Multi-Threaded Card Playing Simulation: Report

Design Choices (Production Code)

Introduction

The project is of an object-oriented structure that is divided into multiple classes including: **Card**, **CardGame**, **CardManager**, **Deck**, **Hand**, **FileCreator** and **Player**. The use of a modular design breaks the code down into multiple problems, making it easier to maintain, extend and debug.

CardGame Class

The **CardGame** class serves as the main class of the game. It handles tasks such as setting up players, decks, and hands from the player input as well as distributing cards.

Key CardGame Class Design Choices

- **Scanner for Input:** The use of the **Scanner** for input allows for easy and flexible user interaction.
- **Round-Robin Distribution** The *assignDecks* method in the **CardGame** class uses a round-robin approach to distribute decks to players. This ensures a fair and even distribution of cards, which is important for the game's fairness.
- **Encapsulation and Data Hiding** Private methods and fields are used to encapsulate the internal workings of the classes. This promotes data hiding and protects the integrity of the objects.

Card Class

The **Card** class represents an individual playing card in the game. Each card is defined by a single *cardValue*. The class provides methods for retrieving the card's value, checking equality with other cards, and verifying if a collection of cards are identical. This class is good for scalability as more complex features could be added onto the class in the future if required.

Key Card Class Design Choices

- **Override toString Method:** The *toString* method is overridden to return the string representation of the card's value. This is useful for debugging and logging, as it provides a human-readable representation of the card.
- **Error Handling:** The *areAllCardsEqual* method checks for an empty list, printing a message if the list is empty. This helps debugging and ensures that the method can handle the edge case.

Player Class

The **Player** class represents an individual player in the card game. It is responsible for executing the gameplay logic via threading. Each player interacts with their leftdeck, rightdeck and hand. The player can perform actions such as drawing, discarding, and checking for victory conditions.

Key Player Class Design Choices

- **AtomicBoolean for Game State:** The *gameOver* field is a static atomic boolean, ensuring thread-safe updates to the game's state. This prevents race conditions when multiple threads check or set the game over status.
- **Thread Control** The *isRunning* field controls the execution of the player's thread. This ensures a clean shutdown. A *Thread.sleep(1)* allows for the scheduler enough time to switch between threads.
- **Synchronization:** The **Player** class uses the *Synchronization* keyword to ensure thread-safe operations during important methods such as *drawAndDiscard* and *checkDiscard*. These operations involve shared resources, such as decks, that are accessed by multiple threads concurrently.

Deck Class

The **Deck** class represents a collection of cards that acts as both a source for drawing and a destination discarding by the player threads.

Key Deck Class Design Choices

Collections are used in this class for several reasons:

- **Thread safety:** collections such as *ArrayDequeue* provide built-in methods that are thread-safe, which simplifies the implementation of synchronized methods for drawing and discarding cards.
- **Atomic Operations:** Collections allow for atomic operations that ensure thread-safe manipulation of the card queue without requiring the use of locking.
- **Readability:** using collections simplifies the management of card list and queue, making the code more readable and maintainable.
- **Queues:** decks function in a first-in-first-out ordering. Because of this, we have decided to implement queues which automatically simplify this process as it is built into their design.

Hand Class

The **Hand** class represents a player's hand during the card game. It also ensures thread safety with synchronized methods, enabling secure access to the cards during gameplay. The class has operations such as adding and removing cards, as well as determining which card to discard based on the player number.

Key Hand Class Design Choices

- **Card Management Methods:** The *addCard* and *discardCard* method provides a controlled way to add and remove card respectively to the hand. This maintains encapsulation and allows for better scalability.
- **Hand is Abstracted:** We could have included this class as an attribute in **Player**, but in order to maintain a clean object-oriented design, we pulled the methods and attributes to its own class. This ensures maintainability and scalability.

Card Manager Class

The **Card Manager** Class is responsible for generating cards from a given set of card numbers.

Key Card Manager Class Design Choices

- **Simplicity and Abstraction:**The class is focused solely on the generation of **Card** objects. This makes the class easier to maintain and test, as it has a clear and singular purpose. The class also abstracts away the details of card generation and is beneficial because it encapsulates the logic for creating **Card** objects, making the code cleaner and more readable.

