ECM2414 Coursework Documentation

Duru Tiryaki & Taylor Roel

Production Code Design Explanation:

To determine how the program should be laid out, we started to consider what classes and objects we would use. The specification mentioned that we would require Card and Player classes.

Card Class

When designing the Card class, we thought about how the card would function in the game. Each card has only one attribute, an integer called `value`. Just like a real card, the value assigned at construction doesn't change (A '4' card won't become a '6' card later in the game, it will always be a '4' card), so this uses the `final` keyword to indicate the fact this value is a constant.

The only action the player does with the card alone (i.e not involving a deck) is looking at the card value (which is required to determine whether it should be discarded), so we only implemented a getter. This is the method `getValue()`, and it simply returns the card's value stored in the `value` variable.

Player Class

For the Player class, we decided this would be the class that would be our thread class. This is because when thinking about how the game would be played, the actions performed are initiated by the players themselves. Therefore it makes sense to make each player a separate thread performing their own actions concurrently with each other.

We obviously need a `playerId` attribute, as this is crucial to determine what card denomination they will prefer. Since this is also does not change after construction, we also use the `final` keyword.

We also have an attribute `hand`, which keeps track of which cards are in the player's hand. This is an ArrayList which stores instances of our Card class.

The attributes `takeDeck` and `giveDeck` are both intended to keep references to instances of our CardDeck class (which will be explained later, for now just know this represents a 'deck' in play), since the Player thread will need to know which deck to discard/add to.

`continueGame` and `hasWon` are flags used for interthread communication. The former is checked by the main thread so that it knows which Player has won, and the latter is modified by the main thread (via the `declareWinner()` method which passes the winning player's id for use in the output file) to let non-winning Player threads know that another player has won and to stop executing game actions.

And finally we have `outputFileContents` which is a list that will contain strings that are due to be written to the .txt file that the Player object outputs at the conclusion of the game. This needs to be a ArrayList because the maximum length of the output file is unknown, so we need the flexibility an ArrayList provides as opposed to just an Array which requires a fixed length.

For the `run()` method, we have a `while` loop which contains the player's strategy which only exits once the aforementioned `continueGame` flag is set to false. We decided to use a `while` loop this was because the player needs to continuously execute their strategy until they or another player has won the game. However before we enter it, we check if the player already has a winning hand using the `checkWinningHand()` method (which simply traverses the `hand` ArrayList and sets `hasWon` to true if all Card objects of the hand attribute have the same value using their `getValue()` method, otherwise it sets it to false). If the condition is met, we set `continueGame` to be false so that we skip this loop because the player has already won. After the loop, it iterates through each element in `outputFileContents` and writes it to a new file called 'player<player number>.txt' as per the specification, and the thread expires.

In the loop itself, it first calls the `drawCard` method of the `CardDeck` object that is referenced by the `takeDeck` attribute, and stores the returned card object in a local variable called `drawnCard`. Then (if the returned object is not null), it adds it to the `hand` ArrayList. This represents how a player would draw a card in the game.

The specification mentions that after drawing a card, the player would then discard a card to the bottom of the right deck, so after this we call the `discardCard()` of this Player object. This traverses the `hand` ArrayList, looks for a card with a value which does not equal `playerID` and then removes and returns that card – the specification mentions that the card that gets discarded will not match the player's number. Otherwise, the it removes and returns the first card in the player's hand by default (index 0 of `hand`), as the

specification says the player *will* (not *can*) discard a card so this is not an optional action. (The only exception to this is if `hand` is empty, which skips the discard to prevent an error.) The card returned from this is then added to the right deck by passing it via the `addToBottom()` method of `giveDeck`.

Since both gameplay actions are complete, we then check again whether the player has won (and perform the same actions mentioned before if so), and put the thread to sleep for 50 ms to prevent starvation of other threads. This is where we encounter a performance issue, as the sleep wastes processing time with no instructions executed but becomes necessary the more players (and threads) there are. We had to balance the chosen sleep time with these factors in mind.

After the `while` loop (i.e when the game has ended), we then go through `outputFileContents` (which contains strings for each the action taken) and writes each line to our output file, letting the thread end after.

CardDeck Class

We decided that since the gameplay treats decks as a separate entity and has several actions related to it, it would be appropriate to implement it as a class. It's fairly simple, containing two attributes: an ArrayList `cards` to store the contents of the deck and an int `deckID` (plus a getter) to store the deck number. We then have synchronized methods `drawCard()` and `addToBottom()`, which represent the possible actions the player can take on the decks. The former removes and returns the 'top' card (at index 0) from `cards` and the latter takes a Card object as an argument and adds it to the bottom (adds it to the end of `cards`).

