**Cover Page**

|  |  |
| --- | --- |
| Student | Allocation |
| 700041287 | 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 | 1hr 30 | Driver | Observer |
| 22/11/22 | 12:00 | 1hr 30 | Observer | Driver |
| 22/11/22 | 13:30 | 1h 30 | Driver | Observer |
| 22/11/22 | 15:00 | 1h 30 | Observer | Driver |
| 23/11/22 |  |  |  |  |
|  |  |  |  |  |

**Production Code Design Choices**

**Classes**

CardDeck

* 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.

Player

* This class handles any player actions after the CardGame class starts the game. It repeatedly performs a single atomic takeTurn method until a player wins and informs others that the player has won.
* The takeTurn method also handles the majority of file output, as when they take their turn, each action is outputted to their personal text file.
* For the Player class, we decided to provide the instance variables **otherPlayers** and **thread**, for ease of access.
* We decided to store the two decks the player accesses as **left** and **right**, where the player takes cards from the left deck and discards cards to the right deck, as this allows easy access for the players to use the decks.

Card

* 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.
* It holds an integer value that is used to determine whether a player has won at any given time in the checkHasWon method.

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.
* The CardGame class handles the start of the game (dealing cards to the players and decks) then runs player threads, that continue the game.
* 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)
* We decided to use the. join() function of threading to effectively pause card game until all threads are finished running. This allowed us to execute final code such as outputting deck contents to text files.

**Other Design Choices**

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.

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.
* Having atomic methods allowed us to perform every action discretely and for us to maintain and comment code more effectively.

Synchronisation

* We ensured that every method which could impact the object if multiple threads were to use it simultaneously was given the “synchronized” keyword.
* This was to ensure the program is threadsafe.

Version Control System

* We decided to use a public GitHub repository and GitHub Desktop as our Version Control System.
* The repository link: <https://github.com/piklman/ecm2414_coursework>
* This allowed collaboration and file sharing very easy to effectively work on the specification.

Text

Description automatically generatedChecklist  
We used a checklist to plan what to work on next.

**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.
* Random sleep times means the threads are more likely to use the same data structure at the same time, so this is a more vigorous test.
* 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 input files are in the “test” 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 an 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:  Works as expected  Player 9 wins | (Number of Players must be normal)  (4 players)  Boundary 1.txt  (Every line is a 1)  Initially caused an error, but after fix player 1 instantly wins.  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)  Player 1 instantly wins  Player output file: same as Boundary 1, but with 0s.  No deck output; game doesn’t end.  (10 players)  Boundary 1.txt  Same as for 4 players, but with 10 output files  Boundary 2.txt  Same as for 4 players, but with 10 output files | (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. |

JUnit Testing

* We used JUnit 4.13.2 for our testing as stated in the README.txt file.
* We used testing suites to test the Player, CardDeck and CardHolder classes.
* Testing suites allows us to group multiple test classes to run as a single batch.
* Within these tests, we used mock objects to recreate different circumstances of the game, so we could test each method thoroughly and with different test-cases.

An example of trying different test-cases is in the testDiscardCard method in the TestPlayer class. Here we try to force the method to discard its preferred card value, when it’s the only card left in a player’s hand. This test shows us that the method is implemented correctly and it keeps the player’s card.

Another example of tests impacting our design choices is the testCheckHasWon method in TestPlayer. This method highlighted that it should be possible for a player to win with 4 of the same cards regardless of what values these cards have. In our initial design we only checked for preferred values, however we have now designed our game logic to check for this unlikely event.