***Cover Page***

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

***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 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

***Documentation – Source Code***

**Card Class**

Our Card class is a class which stores an integer value. The reason we did not just 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**

Basics: Players all have an identifying integer Id and a hand of 4 cards (of type PriotiryBlockingQueue<Card>\*). Players will carry out two actions, 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 without making Player a subclass of Thread. This is due to Java only supporting single inheritance. Thus, using interfaces where able seems like good programming practice. Keeping with the specification, we designed our code to be as thread-safe as possible. We did this by using a PBQ to store cards, as it is already thread-safe.

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, therefore the PBQ enables the players to automatically cycle their cards so that all cards will (at most four turns after being drawn) be discarded. [Insert explanation of how the player keeps preferred cards]

Outputting: The Player class should write updates to an external text file to document the game running. [Explain what we used to obtain the output].

* Should draw from their deck and deposit on the next player’s deck (unless the draw deck is empty)
* Declare victory if either their starting, or subsequent hand are all the same number
* Concede defeat if either an opponent’s starting, or subsequent hand are all the same number.

**CardDeck Class**

The CardDeck class is a wrapper class for a PBC data structure, that also outputs data to an external text file. We decided to make the decks a separate class because it would make it easier to make changes in CardGame and Player, without affecting the behaviour of the decks.

**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 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.

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***

**TestCard Class**

**TestPlayer Class**

* Do the player objects draw from the correct pile?
* Do the players deposit in the correct pile?
* Do the players discard the correct card?
  + Test different hand combinations
  + Test different player combinations
* Do players declare victory when they draw a new card?
* Do players declare victory when they start with a winning hand?
* Do players stop playing when another player draws a winning card?
* Do players stop playing when another player starts with a winning hand?
* Threading: do players wait for decks to be changed by another player before drawing/discarding?

**Test CardDeck Class**

- Does the deck correctly take in

**TestCardGame Class**

* Can the class deal with bullshit inputs
  + Empty deck
  + Too few numbers
  + Some numbers on the same line
  + Too many numbers
* Does the class generate the correct number of decks?
* Does the class deal out cards correctly even when it thinks there are the wrong number of players?

***Questions:***

Should players act simultaneously or as quick as possible?