CardGame Class

This contains the main method, which starts off by asking for the number of players and the directory of the pack file. We then use a `Scanner` to open the file, and iterate through each line. For each line we create a new Card object using the number stored in that line to construct it and give it its value, and then we add that card to a `cardList` ArrayList. The purpose of this is to keep track of all cards in the game to prepare for distribution to players and decks, and the length of this list is equal to the number of lines in the pack file (which equals the total number of cards). We then shuffle this ArrayList (using Collections) to shuffle the card order for every game.

We then use a `for` loop to create as many `cardDeck` objects as there are players (stored in `noOfPlayers`), using a nested `for` loop to iterate through the first four cards in `cardList` and using them to actually construct our `cardDeck`. Each object is then added to an ArrayList `cardDecks` to keep track of them.

A similar method is used to create our Player objects, but we need to pass on a reference to each deck that each individual object will be interacting with. According to the diagram in the specification, the deck to the left of the player has the number just below the player's, and the deck to the right is just the player's number itself. Therefore, we can use the indexes of `cardDeck` to determine which decks should be passed through. For each player object created, we set its `takeDeck` (via the Player's `.setTakeDeck()` method) to the one stored at the same index as its player number, and the `giveDeck` (via the Player's `.setGiveDeck()` method) to be the one stored at the index equal to its player number + 1. The only exception to this when the player's number is N, as Deck N+1 cannot exist. In this case the `giveDeck` is set to be Deck 0, represented by the first element in `cardDeck`. This sets up the 'round robin' structure of the game:

...[Player N-1] -> (Deck N) -> [Player N] -> (Deck 1) -> [Player 1] -> (Deck 2) -> [Player 2] -> (Deck 3)…

We then iterate through `playerList` 4 times, for each iteration taking a card from `cardList` and giving it to the player, so each Player object ends up with 4 cards.

We are then ready to start the game, so the `run()` method of each Player object is started and they start executing their game strategies. We then enter a while loop, which continuously iterates through the `playerList` checking their `hasWon` attributes (via the `.getWinState()` method). It makes sure to sleep after each loop to prevent starvation of the child Player threads. Although this works, this creates a performance issue with high player counts as the program is constantly polling the threads to check if they have won. Once it has found a thread that has won, it traverses the `playerList` one more time to call their `.declareWinner()` methods, passing through the number of the player who has won. It then exits the `while` loop and performs one more traversal of the `cardDecks` list to call their `.writeOutput()` methods which prompts them to generate their output files.

Test Design Explanation:

[Please note that our tests use the JUnit 4.x framework, not JUnit 5.x]

Card Class

Because the card class is quite simple, only containing one attribute and one method, only one test is required for this. We wrote `testCardCreation()` which creates an instance of the Card object, passing in an integer for construction. We then use `assertEquals(value, card.getValue())` which checks that the value returned is the same one we set at construction. This tests all aspects of the Card Class.

Player Class

Before anything happens to a Player object, it first has to be created. So we started off by designing `testPlayerCreation()` which creates a player object with an integer passed through for construction. It checks that the getter for `playerId` works through `Assert.assertEquals(playerId, player.getPlayerId())`, which ensures that the returned value matches the `playerId` passed on creation. In addition, because we have an additional method for getting a player name (which is a string equal to "player <playerId>"), we also check that the returned string follows that format.

Since the primary action that the player performs within the game is moving cards, we need to check the methods corresponding to these function correctly. We have therefore designed two tests, `testAddCardToPlayerHand()` and `testDiscardCard()` for this purpose.

The test `testAddCardToPlayerHand()` firstly instantiates a Player object and a Card object. It then calls the `addCardToHand()` function of that object passing through that Card object as an argument. It then checks that the value of the of that card matches the one that is now in the hand. It does this by using the `.getHandString()` method of the Player object (which should only return a string of a single character, representing the value of the only card in that hand), and checks that it is equal to the string equivalent of the value used to construct the card in the first place. This ensures that cards are correctly added to the player's hand, and that the getter correctly returns the value to represent the hand's state.

The test `testDiscardCard()` follows a similar design patten, placing a card into the player's hand, but instead calls the `.discardCard()` method. Since the card we added should have been removed from the hand, and that card was the only one in the first place, the `.getHandString()` should then return an empty string. This ensures that cards can be correctly discarded from the player's hand, and the card is removed correctly.

We also have two similar tests, `testDiscardCard_WithHandIncludesNoCards()` and `testAddCardToHand_WithHandIncludesFullCards()`. As the names suggest, these are very similar to the previous methods, but are needed to test that the Player thread can deal with edge cases.

The former checks that when a player tries to discard a card when they have no cards to discard, the `discardCard()` method correctly returns null. It follows the same steps as its parent test, but does not add a Card object to the Player object. It additionally checks that the hand is still empty as expected. This ensures that the method can gracefully handle calls when the hand is empty.