FileCreator Class

The **FileCreator** class has a sole purpose of creating the output files for the end of the game.

Key FileCreator Class Design Choices

- **Abstraction** Because we have to generate a lot of files (one for each deck and each player), we decided to have this done by a class with a static method. We never create an instantiation of this class. This allows it to easily be reached from any class at any time when we need the output files.

Design Choices (Testing)

Introduction

The tests are divided into multiple classes including: **testCardGame**, **testDeck**, **testCardManager**, **testHand**, **testCard** and **testPlayer**. Breaking down the tests into multiple different classes makes it easier to debug the production code.
The tests for the project are conducted specifically using **JUnit 4**. This is evident from annotations such as *@RunWith(JUnit4.class)* and *@Test*.

testCardGame Class

The **testCardGame** Class is designed to test the functionalities of the **CardGame** class. It uses reflection to access private methods and fields, which allows testing of the non-public class parts in **CardGame**. In addition to this, assertions are used to validate the methods and behaviour of CardGame.

Key testCardGame Design Choices

- **Simulating User Input:** The tests simulate user input by setting the *System.in* stream to a *ByteArrayInputStream*. This allows the tests to provide predefined inputs to methods that read from the console.
- **Setup Method:** The *setUp* method annotated with *@Before* is used to initialize the **CardGame** instance before each test. This ensures that each test starts with a fresh instance of the game.
- **Testing Edge Cases and Invalid Inputs:** The tests cover various edge cases and invalid inputs to ensure the robustness of the **CardGame** class. For example, *testGetPlayerNumberInvalidInput* tests how the method handles non-numeric input, and *testGetPlayerNumberNullInput* tests how it handles empty input.
- **Temporary File Handling** The tests for *getValidPack* include creating temporary files to simulate different scenarios, such as invalid pack data or non-existent files. This approach ensures that tests are isolated and do not depend on external files.

testDeck Class

The **testDeck** class is designed to test the functionalities of the the **Deck** class. Assertions are used to validate the behaviour of methods in the **Deck** class.

Key testDeck Class Design Choices

- **File Verification:** The *testEndSequence* test verifies the creation of the output file at the end of the game when *endSequence* is called. This ensures that deck correctly handles file output, which is an important part of the card Game.
- **Use of Queue<Card> for Deck Representation:** Deck uses *Queue<Card>* to represent the cards in the deck and it is important to test this as a queue supports FIFO (First In Last Out). This aligns with how a deck is used in real life (Cards are drawn from the top of the deck).

testCardManager Class

The **testCardManager** Class is designed to test the functionalities of the **CardManager** class. Assertions are used to validate the behaviour of methods in that class. The tests cover various scenarios such as generating cards from valid arrays, empty arrays, arrays with duplicates, arrays with negative values, and boundary values.

Key testCardManager Class Design Choices

- **Test Data:** The test methods use different arrays of integers as inputs to the *generateCards* method. The justification for this, help verify that the *generateCards* method handles various inputs correctly, including valid values, empty arrays, duplicate values, negative values, and boundary values.
- **Loop for Assertions:** Some of the tests methods use a loop to check each card in the generated list, this is important to ensure that each card in the list has the expected value. It helps to verify that the *generateCards* method correctly converts the input array into a list of **Card** objects.

testHand Class

The **testHand** Class is designed to test the functionalities of the **Hand** class. Assertions are used to validate the behaviour of the methods in that class. The tests check several methods such as initialization, retrieving the hand, adding or discarding cards and handling edge cases where the payer's hand may not contain the specified card to discard.

Key testHand Class Design Choices

- **Test depth:** Each method of the **Hand** class has a corresponding test case focusing solely on that method's functionality, this ensures precise identification of issues by testing components in isolation, making debugging more efficient.
- **Test Coverage:** All key operations are being tested to validate all the critical functionalities of the **Hand** class. This ensures confidence in the correctness and reliability of the **Hand** class.

testCard Class

The **testCard** Class is designed to test thefunctionalities of the **Card** class. It checks various methods such as retrieving the card's value, comparing cards, and verifying the string representation of a card.

