Card Game Report

Production Code

Card Class

The Card class represents the playing cards. Each instance of this class holds a Value and an ID. This is what is distributed to players and decks and is played around for a win. Cards is a separate class instead of it being represented by an int that is passed around the player hands. This is a design choice to make the code more readable as seeing Card in the code is more intuitive and straightforward than seeing an ambiguous int that may be represent something different.

Additionally, the encapsulation of the ID within the card instance allows to track it easier. Even if separate cards have the same value, they can be distinguished by their built-in ID instead of an index within an array which always changes. Finally, such a design choice allows for further modifications in the future. For instance, an attribute of suits can be added for more complex gameplay, although the specifications don't require that. This class is thread-safe since the cards are created and distributed before the player threads are started. No method within the class or class instance is accessed concurrently during the actual game. This is reflected by the game since there are always 8n cards where n is the number of players.

Deck Class

The Deck class represents the deck. Each deck contains 4 cards in the beginning. The deck contains a **ConcurrentLinkedQueue**. This is data structure is used because the cards in the queue follow the FIFO. When the player discards a card, the card is added to the end of the queue. The first card in the queue is polled when the player draws a card. This correctly simulates the deck being a stack of cards in real life. Furthermore, **ConcurrentLinkedQueue** ensures that the **add()** and **poll()** methods are already synchronised so that the drawing and discarding cards is always atomic. This ensures that there are no race conditions where the players try to add and poll the same card. This class additionally contains the method **createDeckFile** that creates a .**txt** file, while the method **writeAllCardsToFile** is called at the end of the game by the winning player to write the final contents of the deck into the .**txt** file.

PlayerMoveThread Class

PlayerMoveThread class is the most complex class in the project. It represents the players in the game. The player always holds 4 cards. This is implemented by **doBoth** method which both adds and discards a card with every move. While **addCard** method simply takes the top card from the left deck and adds it to the array, discards class also holds the logic of the player. Each player is assigned a preferred value reflected by the **playerID** (e.g. player1 prefers cards with value 1). With every move, the player discards the first non-preferred value in the queue. It does so until it holds 4 cards with the same value. The player class extends the Thread interface. This makes each instance of the player a separate thread. The **run()** method runs until the player gets a winning hand and with each move it writes its move to its respective **player**.**txt** file. When player thread wins, it sets the volatile attribute **gameOver** to positive, informing all the other players that it won and signalling them to stop running.

The **PlayerMoveThread** also contains a nested class **hand**. This class contains the array of cards as well as the actual logic of the player. Such a decision was made to isolate the card management logic into the hand, while the player class is focused on representing the player in the topology of the game. This resulted in the ease of modifying the hand without affecting the rest of the code. Moreover, the nested class makes the code more intuitive as it reflects the logical grouping of the player attributes allowing for better readability and maintainability.

Notably, the **isWinning** method checks whether all 4 cards hold the same value regardless of the player's preference as a condition for a victory. This ensures that if the player is handed 4 cards with the same non-preferred value in the beginning of the game, the player is still declared the winner.

It is worth mentioning that the initial logic we implemented was completely different. The player would first count the occurrence of each card and keep the ones that occurred the most. We later realised that such a logic would not meet the specification requirements since there would be no preference based on the player's index.

CardImplementor

CardImplementor serves as a central hub for managing the game by bringing together all the other classes. It implements such methods as **loadCardsFromFile**, **distributeToPlayers** and **distributeToDecks**. These methods enable the setup of the game where the players and decks each receive the initial 4 cards in the round-robin fashion where the cards are loaded from a separate **.txt** file. This design choice adheres to single responsibility principle where the classes stick to one job: deck class representing the stack of cards and player class modelling the behaviour of the player. By consolidating game setup logic into the **CardImplementor**, the design remains modular, organized, and easy to maintain.

CardImplementor also implements the **CardInterface**. The inclusion of an interface promotes flexibility as it clearly defines a consistent contract for card-related methods. Any class that implements this interface will adhere to the same standardised behaviour.

CardGame

CardGame class is an executable of the project. It is where the game is set up and executed. It firstly prompts the user to input the number of players. This input must be a positive integer above 2 since all players must be linked to 2 different decks. Otherwise, the executable returns an exception. The executable then prompts the input of the location of the file. The file should contain each card, represented by an integer, in a separate line with the total of 8n cards where n is the number of players. It gives an error message if the path to the file is invalid **(note that the path should be without quotation marks).** This user input validation ensures robustness against invalid files or player counts, while also allowing for smoother user experience.

After all the required information has been input, the executable calls the methods from the implementor to load the cards from the file and distribute them to players and decks. It then starts all the player threads that run until the winner is determined.

However, in case of an invalid pack where there is no winning hand with more than 3 cards of the same value, or the winning value is not preferred by any of the players, the game will loop forever until being manually stopped in the command line. (Ctrl + C)

Test Code

JUnit Framework

jUnit 4.x was chosen as the testing framework for this project. It allows for an environment for different test cases and an ease of their implementation by providing various annotations. **@Test** was used to define tests, while **@Before, @BeforeClass** and **@After, @AfterClass** were used for setting up and later cleaning up the environments. Additionally, **.assert()** methods were employed to verify the outputs straightforwardly. Overall, jUnit framework made the testing of the classes and methods much easier and efficient.

CardTest Class

The test cases of CardTest class are structured around the core behaviours of the Card class, such as its constructor including the retrieval of its value and Id. The test constructor ensures that the card is initialized correctly with provided value and is assigned an ID.

