ECM2414 REPORT

DESIGN CHOICE FOR THE PRODUCTION CODE:

CLASSES

Card: Represents an individual playing card. The constructor of the Card class accepts a value as its parameter, representing the fixed and immutable value, of the card. This class emphasizes encapsulation and provides the following key methods:

Key Methods:

- getValue(): Retrieves the card's value in int.
- toString(): Converts the card's value to a string.

Deck: Represents a deck of playing cards, uniquely identified by a deck number. The constructor for the Deck class accepts the deck number and a queue of cards as its parameters. This class is designed to allow each player to easily draw a card from the top of the deck and discard a card to the bottom. It emphasizes thread safety for critical operations, ensuring consistency and reliability in a concurrent environment.

Key Methods:

- getCards(): Returns the current queue of cards in the deck.
- addCard(): Adds a card to the bottom of the deck in a thread-safe manner.
- drawCard(): Removes and returns the top card of the deck in a thread-safe manner.
- isEmpty(): Returns if the deck is empty or not.
- writeFinalDeckToFiles(): Writes the contents of the deck to a text file, formatted with the deck's identifier and its current state.

Player: Represents a player in the game, capable of drawing, discarding, and managing cards in their hand whilst participating in a multi-threaded game environment. The player class has a constructor that holds id to uniquely identify a player, preferred denomination to identify which card values a player prefers, draw deck is the deck from which the player draws the card from and discardDeck is the deck to which the player discards the card to. Each player is dynamically assigned a hand so that it doesn't have to be pre assigned. The class implements a runnable interface making it a thread-safe class, allowing each player to operate as an individual thread simultaneously, contributing to a concurrent gameplay. Each players hand and actions are recorded in output files for traceability.

Key Methods:

- getHand(): Retrieves the current hand held by the player.
- addCard(): Adds a card to the beginning of the player's hand in a thread-safe manner.
- hasWinningHand(): Returns if a player has a winning hand, which is a deck with all same card value.
- chooseDiscardCard (): This method iterates through a player's hand and discard a card with has a different value as preferredDenomination in a thread-safe manner.

- run(): This method handles the operations of the players (threads) in the game. It first checks if a player has a winning hand at the start of the game. If a winning hand is found, the winner notifies all other threads in the game, exits, and the game ends. If no winner is found at the beginning, the threads waiting at the startLatch proceed with the game, drawing and discarding cards from and to the deck in a synchronized manner. Once a winning hand is discovered, the winner notifies the other players, exits the game, and the game concludes.

  CardGame: This is the main class which consists of multiple methods that are used to run the game. An ArrayList of players and a queue of decks are initialized using initializePlayers() and intializeDecks() by retrieving the number of players in the game using initPlayerNum(). The decks are then assigned to the players using assignDecks(). The game is then begun by starting a thread for each player using a loop. The loop to join() threads makes sure that each player has completed its execution before moving on to the next one.
  KeyMethods:

- initPlayerNum(): Gets the number of players by a user prompt.
- loadPack(): Gets a location of a valid input pack text file. Then return the values in the pack as a Queue object. It would also check if the pack includes negative integer and matching pack size to the number of players.
- initializePlayers(): Initialize the player objects and load the pack into players' hand.
- initializeDecks(): Intended to initialize decks matching with number of players by loading the pack to deck objects after drawing to players' hand.
- assignDecks(): Assigns drawDeck and discardDeck to each player.

PERFORMANCE ISSUES

- In the CardGame class if an invalid input is either provided to initPlayerNum() or loadPack, the methods will repeatedly request for a new input until a valid input is receive which can cause delays.
- writeFinalDeckToFiles() in Deck and writeToFile() in Player repeatedly write in to the output files which can cause inefficiency when running the game
- Although the use of synchronized methods like addCard(), drawCard() and isEmpty() help prevent race conditions, it introduces contention amongst threads as these methods are called frequently.
- The Creation of ArrayLists and Queues in tight loops may cause a strain in memory for larger players and decks.
- Although the use of Cyclic Barrier ensures that all threads wait at specific points in the game, they can cause delays especially when there are a large number of players.
- The win condition is checked repeatedly within a synchronized block although it causes an increase in contention and delays amongst other player threads it helps make sure that hand is not being modified by any other threads while the win condition is checked.

DESIGN CHOICE FOR THE TESTS:

The test suite for this card game project demonstrates a comprehensive approach to unit testing, focusing on validating the behavior of individual components by using Junit4 framework for the tests. A test class is created for each of the four classes that are involved in the gameplay. The classes are then all wrapped inside a test suite and can be run by the TestRunner class. In the tests, Assert class is used to verify the return value. In tests for Deck and Player, Before tag is used to setup mock objects before running the tests. Mockito framework is used for creating the mock objects.

CardTest:
This test class is minimal but contains comprehensive tests covering core functionality regarding the Card class. It verifies the methods by testing the constructor, value retrieval and the string representation of the card.

- testCreateCard(): Tests for the constructor to verify if the attributes are correctly set by the constructor.
- testGetValue(): Tests for getting the correct card value.
- testToString(): Tests to verify if the card value correctly returns as String type.

DeckTest:
This class tests the functionality of the Deck class by simulating common game operations with mock cards. It ensures the methods work as intended by creating a custom deck through constructors and verifying the core behaviors of the deck. The tests validate that a mock card can be drawn from the top of the deck, reducing its size, and that a new card can be added to the bottom, expanding the deck and placing the card correctly. This class also handles the scenarios where the deck is empty making sure that the original deck class works as intended.