The latter checks that when it is attempted to add more than 4 cards to the hand, it is handled correctly by not adding that extra card. This is because players can only hold a maximum of 4 cards at any time per the specification. It instantiates a new Player object, but calls `.addCardToHand()` 5 times. It then checks that only the first four have been added. This ensures that the method can handle attempts to add extra cards correctly and without crashing.

We also have tests for checking that the method `checkWinningHand()` functions correctly. Since this can only flip the `hasWon` attribute to `True` or `False`, we have two tests which create conditions for each outcome: `testCheckPlayerHasWon()` and `testCheckPlayerHasWon_Negative()`.

The former creates a Player object and puts 4 cards of equal value in its hand. According to the game, this means that player would be the winner, so it then calls the method and then the getter for `hasWon` to verify that it has been set to `True`. This makes sure that the player can correctly check if it has won and declare itself the winner.

The latter does the same as the former, but instead one of the Card objects added has a different value. This checks that the `checkWinningHand()` method returns false correctly, as not all the cards are the same and therefore the win condition is not met.

CardDeck Class

As usual, we have a test to check we can create the object correctly. `TestCardDeckCreation()` creates a an ArrayList of 4 Card objects, as this is the number of cards that are initially distributed to a deck at the start of the game. It then attempts to create a new CardDeck passing this list and an integer as arguments, and uses the getters for the `cards` and `deckID` attributes to check that they were correctly set.

Like the Player class, decks will also have cards added and removed. So we have two test for this:

The test `addCardToDeck()` creates a new CardDeck with 3 cards, and then calls it's `addToBottom()` method with another Card object. It then uses the getter to ensure that the `cards` attribute has successfully updated with this new card at the *end* of the list. This checks that we can add cards correctly to the deck.

The test `testDrawCard()` follows a similar pattern, but instead only creates a deck with the cards of values 1 and 2. It then calls the `.drawCard()` method which should have removed the 'top' card (i.e the card at index 0 of the `hand` attribute – the card with value 1). To check this, we use the getter to check that the string returned is "2", which is the representation of the `hand` attribute that is produced if it only has a single Card object with value of 2 – i.e it only has a 2 card left. This checks that cards are correctly removed from the deck when a card is drawn from it.

The last test we have for this class is `testDrawCard()_WithNoCard`. As the name suggests, the purpose of this test is to check how the deck handles an attempt to draw a card while it is empty. It checks that the returned card is a null object, instead of throwing an exception. This is to ensure that such a case is handled gracefully.

CardPack Class

Finally, we have designed tests for our utility class CardPack, which is designed to assist in organising our cards at the start of the game. Because this class is relatively simple, we only have two tests for this. One to ensure that we can create a CardPack successfully, and another to ensure we can add a Card object to it.

The test `testCardPackCreation()` first creates a CardPack. To ensure that the CardPack is empty (as it should be, CardPacks are not constructed with Card objects) it calls `.getCards()`, which returns an ArrayList of all the cards in the pack, and checks that the size of that list is 0. In addition, it calls the `.remainingCards()` method (which returns an integer equivalent to how many cards are left in the pack) , which should also be 0 as the pack is empty.

We then have `testAddCardToCardPack()`, which again creates an empty CardPack. This time it calls the `.addCard()` method and passes in a Card object with value 1, and uses the `.remainingCards()` method again to check that it correctly states how many cards are left in the pack (which should be 1). This makes sure that both cards can be added successfully, and that the method returns the actual number of cards left.

Development Log:

Date	Time	Role	Signatures (Candidate Numbers)
30/10/24	12:30 – 13:30	Both - Design Discussion	Duru Tiryaki - 032859 Taylor Roel - 023470
6/11/24	12:30 – 13:30	Driver – Duru Tiryaki Passenger – Taylor Roel	Duru Tiryaki - 032859 Taylor Roel - 023470
13/11/24	12:30 – 13:30	Driver – Taylor Roel Passenger – Duru Tiryaki	Duru Tiryaki - 032859 Taylor Roel - 023470
20/11/24	12:30 – 13:30	Driver – Duru Tiryaki Passenger – Taylor Roel	Duru Tiryaki - 032859 Taylor Roel - 023470
27/11/24	12:30 – 13:30	Driver – Taylor Roel Passenger – Duru Tiryaki	Duru Tiryaki - 032859 Taylor Roel - 023470
3/12/24	15:30 – 16:30	Driver – Duru Tiryaki Passenger – Taylor Roel	Duru Tiryaki - 032859 Taylor Roel - 023470
09/12/24	18:00 – 22:00 (Virtual, not in person)	Both - Finalisation and submission	Duru Tiryaki - 032859 Taylor Roel - 023470