ECM2414 Card Game – Report

- Candidate ID: 730061231
- Partner Candidate ID: 750082802

Production Code Design

Overview

The main goal of this assessment was to do a reliable multi-threaded card game simulation that follows some specifications like respecting the topology, the rules and the file-input/output requirements specified in the task.

Package structure

The game is hosted in the “cardgame” package, which contains all the classes the make the game work. Below a list will show the classes and their functions.

- Card: Defines a simple object that encapsulates the value of a card and guarantees its validity (throwing exceptions like “Card value must be non-negative”) at the time of creation. Once instantiated, the card cannot be modified, making it safe for concurrent use in the multithreaded card-game.
- CardDeck: Represents a thread-safe card deck implemented as a FIFO (First in, first out) queue. It also maintains an ordered collection of cards (Card) associated with an identifier (deckId). The deck allows cards to be drawn and discarded synchronously, ensuring that only one thread accesses the structure at a time.
- Player: This class represents a player in the card game and acts as a concurrent unit within the system. Each player tuns in their own thread and follows the rules established by the simulation. On each turn, the player draws a card from the left deck, discards an unpreferred card to the right deck, and always keeps four cards in their “hand”. Their preferred value corresponds to their identification number (id). If at any point, all four cards in their “hand” are if the same rank, that player declare a victory and inform the other players. The class design ensures safe multithreaded execution through synchronized blocks that prevent deck access conflicts and guarantees deterministic behavior and orderly file writing.
- CardGame: This is the core of the program and is responsible for coordinating entire game flow. Its function is to start the game, prompting the user for the number of players and the card file, validate the data, deal the cards among players and decks, and then create and execute each player’s thread. During execution, CardGame monitors the overall game state using an atomic flag that indicates when there is a winner and who it is. Once all players have finished, the program generates the output files for each deck with its final contents.

Concurrency

Concurrency control was designed to be safe and simple, prioritizing clarity and proper synchronization between threads. Each (CardDeck) synchronizes its own methods, ensuring that drawn and discard operations are atomic and safe from concurrent access. While this introduces slight contention, it is appropriate for the game’s size and prevents race conditions.

Game termination is handled with a shared (AtomicBoolean) that marks the global state. The first player to meet the win condition changes this value atomically, ensuring that only one thread declares the end and that all others detect it immediately.

Furthermore, the use of volatile fields like (WINNER_ID) ensures that threads always see the most recent values, maintaining memory consistency. Finally, each player writes to its own file, isolating input/output operations and preventing interference between threads, ensuring clear and orderly exits.

Error handling

The program's error handling focuses on preventing failures before the game starts and ensuring safe termination in case of exceptions. Before the game begins, the system validates that the input parameters (the number of players and the card pack file) are correct. If the user enters an invalid value, an incorrect format, or a missing file, the program displays a descriptive error message in the console and requests the data again, thus preventing the game from starting in an inconsistent state.

It also checks that all cards are non-negative integers and that the file contains exactly "8n" lines, according to the specification. Otherwise, the program reports a new file. During read and write operations, "try-with-resources blocks" are used, which ensure the automatic and safe closure of files, even if an exception occurs. This structured error management maintains system robustness, prevents data loss, and ensures that all resources are released correctly.

Tests Design

Testing framework

All tests were developed using the Junit 5 (Jupiter) framework, which offers a modern, modular syntax compatible with concurrent environments. The (@Test), (@BeforeEach), and (@AfterEach) annotations were used to structure the test cases, ensuring systematic setup and cleanup before and after each execution. Assertions were implemented using methods from the "org.junit.jupiter.api.Assertions" class, allowing for precise validation of the program's expected conditions. Furthermore, temporary files were used to isolate the execution of each test, and reflection methods were used to reset the global static variables (GAME_OVER and WINNER_ID), thus guaranteeing that the results were independent and reproducible across cases.

Tests suite structure

The tests suite is divided into three main classes that cover different levels of verification: unit, integration, and concurrency. Below a list will show the classes and their functions.

- CardDeckTest: Checks the FIFO behavior of the decks and their security in multithreaded environments. It includes a stress test with producer and consumer threads running in parallel, ensuring that the total number of cards processed is consistent and that there are no losses or duplicates. This test confirms the correct synchronization of the draw and discard methods, as well as the atomicity of operations within the deck.
- PlayerTest: Evaluates individual player behavior in a controlled manner. It verifies that the hand always contains four cards, that the player never discards a card of

their preferred value, and that a win condition is correctly detected when all four cards are the same. To ensure independence between tests, reflection is used to reset the game's static variables before and after each execution.

- CardGameTest: Implements end-to-end tests that reproduce the complete game flow. In the valid scenario, a pre-prepared pack file is used to produce a predictable winner, verifying the creation of all output files (playerX_output.txt and deckX_output.txt) and confirming that only one winner is declared. Another scenario simulates an invalid input followed by a valid one, testing the system's robustness against user errors. “System.in” and “System.out” are captured to validate console messages, and global states are reset to maintain independence between executions.

Coverage

The test suite achieves complete functional coverage of the system's main components.

The (Card) class is tested deterministically to validate object immutability and the correct handling of invalid values in the constructor. (CardDeck) is evaluated using concurrent scenarios that ensure its thread safety and compliance with the FIFO policy. (Player) undergoes unit testing that verifies the discard rule, win detection, and hand integrity at all times.

Finally, CardGame is tested end-to-end to verify the consistency of the overall flow, the generation of correct outputs, and the detection of a single winner. Together, these tests ensure that the system correctly handles concurrency, input/output, and overall game state control.

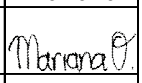
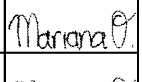
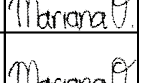
Test isolation and reproducibility

Each test runs in an isolated environment, rebuilding the necessary objects and files from scratch. Temporary directories created using “Files.createTempDirectory” are employed to prevent conflicts between runs and ensure automatic environment cleanup upon completion. Global variables (GAME_OVER and WINNER_ID) are reset before and after each test, guaranteeing that the game state does not propagate between instances. The design avoids any dependency on timing or operating system scheduling, eliminating the need for artificial pauses. This makes the results fully reproducible and consistent across runs.

Limitations

The test suite has certain known limitations stemming from the constraints of the problem statement. Cases where two players win simultaneously are not evaluated, as the specification explicitly states that handling this situation is unnecessary. Finally, differences in thread scheduling are tolerated by verifying the final game states, rather than attempting to reproduce exact execution sequences.

Development Log

Date	Time		Duration	Roles		Signatures	
				Driver	Navigator	Yanbin	Mariana
10/10/2025	10:00:00 a. m.	11:00:00 a. m.	1 hour	Mariana	Yanbin	YanBinH	
15/10/2025	10:00:00 a. m.	11:00:00 a. m.	2 hour	Yanbin	Mariana	YanBinH	
22/10/2025	10:00:00 a. m.	11:00:00 a. m.	3 hour	Mariana	Yanbin	YanBInH	
29/10/2025	10:00:00 a. m.	11:00:00 a. m.	4 hour	Yanbin	Mariana	YanBinH	