- testDeckConstructor(): Tests for the constructor to verify if the attributes are correctly set by the constructor. Noted that a deck contains empty queue of cards at its creation if we don't input card object as arguments.
- testDeckConstructorWithCards(): Tests for the constructor to verify if the attributes are correctly set by the constructor if there is an input of card object. Size of the queue and the card value is verified.
- testAddCard(): Tests if a card object is correctly added to the queue of cards in the deck.
- testDrawCard(): Given there is a card existed in the deck, testing if the drawing action correctly polls the card and returns a empty hand.
- testIsEmpty(): Tests if a deck is empty.
- testDrawCardFromEmptyDeck(): When using poll() on a empty queue, it should return a null value.

PlayerTest:
This class tests the functionality of the player class by simulating game operations to validate the behavior and interactions of the Player Class in the CardGame. It meticulously tests various aspects of player functionality, including card handling, winning conditions and the multi-threading feature of the game. By using java.reflections to invoke private methods and mock objects to simulate dependencies, the test suite ensure that a player is able to add and discard cards, detect winning hands and interact with decks whilst running as threads. This class tests different scenarios such as creating players with different configurations, managing hand contents and simulating player thread behavior during gameplay. The class also tests players strategy for choosing which card to discard, checking winning hand and managing game states ensuring reliability across various gameplay scenarios.

- testPlayerConstructorWithoutDecks(): Tests if a player's hand is null if we don't pass the deck objects to the contructor.
- testAddCard(): Tests if a card is correctly added to player's hand using the method.
- testSetAndGetWinStatus(): Tests if winStatus value can be correctly set and get.
- testGetId(): Player id should be returned correctly.
- testPreferredDenomination: Player preferredDenomination should be returned correctly.
- testGetWinningPlayerId(): Verify the winning player ID is returned correctly.
- testSetDrawDeck(), testSetDiscardDeck(): They are both similar as testing a deck is correctly set as either discard deck or draw deck.
- testSetWinningPlayerId(): Tests if a winning player id can be setup correctly.
- testHasWinningHand_withWinningHand(): Tests by adding two cards with same value to the player's hand. Then invokes the private method by reflection, to verify if this method return True.
- testHasWinningHand_withNonWinningHand(): Tests by adding two cards with different value to the player's hand. Then invokes the private method by reflection, to verify if this method return False.
- testChooseDiscardCard(): Adds a card to player's hand and invokes the private method by reflection, checks if the card is correctly returned.
- testChooseDiscardCard_nonPreferredCard(): Tests by adding two cards with same value to the player's hand. Then set the preferredDenomination to the value of one of the cards. Then invokes the private method by reflection, to verify if this method return the card different from the player's preferredDenomination.
- testRun_withWinningHand(): Set up the player's hand with a winning hand and simulate the run method logic to test if a player has a winning hand.
- testRun_withoutWinningHand(): Set up the player without a winning hand and simulate the run method logic to test if a player has a winning hand.

CardGameTest:
This class focuses on the testing of the initialization and setup processes of the card game by validating complex game configuration methods. It simulates user interaction and file inputs. The test suite ensures the robustness of the initialization of the game by determining the number of player, loading sample card packs from files and setting up players and decks and then assigning the players with there draw and discard decks. The tests creates a temporary file to check if the game loads the pack and simulates user input streams to test whether a user is able to enter the number of players that will be playing the game. The class also uses reflection to invoke private method form the CardGame class. Key test scenarios include player number input, testing pack loading with both valid and invalid files, initializing players with the correct number of cards in a round robin fashion and then initializing the decks with cards in a round robin fashion and then verifying the correct assignment of draw and discard decks to players. This class ensures the methods in CardGame function as intended and validates successful game initialization.

- TestInitPlayerNum_ValidInput(): Tests to check if the method accepts the valid input entered by the user.
- TestInitPlayerNum_InvalidInput(): Tests to check if the method does not accept the Invalid Input entered by the user.
- TestLoadPack_ValidPack(): Tests whether the loadPack method accepts the temporary valid card pack file.
- testLoadPack_InvalidPackSize(): Tests to check if the loadPack method does not accept the temporary card pack file with an unacceptable number of cards.
- testInitializePlayers(): This method tests the player initialization process by creating players using the constructor, making sure that each player receive four cards a hand and checks the distribution logic each by distributing cards to each player in a round-robin fashion.
- testInitializeDecks(): This method tests the deck initialization process by first initializing players, then it initializes the decks using its constructor and then calling its method. After the cards being distributed the players it, the remaining cards are then distributed into the decks in a round robin fashion ensuring that the distribution logic works as intended.
- testAssignDecks(): This method tests the deck assignment to players. It first initializes the players and the deck. After that it calls the method to assign the decks to players as draw decks and discard decks. The deck assignment verifies the logic if the last player’s drawn and discard decks are assigned correctly whilst also ensuing the assignment of other players’ draw and discard decks.

Development Log

DATE AND DURATION	022072	027277
31-Oct-2024 3 hrs 30 mins	Driver	Spectator
2-Nov-2024 2hr 20 mins	Spectator	Driver
2- Nov- 2024 2 hr	Driver	Spectator
8-Nov-2024 2hr	Spectator	Driver
10-Nov-2024 2hr 30 mins	Driver	Spectator
12-Nov-2024 2hr 30 mina	Spectator	Driver
18-Nov-2024 2hr	Driver	Spectator
30-Nov-2024 2hr	Spectator	Driver
2- Dec -2024 2 hr	Driver	Spectator
4-Dec-2024 2hr 30 mins	Spectator	Driver
9-Dec-2024 2hr 30 mins	Driver	Spectator