Key testCard Class Design Choices

- **Static Method testing:** The static method *areAllCardsEqual* is tested with different scenarios.
  - **Valid:** All cards have the same value.
  - **Invalid:** Cards have different values.

  - **Empty:** No cards exist in the ArrayList. It is important to note that this test case does not pass as the code is unable to handle when no cards are in the ArrayList. This is not an important issue as there will never be a case where no cards are present in a pack. However, this may be good to correct in the future.
  - **Other Special Cases:** Empty list of cards, Single card in list, Boundary values for card values.

- These tests exist to cover a variety of cases to ensure that the method works reliably for all potential inputs and edge cases.

- **Method Testing:** Each method of the **Card** class is tested individually to ensure focused and precise validation.

  - *testGetValue:* Creates a card with a specific value, retrieve the value using *getValue* and enures that the **Card** class properly stores and returns its value, which is very important for it's functionality
  - *testEqual:* Compares two cards with the same value, two cards with different values, and a card with null.

- **Boundary Tests:** Boundary tests were conducted for the above methods these include:

  - *testNegativeIsEqual:* This assigns negative values to the cards, we took an assumption that negative integers are valid and therefore this test passes.
  - *testMaxBoundaryIsEqual* and *testMinBoundaryIsEqual:* Tests extreme values (Integer.MAX_VALUE, and Integer.MIN_VALUE). This verifies that the **Card** class behaves correctly under edge-case scenarios, ensuring robustness.

testPlayer Class

The **testPlayer** class is responsible for testing the methods in the **Player** class. The tests cover multiple different methods within that class. The design also ensures proper handling of edge cases. Reflection is also used to access some of the private methods within the class.

Key testPlayer Class Design Choices

- **Thread Management tests:** *testRunMethod()* and *testStopPlayerThread* test the **Player** class's threading behaviour, ensuring player thread can start and stop correctly.

- **Victory Condition Tests:** Several tests check the *checkVictory* method in the **Player** class. These include:

  - **Valid:** The *checkVictory* method works with the expected cards.
  - **Invalid:** The *checkVictory* method correctly identifies an invalid card match.
  - **Special Cases:** *testNonPlayerNumberCheckVictory* refers to when there is a matching pair that is not equal to the Player's number. This is to check that if, at the start of the game, the player wins with cards not equal to the player's number if they receive cards of the same value. Boundary test cases are also in place to check that *checkVictory* works with the maximum and minimum integer limit.

Improvements

- Null Cards - We know that we do not handle null cards, which can be a problem and causes one test case to fail (in testCards). We did not handle null cards due to time restrictions and focusing on enabling the program to function correctly. This should not pose a problem in running the program, but there may be untested cases where it causes a crash.

- **CardGame** Abstractions - Currently, **CardGame** is around 200 lines of code. We believe that it would be a more elegant and maintainable project if we broke the functions of **CardGame** into one or two more classes

Development Log					
Date of Activity	**Start Time**	**End TIme**	**Duration (h)**	**730049513 - Role**	**730040399 - Role**
23/10/24	16:30	17:45	1.25	Planning	Planning
24/10/24	14:20	15:20	2	Design Choices	Design Choices
27/10/24	16:30	18:30	4	UML Diagram	UML Diagram
28/10/24	16:30	18:30	1	Planned Testing	Planned Testing
29/10/24	16:00	18:30	2.5	Planned Testing	Planned Testing
30/10/24	16:00	19:00	3	Navigator	Driver
1/11/24	16:30	18:00	1.5	Driver	Navigator
04/11/24	13:00	14:00	1.5	Navigator	Driver
06/11/24	16:30	18:30	2	Driver	Navigator
08/11/24	16:30	18:30	4	Navigator	Driver
11/11/24	13:00	15:30	2.5	Driver	Navigator
12/11/24	14:30	18:30	2	Driver	Navigator
13/11/24	16:30	18:30	4	Navigator	Driver
20/11/24	18:00	20:00	4	Driver	Navigator
23/11/24	14:00	17:00	3	Navigator	Driver
27/11/24	16:00	18:00	4	Report writing	Report writing
08/12/24	11:00	16:00	5	Report writing and Submission	Report writing and Submission

Developer Log