**Cover Page**

|  |  |
| --- | --- |
| Student | Allocation |
| 700041827 | 50% |
| 710032484 | 50% |

**Development log**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Time | Duration | 700041827 | 710032484 |
| 08/11/22 | 14:30 | 2h | Observer | Driver |
| 09/11/22 | 12:00 | 2h 45m | Observer | Driver |
| 15/11/22 | 2:30 | 2hr 15m | Driver | Observer |
| 17/11/22 | 11:30 | 30m | Driver | Observer |
| 17/11/22 | 12:00 | 30m | Observer | Driver |
| 20/11/22 | 14:00 | 1hr | Observer | Driver |
| 20/11/22 | 15:00 | 1hr | Driver | Observer |
| 22/11/22 | 10:30 |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Production Code Design Choices**

Decks

* For the decks used in-game, we decided to develop a CardDeck class, which has all the functionality of a real deck.
* We realised that decks share much the same functionality as the hand of a player, so took all the common methods of the CardGame and Player classes, and grouped them into the CardHolder class, which is then inherited by CardGame and Player.

Interrupts

* We decided to use the built-in Thread.interrupt() method to interrupt the Player threads after the game ends.
* When a Player wins, handleWin() is called, which in turn calls interrupt() on every thread.

Players

* For the Player class, we decided to provide the instance variables **otherPlayers** and **thread**, for ease of access.

Cards

* The Card class has limited functionality, because all the functionality of any deck is provided by CardDeck, and the functionality of any player hand is provided by Player.

PackGenerator

* At the start of the project, we decided to make a small testing program which creates a random, valid deck for a given number of players.
* This made testing for differing numbers of players and different packs much easier.

CardGame

* As specified in the coursework sheet, CardGame is our executable class.
* To make testing easier, we added various try-except blocks to catch any filesystem errors.
* We also made sure that each individual block was working properly before moving onto the next (e.g. by using print statements to display the contents of an ArrayList)

Access Modifiers

* We decided to make all instance variables private as a standard, and use the protected modifier for the instance variables in CardHolder, to allow the use of them in CardDeck and Player methods.

Atomic Methods

* To ensure the program was easy to develop, we made sure each action which consists of multiple smaller actions was split into multiple methods.
* For example, the handleWin and informPlayers methods could be merged, however it is easier to read and maintain when they are separated.
* Another example is the run and takeTurn methods – while run could be one large method containing the entirety of takeTurn, the code is much more readable when they are two separate methods.

**Testing Design Choices**

CardGame Testing

* These tests were done with the Thread.sleep in the Player class set to a random time (up to 1000 ms), to ensure the program is threadsafe.
* As a result, some decks would end up with more cards in than others during and after the game, as some threads were faster than others.
* Any files are in the “Testing” folder
* “Input re-requested” – Input loop restarts, asking for the number of players

|  |  |  |  |
| --- | --- | --- | --- |
| Input Field | Normal | Boundary | Erroneous |
| Number of Players | The next input is requested: the input file for the pack. | -1 (Attempt 1)  Initially, the program accepted it; added a if-statement check (>= 0).  -1 (Attempt 2)  “Number of players must be greater than zero.”; input re-requested  0 (Same as above) | a  “Invalid number of players.”; input re-requested |
| Input file | (Number of players must be normal)  (4 players)  Normal 1.txt  (Generated by PackGenerator)  Trial 1:  Works as expected  Player 4 wins  Trial 2:  Works as expected  Player 2 wins  (10 players)  Normal 1.txt  (Generated by PackGenerator)  Trial 1:  Works as expected  Player 8 wins  Trial 2: | (Number of Players must be normal)  (4 players)  Boundary 1.txt  (Every line is a 1)  Initially, this caused the program to break, as the winning check only checked for 4 of the same card – they didn’t need to be the preferred denomination.  Player 1 output file:  player 1 wins  player 1 has exited.  player 1 final hand 1 1 1 1  Other player output files:  player 1 has informed player n that player 1 has won  No deck output; the decks were never populated.  Boundary 2.txt  (Every line is a 0)  The program runs forever, as expected.  No deck output; game doesn’t end.  (10 players)  Boundary 1.txt  Same as 4 players  Boundary 2.txt  Same as 4 players | (Number of players must be normal)  (4 players)  Erroneous 0.txt (Doesn’t exist)  “Couldn’t open pack file.”  Input re-requested  Erroneous 1.txt  (Has 8n+1 = 33 rows)  “Invalid pack file length.”  Input re-requested  Erroneous 2.txt  (Line 32=’a’; non-integer file row)  Initially threw error, but after a catch was implemented:  “Couldn’t read pack file: it has a non-integer value in it.”  Replacing a with a float (2.2) gave the same result.  (10 players)  Erroneous 0.txt  Couldn’t open pack file. Re-requested.  Erroneous 1.txt  (8n+1 = 81 rows)  Invalid pack file length. Re-requested.  Erroneous 2.txt  Line 80 = ‘Hello’  Couldn’t read pack file: it has a non-integer value in it. Re-requested.  (Wrong number of players for a pack)  Invalid pack file length. Re-requested. |

Other Testing

* For the other classes, we used testing suites.