**SOFTWARE DEVELOPMENT REPORT**

**1. INTRODUCTION**

**1.1 Purpose**

This document outlines the requirements for the card game implementation, including user stories, use case models, and the rationale behind these choices. The objective is to develop a fully functional multiplayer card game where players can draw, discard, and win based on specific game rules.

**1.2 Scope**

The card game consists of multiple players interacting with individual decks, drawing and discarding cards in a turn-based system. The game should be implemented with proper object-oriented design and should include multi-threading for concurrent player actions. The system must provide clear game logs and ensure smooth gameplay until a winner is declared.

**2. REQUIREMENTS AND USE CASE MODEL**

**2.1 User Stories – assuming the game may add an interactive human player in the future.**

* As a player, I want to draw a card from the deck so that I can build my hand.
* As a player, I want to discard a card to the right deck so that I can manage my hand effectively.
* As a player, I want to win the game if I collect four identical cards so that I can achieve victory.
* As a game administrator, I want to initialize players and decks correctly so that the game starts with valid configurations.
* As a player, I want to receive notifications when another player wins so that I can exit the game appropriately.

These user stories ensure a structured and fair gameplay experience while covering essential game mechanics.

**3. USE CASE MODEL**

A diagram of a card game system

AI-generated content may be incorrect.**3.1 Actors**

* **Player:** A thread representing an individual participant in the game. Responsible for performing draw and discard actions, checking for a win condition, and responding to game-end notifications.
* **Game Administrator:** The human agent who begins the game, provides the pack file and sets the number of players at launch.
* **System**: The internal game logic managed primarily by the CardGame, Turn, LogWriter, and Messager classes. Handles game setup, deck and player coordination, validation, logging, and winner notification

**3.2 Use Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Use Case ID** | **Description** | **Actors** | **Preconditions** | **Postconditions** |
| UC1 | Draw a card from the deck | Player | The deck has at least one card | The player’s hand increases by one card |
| UC2 | Discard a card to the right deck | Player | The player has at least one card | The player's hand decreases by one card |
| UC3 | Determine the winner | Player, System | A player collects four identical cards | The game ends, and all players are notified, |
| UC4 | Initialize the game | Game Administrator, System | A valid card pack is provided | Players and decks are created and assigned |
| UC5 | Notify players of the winner | System | A player wins the game | All other players are informed and exit |
| UC6 | Log player and deck actions | System | An action (draw, discard, win, exit) occurs | Action is written to correct output file |
| UC7 | Provide pack file and initialize game | Game Administrator | Pack file exists, enters a number of players | Game starts only if the pack is valid, and admin initializes game |

The following sequence diagrams illustrate the dynamic interaction between objects during key use cases of the game: taking a turn and notifying players upon a win:

.A diagram of a card

AI-generated content may be incorrect. A diagram of a computer program

AI-generated content may be incorrect.

**4. RATIONALE FOR REQUIREMENTS**

**4.1 Justification for User Stories**

* **User engagement**: Ensuring players can draw, discard, and win creates an engaging experience.
* **Game fairness**: Structuring the game mechanics to prevent errors ensures fairness.
* **Automation and notification**: The observer system maintains smooth gameplay without manual intervention.
* **Concurrency support**: multi-threading ensures efficient execution in multiplayer settings.

**4.2 Justification for Use Case Model**

* The use case model clearly defines system boundaries and interactions.
* The distinction between different actors helps in modular implementation.
* Precondition and postcondition constraints ensure robustness in execution.
* The design choices align with best practices in object-oriented programming.

**5. DESIGN CHOICES AND TESTING STRATEGY**

**5.1 Production Code Design**

The program follows an object-oriented design with modular classes:

* **Card:** Represents a playing card with a value.
* **Deck:** Manages a queue of cards, providing methods to draw and discard.
* **Player:** Implements game logic, including drawing, discarding, and checking for a win.
* **GameDriver:** Singleton controller that handles game initialization, player creation, and execution flow.
* **LogWriterInterface:** Interface containing the abstract method writeLog(String stage, int player)
* **LogWriterFactory:** Contains methods to create either a PlayerLogWriter or a DeckLogWriter object
* **PlayerLogWriter:** Handles file output for a player.
* **DeckLogWriter:** Handles file output for a deck
* **Messager:** Notifies players when a winner is declared.

