Software Development Continuous Assessment Report

1. Requirements Specification

1.1 User Stories

The project is dedicated to generating a multi-threaded application that shall be a multi-threaded card game simulation system that is scalable, thread-safe and supports concurrent player operations, and shall also be able to generate log files to validate game results. The game shall enable players to draw, discard, sort their hand and declare victory when they have four cards of the same number.

As a player, I would like a fair distribution of initial hands.

- **Requirement:** The system should distribute 4 cards from the pack to each player in a round-robin fashion to ensure all players have the same starting conditions.

- **Verification:** Each player's initial hand must consist of 4 cards, distributed in a round-robin fashion from the pack.

As a player, I want to be able to draw and discard cards and win the game through a certain strategy.

- **Requirement:** Each player should be able to draw a card from the left-side deck and discard a card into the right-side deck; the discarded card should be a card with a non-preferred value.

- **Verification:** After each draw-discard operation, the player's hand remains at 4 cards, and the discarded card is verified to be a non-preferred card according to the strategy.

As a player, I want to win the game when the victory conditions are met.

- **Requirement:** If a player has four cards of the same value in their hand, the system should allow that player to immediately declare victory and end the game.

- **Verification:** Each player should have a log file that holds the contents of their hand at the end of the game, and the winning player's log should contain the text of that player's victory.

As a developer, I want to ensure thread-safety for multi-threaded operations.

- **Requirement:** During card drawing and discarding operations, appropriate locking mechanisms should be used to prevent multiple threads from interfering and to ensure data consistency.

- **Verification:** The number of cards in the deck always changes as expected under concurrent multi-threaded operations.

As a developer, I should be able to verify the legitimacy of inputs and files.

- **Requirement:** The player format should be a positive integer, and the card pack file must contain 8n non-negative integers (n is the number of players).

- **Validation:** The system should be able to reject invalid files (e.g., containing non-numbers, negative numbers, or insufficient number of cards) with clear error messages.

As a user, I want to start the game with simple commands.

- **Requirement:** The program should be able to receive the number of players and the path to the card pack file as command line inputs.

- **Validation:** The user should be able to enter the number of players (n) and the file path in sequence after starting the program.

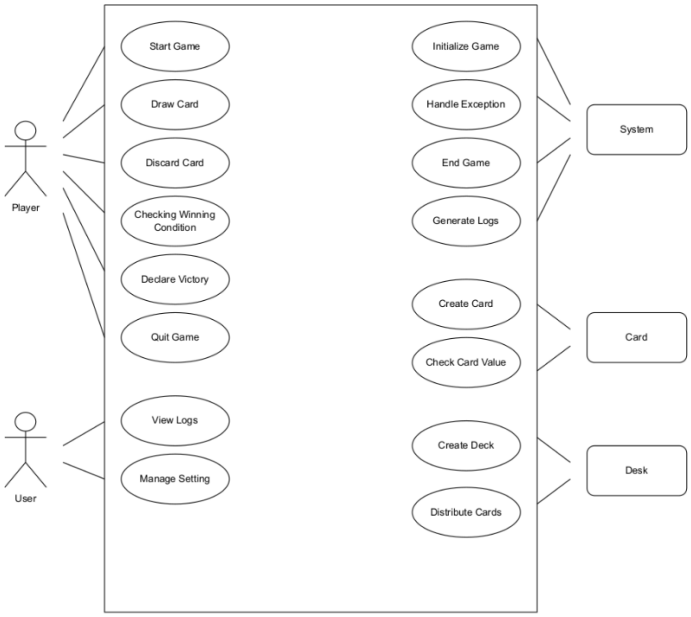
As a user, I would like to generate comprehensive log files to review the gameplay.

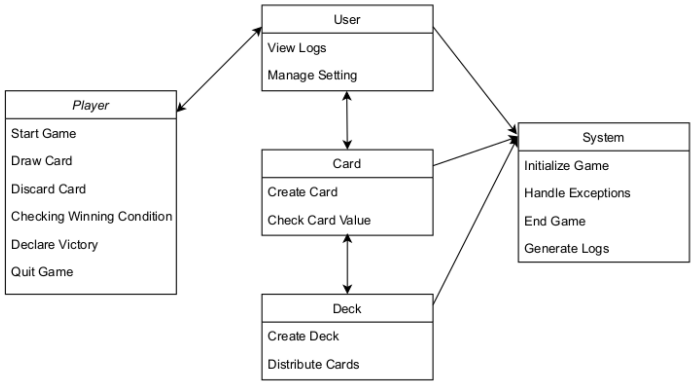
- **Requirement:** A separate log file should be generated for each player and each deck, recording the initial hand, the action history, and the final state.

- **Verification:** The log file should be generated at the end of the game, and its content should contain a chronological record of operations.

1.2 Use Case Model

The use case diagram illustrates the interactions between actors and the system:





1.2.1 Player Related Use Cases

Start Game: The system generates the initial deck and cards, distributes the cards to the players and the decks, and the game enters the running state.

