***Cover Page***

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Student Number | Candidate Number | Allocation |
| Benjamin Finch | 710005114 |  | 50 |
| Patryk Biegalski | 700037440 |  | 50 |

***Development Log***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Time** | **Driver** | **Observer** | **Ben’s Signature** | **Patryk’s Signature** |
| 27/10/22 | 12:25 – 13:25 | Ben | Patryk |  |  |
| 28/10/22 | 17:30 – 20:00 | Patryk | Ben |  |  |
| 05/11/22 | 13:00-15:00 | Patryk | Ben |  |  |
| 05/11/22 | 16:00-18:00 | Ben | Patryk |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

***Documentation – Source Code***

**Card Class**

Our Card class is a class which stores an integer value. The reason we did not use integers without a wrapper class is to protect our code from using the values in incorrect ways. In the game, cards cannot be added, subtracted etc., and should only be stored in data structures intended to store them, such as the “pack” variable in CardGame or the “hand” variable in Player.

**Player Class**

Design:

Players all have an identifying integer Id and a hand of 4 cards (of type BlockingQueue<Card>). Players will carry out three actions, checking if they have won, drawing from their draw deck or discarding to their discard deck. Draw and discard decks are calculated using the Ids of the players and decks.

The player draws from the deck with a matching Id and discards to the deck with the Id one more than their own; except for the last player, who discards to the first deck. In order to facilitate asynchronous interaction with each other, the Player class contains a static ArrayList<Player> attribute which contains all created player objects.

Threading:

Our player class implements the Runnable interface in order to make use of Java’s Thread class. Keeping with the specification, we designed our code to be as thread safe as possible. We did this by using a BlockingQueue to store cards, as it is already a thread-safe data structure. We also implemented try-catch blocks to catch interrupt exceptions so that players would stop playing if they are interrupted.

Hand:

The players hand should hold exactly 4 cards at any time, we achieved this by synchronizing the drawCard() and discardCard() functions. When discarding, the player keeps any cards which match their Id, and when all cards match, the player updates the static “winner” attribute with their player Id. As part of this, players should make sure that no cards except their preference stay in their hand forever, to this end, the BlockingQueue enables the players to automatically cycle their cards so that all necessary cards will (at most four turns after being drawn) be discarded.

When the player discards a card, they check the front of their hand. If the card if their preferred type, they add it to the back of their queue. If the card is not their preferred type, the add it to their discard deck and draw a new card. In theory, this could lead to a situation where a player checks their already winning hand for a card to discard. However discardCard() can only be executed if the hand is not a winning hand, therefore this situation is unlikely, if not impossible to occur.

Output:

The Player class should write updates to an external text file to document the game running. To achieve this, we used the BufferedWriter class. BufferedWriter writes text to a character-output stream, buffering characters in order to provide for the writing of arrays, lists and strings. BufferedWriter also overwrites duplicate files, which simplifies repeated testing substantially.

**CardDeck Class**

The CardDeck class is a wrapper class for a BlockingQueue<Card> data structure, that also outputs data to an external text file. We decided to make the decks a separate class because it would both make it easier to make changes in CardGame and Player, without affecting the behaviour of the decks and make the system easier to visualise when implementing the game loop.

CardDeck uses the same BufferedWriter class as Player to output the contents of the decks to text files.

**CardGame Class**

CardGame uses a main() function to play the card game, then uses extra functions to simplify main(), the goal being to make main() look as self-explanatory as possible by outsourcing any sections of complex code to their own small, self-contained (and easily documented) function. The exception to this is the section of main() which takes the user input. This was left in main() to simplify the scope of the variables storing the player inputs.

The main() function asks for number of players (n E N) and a list of card values (the size of which should be 8n) before generating the required number of Player and CardDeck objects, before distributing Card objects to each Player and CardDeck then starting all the Player threads. Note that, because decks are assigned to players during player initialisation, CardDecks are generated first; however, players are still dealt to first in keeping with the specification.

By starting the Player threads as the last step of main(), main() does not execute any code while the Player threads are running. This is intentional as it reduces the amount of code running simultaneously and therefore maintains orthogonality, since any change in the Player’s run() method do not affect the execution of main() and changes in main() (other than any which prevent Player threads starting) do not affect the execution of run().