A screenshot of a computer

AI-generated content may be incorrect.

**Key Design Choices:**

* **Singleton pattern:** A single instance of the GameDriver class runs the game
* **Simple Factory pattern:** A very simple version of the factory pattern creates different Loggers for the deck or players – as, while the same in principle, the writeLog method logs in different ways.
* **Synchronization:** CardDeck methods are synchronized to ensure thread safety.
* **Notification Logic**: Players are manually notified using a central method that checks all running threads and calls their onGameEnd() method. While not a formal Observer pattern, this serves a similar purpose of broadcasting game-end events.
* **Threading:** Each player runs as a separate thread for concurrency.
* **File Logging:** Logs are written to track player actions and final results.

**6. DESIGN CHOICES AND JUSTIFICATIONS**

**6.1 Object-Oriented Design principles and patterns**

The system is implemented using an object-oriented programming (OOP) approach to ensure modularity, reusability, and scalability. The key design principles applied include:

* **Encapsulation**: Each player and deck are encapsulated within separate classes to manage their respective states and behaviors. The design uses composition over inheritance. Player and Deck are modular and encapsulate their own logic without relying on a superclass. While a “GameEntity” superclass was trialed initially, and later a “Loggable” interface implemented by CardDeck and Player, the different ways of logging a players behaviour were too unique and it was found to overcomplicate the design while adding nothing.

**6.2 Multi-Threading and Concurrency**

**Balancing turn atomicity and concurrency:**

* Each player is run on a separate thread using the Player class, which extends Thread.
* The core gameplay occurs inside each player's run() method, which repeatedly calls a static method Turn.takeTurn(...). This method handles both the draw and discard phases of a player's turn, keeping them atomic.
* To ensure atomicity, the Turn class locks both the left and right decks using synchronized blocks. The two decks are locked in a consistent order (by deck ID) to prevent deadlocks.
* Multiple players can still take turns concurrently if they are using different, non-overlapping decks — satisfying the concurrency requirements as well as the atomicity.

**Player behaviour once a Win is processed:**

* Inside takeTurn(...), the game checks a shared gameOver flag both before and after drawing a card. This prevents players from starting a turn if the game has already ended.
* However, if a player was already mid-turn when another player won, they are allowed to finish their discard, as in a real game. This ensures every player ends the game with exactly four cards, preserving the atomic draw-discard rule required by the specification.
* A win is only declared after a full turn is completed, specifically after a player has discarded and their hand consists of four matching cards.
* The game does not allow a player to "win early" with five cards in hand, ensuring alignment with standard card game rules – you can imagine in a real game, a player would have to discard their previous card before they can shout that they’ve won!
* A delay (Thread.sleep(100) is used at the end of Run() in the Player class to ensure all players have time to log their outcomes before exiting.

**6.3 Choice of Test Framework: JUnit 4**

* JUnit 4 was selected as the testing framework due to its simplicity, reliability, and wide adoption across the Java ecosystem.
* It provides all the core features required for this project, including @Test annotations, assertions, and exception testing, with minimal setup and overhead.
* Although JUnit 5 offers additional features like parameterized tests and a modular test engine, these were not essential for the needs of this coursework.
* For a small-to-medium scale project like this, JUnit 4 is not only sufficient but also more universally compatible — many development environments and build tools (e.g., Eclipse, IntelliJ, Maven) still default to supporting JUnit 4 out of the box.

**6.4 Test Cases and Coverage**

The testing strategy covers unit tests, integration tests, and edge case handling. Example test cases include:

**Unit Tests**

* **Test drawing a card**: Verify that drawing increases the player's hand size.
* **Test discarding a card**: Ensure that discarding reduces the player's hand size and places the card in the correct deck.
* **Test winning condition**: Check if a player with four identical cards is correctly declared as the winner.

**Integration Tests**

* **Simulated gameplay**: Run a full game cycle with multiple players to ensure correct behavior.
* **Concurrency handling**: Test scenarios where players draw and discard simultaneously to validate synchronization.