Draw Card: The player draws a card from the top of the left-side deck, the card is added to the player's hand, and the system updates the status of the deck.

Discard Card: The system removes the selected (non-preferred) card from the player's hand, adds the card to the bottom of the right-side deck, and updates the status of the player's hand and the deck.

Check Winning Condition: The player verifies if there are 4 cards of the same value in their hand and triggers the Declare Victory case if the condition is met.

Declare Victory: The system prints the winning player's information on the terminal, updates the game status to finished, and notifies other players.

Quit Game: The system closes the game window and saves the current game state (e.g., log file).

1.2.2 User Related Use Cases

View Logs: The user reviews the game logs to understand the game process and results.

Manage Settings: The user inputs the number of players and the card pack file path, then the system verifies the legitimacy of the number of players and the card pack file.

1.2.3 Card Related Use Cases

Create Card: The system generates a new card object, assigns a value to it from the pack file, and adds the card to the specified deck.

Check Card Value: The system checks if there are 4 cards of the same value in the player's hand and triggers the Declare Victory case if the condition is met.

1.2.4 Deck Related Use Cases

Create Deck: The system creates an empty deck and adds the generated cards to the deck.

Distribute Cards: Cards are distributed to players in a round-robin manner until each player receives 4 cards. The remaining cards are similarly distributed to the decks.

1.2.5 System Related Use Cases

Initialize Game: The system generates the initial card pack, distributes the cards to players and decks, and starts the player threads.

Handle Exceptions: The system checks for inconsistencies (such as an empty deck) and triggers appropriate actions (such as ending the game) if necessary.

End Game: The system stops all player threads, generates the final log file, and closes the game window.

Generate Logs: The system records player actions (e.g., card draw, card discard) and writes the logs to player(n)_output.txt and deck(n)_output.txt.

2. System Design Specification

2.1 System Development Section

In this section, we did not use any framework. In the design, we created 5 core classes named **Card**, **CardGame** (which includes main method), **Deck**, **GameStatus**, and **Player**. Each part works together to create a multi-threaded simulation where players interact concurrently through decks while ensuring that thread safety and game flow rules are maintained.

2.1.1 Card Class

Purpose: Represents a single playing card with a non-negative integer face value.

Key Points: The card's value is set when the card is created and cannot change. This makes the class inherently thread-safe.

Explanation: When a Card is created, it holds the value for its lifetime. No other thread can change this value, making it safe to share between threads.

2.1.2 Deck Class

Purpose: Models a deck of cards used in the game. There is one deck per player, and they are arranged in a ring topology. Each deck is accessed concurrently by players, so thread safety is crucial.

Key Points: Each deck has an ID (for example, deck 1, deck 2, etc.). Uses a FIFO queue (implemented with a LinkedList) to hold cards. A ReentrantLock is used to protect operations (drawing from or adding to the deck) from concurrent access.

Explanation: The lock in the deck ensures that when a player draws or adds a card, no other thread interferes with the deck's state. Additionally, when two decks need to be locked simultaneously (for the atomic draw-discard operation), the code always locks them in a consistent order (based on deck ID) to avoid deadlocks.

2.1.3 GameStatus Class

Purpose: Serves as a shared communication mechanism to indicate when the game is over and which player (if any) won.

Key Points: The use of `volatile` ensures that changes to `gameWon` and `winner` are visible to all threads immediately. The method `declareWinner` is synchronized to ensure that only one player can set the win condition

Explanation: When a player detects that their hand meets the winning condition, they call `declareWinner`. This method sets `gameWon` to true and records the winning player's ID, ensuring that no other thread can override this declaration.

2.1.4 Player Class

Purpose: Represents a player in the game. Each player runs on its own thread and is responsible for managing its hand of cards, interacting with two decks (for drawing and discarding), and logging its actions.

Attributes:

- **`id`:** The player's unique ID
- **`preferredValue`:** The card value that the player prefers (equal to the player's ID)
- **`hand`:** A list that stores the four cards in the player's hand
- **`leftDeck` & `rightDeck`:** References to the decks from which the player draws and to which the player discards
- **`gameStatus`**: A shared reference to the GameStatus object for checking if the game is over

Game Flow:

1. **Initial Check:** Logs the initial hand and checks if it is already a winning hand
2. **Game Loop:**
    - **Draw a Card:** Locks both the left and right decks (in a fixed order to avoid deadlock) and draws a card from the left deck.
    - **Discard a Card:** Randomly selects a non-preferred card and discards it to the right deck.
    - **Log Actions:** Records the draw, discard, and updated hand in the player's output file.
    - **Win Check:** Checks if the current hand is a winning hand and, if so, declares a win.
    - **Exit:** If the game has been won by another player, logs that it was informed and exits.

2.1.5 CardGame Class

Purpose: Acts as the entry point of the application. It reads user input, sets up the game (distributes cards to players and decks), starts player threads, and finally writes out the decks' contents once the game concludes.

