Cover Page

Student ID	710004133	710008559
Weight	50%	50%

Development Log

Date	Time	Duration (h)	710004133 Role	710008559 Role	Signatures
07-11-22	12:30	2	Planning	Planning	029443 246654
10-11-22	11:40	1	Driver	Navigator	246654 029443
12-11-22	13:00	4	Driver	Navigator	246654 029443
14-11-22	14:00	3	Driver	Navigator	246654 029443
17-11-22	12:30	1	Navigator	Driver	029443 246654
18-11-22	21:00	2	Navigator	Driver	029443 246654
21-11-22	16:00	2	Report	Report	029443 246654
23-11-22	13:15	3	Report	Report	029443 246654

Design - Production Code

Classes

We created the three classes that the specification requires: a thread-safe *Card* class, a thread-safe *Player* class, an executable *CardGame* class. We also created an additional thread-safe *CardGame* class.

User Input and Validation

First we developed the *requestNumPlayers* and *requestPack* methods to get the user's input on the number of players, and the pack used in the game. These methods get called in main.

requestNumPlayers creates a new Scanner from Java's Scanner class and asks the user in the terminal to enter the number of players. The method checks that the value entered is a valid integer, and that the number of players entered is not below 2 as the game is not playable with less than 2 players. If these validation checks fail the terminal informs the user, and the method gets called again. The method sets the static field for *numPlayers* in *CardGame*.

requestPack also creates a new Scanner and asks the user to enter the location of the pack to load in the terminal. The method checks that the file path exists using Java's Path class, if it doesn't, the terminal will inform the user and call the method again, otherwise it will then check that the lines in the file are equal to 8n (n being the number of players), and again call the method again and inform the user if the pack is not valid. A new File object will be created, and another Scanner to read the contents of the file. We used a while loop to iterate over each line in the file, and the text on each line attempts to be converted into an int. If it cannot, or if the integer is negative, the user is informed and the method called again. A new Card object is created, passing the integer value from the line in the file to the constructor, and the Card is added to a Queue which acts as the pack. We chose to use a queue as it allows us to process elements in FIFO order, which is useful when dealing the cards.

Creating Players and Decks

We next decided to develop the *createDecks* and *createPlayers* methods in the *CardGame* class. These methods are called in main, and both instantiate the Deck and Player objects, and add them to ArrayLists in *CardGame* for later reference. The *createDecks* method uses a for loop to create a deck for each player in the game, passing an incremented deck number into the constructor each iteration. *createPlayers* does the same, but also passes the deck to each player's left and right, as well as the number of players into the constructor too. Due to the Topological relationship of the game players and card decks, the method also checks if the player being created is the last player, as their right deck will circle back to deck 1.

Drawing and Discarding Cards

We also chose to create *addCard* and *discardCard* methods in *Player*, and *addCard* and *drawCard* in *CardDeck* as we would need these for dealing the cards, and whilst playing the game. Both *addCard* methods take a Card object, and add it to the deck/hand ArrayList. *CardDeck*'s *drawCard* method get's the first card in the deck, then removes and returns it. These methods all use the synchronized block to avoid race conditions when accessing shared data.

Player's *discardCard* method implements the simple game strategy from the specification, in which each player will not discard a card if the value of it is the same as their player number. It creates a new *nonPreferredCards* ArrayList and loops through the player's hand and adds a card to it if it isn't the same value as the player's number. We then used Java's Random class to generate a random number and pick that card index from *nonPreferredCards*, remove that card from the player's hand, and return it.

Dealing Pack

We next developed the *dealCards* method in *CardGame* which is called in main after the players and decks are instantiated. It first deals the cards using the *addCard* methods to the players, then the remaining cards to the decks. We used two for loops to deal the cards in a round-robin fashion, one at a time, from the head of the pack queue. The first loop ensures 4 cards are dealt to each player/deck and the second is so each player/deck gets dealt one card at a time around the ring.

Win Condition

We developed the *checkWin* method which checks if a player has met the win condition for the game. It compares every card value in the player's hand to see if they all match, and returns true if they do.

Output Files

We developed the *createPlayerFile* and *createDeckFile* methods in *Player* and *CardDeck* using Java's File class to create a new text file with the name in the format player1_output.txt/ deck1_output.txt using string concatenation, then returns the file.

The *writeToPlayerFile* method in *Player* uses Java's FileWriter to append text to the player's file which are taken as parameters. The *writeToDeckFile* in *CardDeck* does the same thing, however as the deck files only need to have one line, it just takes the file as a parameter and writes the deck's contents to the deck file.

In the *Player* class we used a few methods to process data to the player's output file. *writeInitialHand* is used to write the player's starting hand of cards to their file, *writeTurn* is used to write the events of the turn to their file using *addedCard* and *discardedCard* as its parameters, and *writeWinner* and *writeNonWinner* are used to write the final lines to each player's file at the end of the game. *writeWinner* also declares it has won by printing a message to the terminal.

Threads

We decided to handle multi-threading by implementing the Runnable interface. The *createThreads* method in *CardGame* loops over the ArrayList for players and creates a new thread for each player, then adds these to an ArrayList of threads. In main, a loop starts each thread in the list of threads.

The *run* method in *Player* calls *createPlayerFile* and *writeInitialHand*, then calls the *checkWin* method to check if a player has won the game at the start of the game before the turns begin. If the game is not won immediately, we used a while loop to play each turn until a static AtomicBoolean field in Player had been set to true. Each turn the player draws a card from their left deck, and adds it to their hand, and discards a card from their hand, and adds it to their right deck. *writeTurn* is then called which writes the events of the turn to the player's file and each thread is then made to wait at a CyclicBarrier to ensure every thread carries out their actions on the same turn. The thread checks for a winner again, which if there is a winner the while loop will be broken, and the static *winningPlayer* field is set to the player number of the winning player. This is how we communicated to the other threads which player had won the game. The threads check if it is the winning thread and if it is, will call the *writeWinner* method and if not the *writeNonWinner* method. For each player's left deck the *createDeckFile* method gets called on it and then passed into the *writeDeckFile* method.