- The **testGetValue** and **testDontGetValue** verifies that a card's value can be correctly retrieved.
- The **testGetId** and **testUniqueIds** ensure that a card returns its appropriate ID and that no IDs are repeated.
- The **testToString** confirms that the string representation of the card matches its expected format.
- Additionally, the **resetIdCounter** method is used to ensure that the ID counter is reset to 0 before each test, which maintains consistency across test runs.

The tests are structured to handle edge cases while adhering to good practices like cleaning up resources and isolating tests to prevent any interference. This approach ensures robustness, accuracy and reliability in every test class's functionality.

DeckTest Class

The test cases of DeckTest class are structured around the core behaviours of the Deck class and ensure that it behaves as expected under various scenarios.

- The **setUp** method employs **@BeforeClass** annotation to initialize one Deck that will be used by all the test cases.
- The **deleteFiles** method employs **@AfterClass** annotation to delete the deck file created after finishing all test cases.
- The **resetDeck** method employs **@Before** annotation to clear all cards in a Deck before each test cases.
- The **testDeckFileCreation** verifies the existence of a deck file when a deck is created.
- The **testAddCardAndSize** adds cards to the deck and checks the consistency of the size.

- The **testDrawCard** ensures that the card drawn from the deck always follows the FIFO convention and not null.
- The **testWriteAllCardsToFile** verifies the correct writing of deck content to files and **testShowCards** validates the printed output of cards, ensuring the correctness of **showCards**.

PlayerMoveThreadTest Class

The test cases of **PlayerMoveThreadTest** class are structured around the **PlayerMoveThread** class emphasizing on modularity, clarity and comprehensive coverage of key functionalities.

- The **setUp** method employs **@BeforeClass** annotation to initialize a player with two decks assigned.
- The **resetPlayerState** method employs **@Before** annotation to clear the decks and hand of the player before every test case.
- The **deleteFiles** method employs **@AfterClass** annotation to delete the deck and player files created after finishing all test cases.
- The **testDoBoth** ensures proper functionality of the drawing and discarding of cards as well as verifying these actions are logged correctly to the output file.
- The **testDrawCard** methods validate that cards are correctly drawn from the left deck and added to player's hand.
- The **testDiscardCard** methods validate that cards are correctly discarded from the hand and added to the right deck.
- The **testAddCardToHand** validates the exact sequence in which cards are added to a player's hand using **assertArrayEquals**.
- The **testWinningHand** and **testNotWinningHand** methods deal with a variety of card combinations, ensuring correctness of the **isWinningHand** method.

The decision to use specific assertion types—such as **assertEquals**, **assertArrayEquals**, **assertTrue**, and **assertNotEquals**—was guided by the need to comprehensively validate individual outcomes while ensuring readability and maintainability.

CardImplTest Class

CardImplTest is a test class used to test the functionality of the CardImplementor class. **@BeforeClass setUp** and **@AfterClass deleteFiles** are methods used for setting up and later cleaning up the environment to avoid unnecessary clatter. For each test, 4 player threads are created since all the methods interact with them.

- **testCreatePlayers** verifies whether the number of players and decks created is n where n is the input given by the supposed user.
- **testShowPlayerDetails** tests whether the topological relationship between players and decks exists and is correct i.e. left and right decks of the players. This is vital since it simulates how decks would be positioned relative to the player in real-life scenario.

- **testLoadCardsFromFile** verifies if the number of cards given in file matches the count of the total number of cards in the game. It must be correct since the number of cards is directly proportional to number of player (8n cards with n players)
- **testShowCardValues** tests whether values match the actual values received from the file.
- **testDistributeToDecks** and **testDistributeToPlayers** check whether the decks and players have 4 cards each after a round-robin distribution as specified in the specifications.
- **testShowCardInDeck** and **testShowCardInHand** verify whether their respective tested methods can handle and throw ID exceptions.

It is important to mention that such methods as **showCardsInHand** and **showCardsInDeck** in the CardImplementor are not directly used for the game loop. They were used to test whether the initial setup of the game is correct. This was verified by inputting specific packs and tracking where each card was located by printing the ID and the card values of each card in the deck and player instances. For instance, if the card with value 6 was in the 3rd line of the pack file (where each line is represented by card ID), it would be tracked down to the third player.

Logging

Date	Time	Duration/hours	730100664	730033161	Signature	Signature
07/11/2024	14:30	0.5	Planner	Planner	730100664	730033161
07/11/2024	17:00	1.5	Planner	Planner	730100664	730033161
07/11/2024	19:30	3	Planner	Planner	730100664	730033161
13/11/2024	17:00	5	Pilot	Co-Pilot	730100664	730033161
14/11/2024	17:00	2.5	Pilot	Co-Pilot	730100664	730033161
17/11/2024	17:00	2	Pilot	Co-Pilot	730100664	730033161
20/11/2024	19:00	1	Co-Pilot	Pilot	730100664	730033161
22/11/2024	11:00	4	Co-Pilot	Pilot	730100664	730033161
22/11/2024	15:30	2	Planner	Planner	730100664	730033161
26/11/2024	17:00	4	Co-Pilot	Pilot	730100664	730033161
28/11/2024	17:00	5	Co-Pilot	Pilot	730100664	730033161
30/11/2024	19:00	1	Co-Pilot	Pilot	730100664	730033161
30/11/2024	23:00	0.5	Pilot	Co-Pilot	730100664	730033161
04/12/2024	13:00	3	Pilot	Co-Pilot	730100664	730033161
06/12/2024	14:30	4	Pilot	Co-Pilot	730100664	730033161
08/12/2024	09:30	4	Co-Pilot	Pilot	730100664	730033161
08/12/2024	15:00	6	Pilot	Co-Pilot	730100664	730033161
09/12/2024	01:00	0.5	Pilot	Co-Pilot	730100664	730033161