Attributes:

- **User Input:** Prompts the user for the number of players and the location of the pack file containing exactly 8 × (number of players) cards.
- **Input Validation:** Reads the pack file and checks that it contains the correct number of cards.
- **Distribution of Cards:** The first 4n cards are distributed round-robin to the players' hands. The remaining 4n cards are distributed round-robin to the decks.
- **Topology Setup:** Arranges decks and players in a ring. For player *i*, the left deck is deck *i*. The right deck is the next deck in the ring (wrapping around for the last player).
- **Thread Management:** Creates a Thread for each player and starts them concurrently.
- **Final Output:** After player threads finish, writes each deck's final card sequence to an output file.

2.2 System Test Section

In this section we use the **JUnit 4** framework, which is a widely used Java unit testing framework for writing and running unit tests in Java applications.

We have used Java annotations in our test code instead of the traditional test method naming conventions and other configurations. We also use JUnit 4 assertions to check the consistency between expected and actual results, including `assertEquals`, `assertTrue(condition)`, `assertFalse(condition)` and `assertNull(object)`.

Finally, we use Test Suites, which combine multiple test classes and run them as a whole using the `@RunWith` and `@SuiteClasses` annotations.

2.2.1 Card Test

- Test that the constructor and `getValue()` method of the Card class work properly and ensure that the returned card value is what expected
- Test that the `toString()` method returns a string in the correct format based on the card's properties to ensure that the output meets the specification

2.2.2 Deck Test

- Test that a newly created deck is empty and ensure that calling `drawCard()` on an empty deck returns null
- Test adding and drawing cards to ensure that the `addCard()` and `drawCard()` methods work properly
- Test that the `getContents()` method can correctly list all the cards in the deck to ensure that the method can reflect the current state of the deck
- Test the order of multiple cards to ensure that `drawCard()` draws cards in first-in, first-out order

2.2.3 CardGame Test

- Test that the `readPackFile()` method in the CardGame class can correctly parse the file to ensure that the correct card list is generated based on the number of players and deck size
- Simulate the actual execution of the game to ensure that output files (such as player operation records) are generated correctly, and verify the correctness of the game flow through GameFlowValidatorSorted

2.2.4 GameStatus Test

- Test the default state of the GameStatus class at the beginning of the game to ensure that the initial state is as expected
- Test whether the `declareWinner()` method can correctly update the game state and ensure that only the first winner is retained when multiple players declare victory

2.2.5 Player Test

- Test by `testInitialHand()` method to verify that the player can correctly end the game and be judged as the winner when they have a hand that meets the winning conditions
- Test by `testCardSelection()` method to verify how the player decides to discard cards based on their preferred card selection strategy and ensures that the discarded card is correctly moved to the right deck

3. Development Log

Date	Time	Duration	Roles	Signature
2025/3/1	13:00-17:00	4 hours	Driver: Mert Lale, Navigator: Zhenwei Song, Ruizhe Liu	740087634, 740091412, 740075877
2025/3/2	14:00-18:00	6 hours	Driver: Mert Lale, Navigator: Zhenwei Song, Ruizhe Liu	740087634, 740091412, 740075877
2025/3/3	20:00-22:00	2 hours	Driver: Mert Lale, Zhenwei Song, Navigator: Ruizhe Liu	740087634, 740091412, 740075877
2025/3/4	19:30-23:00	3.5 hours	Driver: Mert Lale, Navigator: Zhenwei Song, Ruizhe Liu	740087634, 740091412, 740075877
2025/3/5	14:00-17:00	3 hours	Driver: Mert Lale, Zhenwei Song, Navigator: Ruizhe Liu	740087634, 740091412, 740075877
2025/3/6	15:00-19:00	4 hours	Driver: Mert Lale, Navigator: Zhenwei Song, Ruizhe Liu	740087634, 740091412, 740075877
2025/3/7	14:00-18:00	4 hours	Driver: Zhenwei Song, Navigator: Mert Lale, Ruizhe Liu	740087634, 740091412, 740075877
2025/3/8	13:30-17:00	3.5 hours	Driver: Zhenwei Song, Navigator: Mert Lale, Ruizhe Liu	740087634, 740091412, 740075877
2025/3/9	15:00-18:30	3.5 hours	Driver: Zhenwei Song, Ruizhe Liu, Navigator: Mert Lale	740087634, 740091412, 740075877
2025/3/11	18:00-20:00	2 hours	Driver: Zhenwei Song, Mert Lale, Navigator: Ruizhe Liu	740087634, 740091412, 740075877
2025/3/12	17:00-22:00	6 hours	Driver: Ruizhe Liu, Navigator: Mert Lale, Zhenwei Song	740087634, 740091412, 740075877
2025/3/14	15:00-19:00	4 hours	Driver: Ruizhe Liu, Navigator: Mert Lale, Zhenwei Song	740087634, 740091412, 740075877
2025/3/15	14:00-17:00	5 hours	Driver: Ruizhe Liu, Zhenwei Song, Navigator: Mert Lale	740087634, 740091412, 740075877