Design - Tests

Testing

To test the program we used the method of unit testing to test our code using **JUnit 4.13.2**. We made a test file corresponding to each class. Similarly to test each key method in the production code wrote a test for it in its respective test file. We did not test any get and set methods directly, as these were tested during the course of our other tests. In the src/packs/ folder there are a variety of card packs that we used during testing.

Testing Card

Method being tested	**Design choice and reasons**
Card constructor	We created a Card object with a new value, and checked that the *getValue* method returns the same value that was passed into the constructor. This is to ensure that a Card object was being instantiated properly, and with the correct Card value.

Testing CardDeck

Method being tested	**Design choice and reasons**
CardDeck constructor	We created a new CardDeck object with a value passed into the constructor, then checked that the *getDeckNumber* method returns the same value as the one passed in to ensure the object could be instantiated properly.
addCard	We created a new CardDeck and a Card object, and called *addCard* to add the card to the deck. We used getDeck to get the deckArrayList and *getValue* at the first index to make sure the card in the deck was the same as the card we added, and that the card was successfully added to the deck.
drawCard	We created a new CardDeck and Card object, added the card to the deck, got the size of the deck and then called the *drawCard* method. We checked that the card drawn was the same card as the card added, and also checked that the deck size was 1 element smaller than before to ensure that the card had been removed from the deck.
createDeckFile	We created a new CardDeck and called the *createDeckFile* method on it. We created a new Path object from the file created and checked whether the path existed or not to see if the file had been created or not.
writeDeckFile	We created a new CardDeck, then used *createDeckFile* and *writeDeckFile* with the created file and some test text. Then we checked that the length of the file wasn’t 0, which meant that the text had definitely been written to it and the file wasn’t empty.

Testing Player

Method being tested	**Design choice and reasons**
Player constructor	We created a new Player object and two deck objects which acted as the Player’s left and right decks, and passed in the number of players. We used *getPlayerNumber* to ensure the player number was the same as the one set, and *getLeftDeck and getRightDeck* to check that the decks were successfully assigned to the player too.
addCard	We created a new Player and a Card object, then used *addCard* to add the card to the player’s hand, then used *getHand* and got the first value in the hand to check that it was the same card as the card added.
discardCard	We created a new Player and two new cards, one of which had the same value as the player’s number and one that did not. We added these cards to the player’s hand and got the size of the hand. Then we used the *discardCard* method and checked that the card returned was the same as the card whose value wasn’t the same as the player number. We also checked that the size of the player’s hand had decreased by 1 after the *discardCard* method was called.
checkWin	We created a new Player, then added four new Card objects, all with the same value into the player’s hand. We then called the *checkWin* method to ensure it would return true, as four matching cards in a player’s hand means that they have won.
createPlayerFile	We created a new Player, then called the *createPlayerFile* method on the player. We then used the file to create a Path, and check that the file actually existed and was successfully created with the correct name.
writeToPlayerFile	We created a new Player, then created a file for the player. We then used the *writeToPlayerFile* method to write some test text to the file we created. We then checked that the length of the file wasn’t 0, which meant that the text had definitely been written to it and the file wasn’t empty.
writeInitialHand	While testing *writeInitialHand* we ran *writeInitialHand* on an empty file then checked that there was something in the file.
writeTurn	While testing *writeTurn* we ran *writeTurn* on an empty file then checked that there was something in the file.
writeWinner	While testing *writeWinner* we ran *writeWinner* on an empty file then checked that there was something in the file.
writeNonWinner	While testing *writeNonWinner* we ran *writeNonWinner* on an empty file then checked that there was something in the file.

Testing CardGame

Method being tested	**Design choice and reasons**
validateNumPlayers	We created a new CardGame and used the *validateNumPlayers* method, passing in the number 4. We checked that the method returned true, as 4 isn't negative. This made sure the method worked correctly.
checkFileExistence	We created a new CardGame and used the *checkFileExistence* method with a file that did not exist, to check that it would return False. We then tried it again with a file that did exist to see if it would return True.
checkIfNegative	We created a new CardGame and used the *checkIfNegative* method on the number -1 to check that it would return false. Then we tried it again with the number 1 to check that it would return true.
checkValidRows	We created a new CardGame and used *setNumPlayers* to set the number of players to 4. We then used one of our 4 player test packs to check if it had a valid number of rows using *checkValidRows* and that the method would return True.
createDecks	We created a new CardGame and set the number of players to 4, then used the *createDecks* to create 4 decks. We then used the *getDecks* method and checked the size of it to make sure it contained 4 objects.
createPlayers	We created a new CardGame and set the number of players to 4, created 4 decks and then we used *createPlayers* to create 4 players. We next used *getPlayers* and checked that it contained 4 objects.
dealCards	We created a new CardGame, set the number of players to 4, used a for loop and *addToPack* to add 32 cards to the pack, created 4 decks and players, and finally called the *dealCards* method. We then made sure that the size of the pack was 0 after dealing the cards to make sure they were all getting removed from it. We also checked that the size of a player's hand and a deck was equal to 4, so that we could confirm that each player and deck was getting dealt the right number of cards.
createThreads	We created a new CardGame, created 4 decks and players. Then we called the *createThreads* method and used *getThreads* to check that it contained exactly 4 threads.