***Documentation – Testing in Junit 4***

**TestCard Class**

|  |  |  |
| --- | --- | --- |
| **Test Name** | **Query** | **Design** |
| testCompareEqual() | Do cards with equal values return 1 when compared? | Create two card objects with different values. If compareTo() returns 1 in both directions, the test passes. |
| testCompareUnequal() | Do cards with unequal values return 0 when compared? | Create two card objects with different values. If compareTo() returns 0 in both directions, the test passes. |

**TestPlayer Class**

|  |  |  |
| --- | --- | --- |
| **Test Name** | **Query** | **Design** |
| testConstructorDraw() | Do players choose the correct draw pile? | Create a Player with playerId = 1 and a cardDeck with deckId = 1. If the Player is assigned the cardDeck as their draw pile, the test passes. |
| testConstructorDiscard() | Do players choose the correct discard pile? | Create a Player with playerId = 5 and a cardDeck with deckId = 6. If the Player is assigned the cardDeck as their discard pile, the test passes. |
| testDiscard() | Do players discard the first card in their hand if it isn't the preferred card? | Create a Player with playerId = 1 with a hand 2345. If the hand after executing the discardCard() function is 345, the test passes. |
| testDiscard2() | Repeat testDiscard() with playerId = 2. | Create a Player with playerId = 2 with a hand 1234. If the hand after executing the discardCard() function is 234, the test passes. |
| testDiscardCycle() | Do players discard the second card in their hand if the first is the preferred card? | Create a Player with playerId = 1 with a hand 1234. If the hand after executing the discardCard() function is 341 the test passes. |
| testDiscardCycle2() | Repeat testDiscardCycle() with playerId = 2. | Create a Player with playerId = 2 with a hand 2345. If the hand after executing the discardCard() function is 452 the test passes. |
| testDiscardFullCycle() | Do players discard the last card in their hand if the first three are the preferred card? | Create a Player with playerId = 1 with a hand 1112. If the hand after executing the discardCard() function is 111 the test passes. |
| testDiscardFullCycle2() | Do players discard the last card in their hand if the first three are the preferred card? | Create a Player with playerId = 2 with a hand 2221. If the hand after executing the discardCard() function is 222 the test passes. |
| testWinningAtStart() | Do players declare victory if they start the game with a winning hand? | Create a Player with playerId = 1 with a hand 1111. If the static variable winner is changed to 1, the test passes. |
| testWinningAfterStart() | Do players declare victory if they draw a card and get a winning hand? | Create a Player with playerId = 1 with a hand 2111 and a cardDeck with a single (1) card. If the player declares victory with a hand 1111, the test passes. |
| testLosingAtStart() | Do players concede defeat if another starts with a winning hand? | Create a Player with playerId = 1 and hand 2222, an empty cardDeck, and set winner = 2. If the player does not try to take a card, the test passes. |
| testLosingAfterStart() | Do players concede defeat if another obtains a winning hand? | Create a Player with playerId = 1, a hand of 2222, a cardDeck with 3333, and after one second set winner = 2. If the player does not try to draw from an empty deck, the test passes. |

**Test CardDeck Class**

- Test that outputs are correctly formatted

**TestCardGame Class**

|  |  |  |
| --- | --- | --- |
| **Test Name** | **Query** | **Design** |
| testMainEmptyPack() | Does cardGame refuse an empty pack? |  |
| testMainFakePack() | Does cardGame refuse a non-existant pack? |  |
| testMainSmallPack() | Does cardGame refuse a pack with too few numbers? |  |
| testMainLargePack() | Does cardGame only take the require numbers from a list with too many numbers? |  |
| testMainInvalidLinesPack() | Does cardGame refuse a pack with numbers on the same line? |  |
| testMainZeroPlayers() | Does cardGame refuse to play with zero players? |  |
| testMainNegativePlayers() | Does cardGame refuse to play with negative players? |  |

* Does the class deal out cards correctly even when it thinks there are the wrong number of players?