement and ending the game upon victory.
* The CardGame class, which handles setup and cleanup for the game. It gets the necessary user input, creates the card, deck and player objects, then starts the player threads. It also handles interrupting the player threads when the game has finished.

A diagram of a deck

Description automatically generated

**Avoiding Deadlock**

Before starting to program our solution, we first considered how the threads would operate in tandem, and what situations might cause deadlocks. The principal issue we identified was the need for draw and discard to be atomic- this meant a player would need the monitor for both of its adjacent decks for this action, which could lead to deadlock establishing (if each player gets the monitor of the deck to their left, then waits for the one on their right, for example).

To avoid this, we opted to make each deck consist of three lists, with two daemon threads moving cards between them. That way, a player would only need to secure the monitor for the out pile of its draw deck and the in pile of its discard deck, which no other player would be competing for. The diagram below shows the flow of cards through this structure:

A screen shot of a computer

Description automatically generated

The in and out daemon threads spend most of the time waiting on the monitors for the in pile and out pile respectively. When a player discards a card to the deck, it notifies the in daemon thread, which will then move the card to the mid pile. Similarly, if a player finds the out pile empty when preparing to draw it notifies the out daemon thread, which will move any cards in the mid pile to the out pile. In the case there are no cards in the mid pile, it will instead wait on the mid pile’s monitor, and be notified when the in daemon next moves a card.

**Finishing the Game**

When a player determines it has a winning hand, it will call the *finish* method of CardGame with its index as a parameter. This method then sets the *winnerIndex* attribute of CardGame and interrupts all other player threads, which causes them to write their final messages to the output files and terminate. As player threads cannot detect an interrupt during their *draw* and *discard* methods, they are guaranteed to have four cards when they last write to the output files. When writing these final messages, threads call *getWinnerIndex* on CardGame to find the winner. As this attribute is only accessible by a public get method, there is no issue with multiple threads potentially accessing it at the same time. Finally, CardGame will call *printDeckContents* on all decks, making them interrupt and terminate their daemons and generate their output files.

**Performance Issues**

As the out daemon is only called when the out pile is empty, it causes the players to wait, slowing the program. However, we could not see a way to lessen this without either significantly more overhead, or more collisions between the two daemons. The file writing of players is also slower than necessary in places where lines are written with separate calls of *writeToFile*- this requires the file be closed and opened multiple times. However, this made the code much more readable, so we judged it a worthwhile trade-off.

**Test Design**

**Test Planning**

When designing our system, we planned how and when testing would be done:

* Firstly, we felt it was important to write all classes with simple methods that wouldn’t require any synchronization. This included implementing all getter methods for private attributes that were required in other classes – e.g. getN(), getStaleness() etc. As well as this, methods like giveCard(), isEmpty() and more were written, due to their independence from synchronization blocks.
* We tested every function using Junit 4.10. This was to ensure we had a well-structured system with a clear plan of how all features would be linked to one another.
* During testing, as most of the functions utilized other parts of the code, there was little need for input validation. The only usage was for checking the number of players a user entered and the pack location the user entered.
* We identified that users may not include the file extension on their file inputs, therefore decided to handle this accordingly by appending .txt to the file name entered where appropriate.
* After all tests were passed, we then finished the source code by implementing multithreading.
* Then, we rigorously tested all new functions added, ensuring that all tests were passed.

**Testing Inputs and Outputs**

The CardGame methods we tested involved interaction with the user through the terminal, which is not possible when running unit tests. To replicate this interaction, we replaced the System.in and System.out streams with our own input/output streams. However, this solution was not perfect- due to the way we used scanner objects in the CardGame class, we were unable to give more than one input. Therefore, we tested for the NoSuchElementException which would be generated if a method asked for input multiple times as a way of confirming that it did ask again if incorrect data was entered.

A computer code on a white background

Description automatically generated

We also experienced issues with reading the output; the string we retrieved from the output stream was formatted in such a way that we could not use the String.equals method to check it. Instead, we used the String.contains method in order to check the right error message was given and checked the formatting of the output outside of the programmed unit tests by running the methods. This way, we can be confident enough that test correctly assesses the output.

A close-up of text

Description automatically generated

A close-up of a message

Description automatically generated

**Mock Objects**

During the testing of Player, we identified that the Player class required two variables of type Deck to be passed to the constructor to initialize a player. Due to decks complexity, we implemented these as mock objects instead. We made the MockDeck class nested and private, to ensure there was no interference with other parts of the code.

MockDeck extends deck, which allowed MockDeck objects to be used as Deck objects by a player. However, we overrode the implementation of the player-facing methods to make them simpler by ignoring the three piles originally used in the Deck objects. We also stored a reference to the internal storage of the MockDeck before casting it to Deck, allowing us to bypass the encapsulation to confirm if tests succeeded.

A screenshot of a computer code

Description automatically generated

**User Acceptance Testing**

Due to the complicated and random nature of the program, we decided that user acceptance testing would be integral to observing the success of the system. Unit testing wasn’t deemed appropriate for being the only method of testing of the system; therefore, we compiled and ran the program multiple times.

After checking the logs, we could clearly see how each player had moved throughout the game, observing which cards had been discarded and which cards had been drawn. This was a clear indicator that the game was functioning correctly.

For one of the packs, different winners could be produced, due to the nature of how the game works. Due to the non-deterministic nature of the program, players could take their turns at a different rate to previous test, thus producing different winners as they would receive their winning hand slower than another player.