**6.5 Code Maintainability and Scalability**

* **Modular structure**: Each component (Player, Deck, Game) is independently managed to facilitate modifications.
* **Dependency injection**: Ensures flexible testing by decoupling object creation.
* **Logging mechanism**: Provides game status updates for debugging and user feedback.

**6. DEVELOPMENT LOG**

| **Date** | **Time** | **Duration** | **Developer Role(s)** | **Summary of Work Completed** | **Candidate Numbers** |
| --- | --- | --- | --- | --- | --- |
| 01/03/2025 | 14:00 – 17:00 | 3 hours | Analyst / Designer | Planned the overall game flow and structure. Wrote user stories and use case models. Identified key classes (Player, CardDeck, CardGame, superclass GameEntity). | 680049063, 740074572 |
| 04/03/2025 | 12:00 – 14:00 | 2 hours | Navigator/  Driver | Implemented Card and CardDeck classes. Set up synchronized drawCard and discardCard methods. Wrote initial unit tests. | 680049063, 740074572 |
| 07/03/2025 | 15:00 – 17:00 | 2 hours | Both Drivers | Implemented Player class with basic gameplay loop. Introduced multi-threading using Thread and Runnable. | 680049063, 740074572 |
| 10/03/2025 | 13:00 – 15:00 | 2 hours | Navigator/  Driver | Added CardGame class with pack loading and player initialization. Integrated thread-safe round-robin hand dealing. Removed GameEntity superclass. | 680049063, 740074572 |
| 12/03/2025 | 14:00 – 17:00 | 3 hours | Navigator/  Driver | Designed and implemented the Turn class to enforce atomic draw-discard operations with deck locking. | 680049063, 740074572 |
| 15/03/2025 | 11:00 – 13:00 | 2 hours | Tester / Debugger | Built unit tests for CardDeck behavior (FIFO draw/discard). Verified player hand size after turns. | 680049063, 740074572 |
| 18/03/2025 | 13:00 – 15:00 | 2 hours | Navigator/  Driver | Enforced win condition checking and gameOver logic. Used synchronized block to ensure only one winner. | 680049063, 740074572 |
| 21/03/2025 | 10:00 – 2:00 | 4 hours | Debugger/  Driver | Experimented with a Loggable interface to unify player and deck logging. Reverted to simple, manual logging methods after evaluating complexity. | 680049063, 740074572 |
| 22/03/2025 | 14:00 – 16:00 | 2 hours | Tester / Debugger | Developed test cases for atomic turn behavior and concurrency. Wrote integration tests to validate draw/discard and logging order. | 680049063, 740074572 |
| 23/03/2025 | 12:00 – 17:00 | 5 hours | Debugger/  Driver | Implemented Simple Factory Pattern to decouple logging logic from the core game logic, splitting Player and Deck logging into separate classes. Cleaned up comments. | 680049063, 740074572 |

**7. ERROR HANDLING AND EDGE CASES**

**7.1 Common Error Scenarios**

To ensure robustness, the system must handle various error scenarios gracefully:

1. **Empty Deck Handling**
   * **Scenario**: A player attempts to draw a card from an empty deck.
   * **Expected Behavior**: Notify the player and prevent the action until new cards are available.
2. **Invalid Discard**
   * **Scenario**: A player tries to discard a card that does not belong to their hand.
   * **Expected Behavior**: + Expected Behavior: This scenario is prevented by design — players can only discard from their own hand.
3. **Simultaneous Moves Conflict**
   * **Scenario**: Multiple players try to draw from the same deck simultaneously.
   * **Expected Behavior**: Use synchronization mechanisms (deck locking) to process actions atomically without overlap, while still running unaffected players simultaneously
4. **Game Initialization Errors**
   * **Scenario**: The administrator provides an incorrect number of decks or an invalid card set.
   * **Expected Behavior**: Halt the game setup and prompt the administrator to correct the input.
5. **Multiple Winners Conflict**
   * **Scenario**: Two players achieve the winning condition simultaneously.
   * **Expected Behavior**: The system should have predefined tie-breaker rules or declare a shared victory.