Card Game Simulation - Coursework Report

1. Production Code Design and Performance

1.1.1 Overall Architecture

The program is divided into small classes, each handling a distinct part of the game. We decided to make these divisions to improve readability and maintainability. These are:

- Card represents a single immutable playing card with a non-negative integer value. It is thread-safe because its value never changes after creation.

- Deck stores a queue of cards and uses a ReentrantLock with a Condition to coordinate concurrent access. Only one player can draw or discard from a deck at a time. Each deck sits between two players and connects their actions.

- Player runs on its own thread. It draws a card from its left deck, discards to its right deck, and checks whether its hand of four cards shares the same value. Each player maintains its own hand and output file, with no direct access to any other player's data.

- CardGame initializes the simulation. It reads the pack file, creates decks and players, distributes cards, starts all threads, and terminates the game once a winner is declared. It manages global flags that record whether the game has been won and which player won.

- PackReader handles input validation and parsing. It reads the text file, ensures all lines contain non-negative integers, and checks that the total number of cards matches 8 × number of players.

Program flow:

1. PackReader reads and validates the input file.
2. CardGame creates decks and players using the parsed data.
3. Each Player thread repeatedly draws from its left deck and discards to its right until a winner is found.
4. CardGame detects the win, signals all threads to finish, and writes final deck outputs.

1.1.2 Threading and Concurrency Model

Each player executes as an independent thread, simulating concurrent gameplay. Threads interact through shared Deck objects.

Each deck maintains a private ReentrantLock to ensure thread-safe access. When a player needs to draw or discard, it locks the relevant deck, performs the operation, and unlocks it. If the deck is empty, the player waits on a Condition (notEmpty) associated with that lock. When another player adds a card, the condition is signaled to wake waiting threads.

To prevent deadlocks, players acquire deck locks in ascending deck-ID order. This guarantees a consistent global locking order and avoids circular waits.

Thread termination is coordinated using shared volatile flags (gameWon and winnerId) stored in CardGame. All threads check these flags periodically to exit cleanly when a winner is declared. This approach avoids data races, ensures consistent deck updates, and allows multiple decks to operate in parallel when unrelated threads access different decks.

1.1.3 Error Handling and Robustness

Input checks are handled in PackReader. It checks that the file exists, contains only non-negative numbers, and has the correct number of cards. If the file is invalid, the user is asked to try again until a valid pack is provided.

During play, shared state changes are protected. CardGame.tryDeclareWin() is synchronized so only one player can declare a win.

Each Player uses try-with-resources for file writing and ends cleanly if an error or interruption occurs. Errors are handled within their class so the rest of the game continues safely

1.1.4 Extensibility and Scalability

The modular structure enables easy extension. It can support virtually any player count (memory and time allowing), and makes changes to logging formats relatively simple due to the clear file structure. Runtime grows approximately linearly with the number of players under moderate load, but synchronization and I/O operations could introduce limits on scalability.
Scalability and performance considerations:

- *Per-deck lock contention*: Each deck is shared by only two players, so contention per deck stays roughly constant as player count grows.
- *I/O throughput*: Frequent writes to each output file can slow the game, since every player and deck has its own file.
- *Output file count*: The number of files grows with players and decks, which can become unwieldy for large games

1.2 Performance Considerations

Thread Contention

Each player runs on its own thread and uses locks on its two adjacent decks for draw and discard actions. Decks use a fixed lock order to avoid deadlocks. Only neighboring players share a deck, so contention stays local. More players mean more waiting, but unrelated decks still run in parallel.

I/O Latency

Each player writes their actions to their own output file using PrintWriter. This makes the output easy to read and keeps players from interfering with each other's files. However, because the program writes to the file many times during the game, these disk operations can slow things down compared to doing all the writing at the end. This puts the focus on clarity and correctness rather than raw speed. One improvement would be to collect several lines of output and write them all at once to reduce file-access time.

Scalability and Theoretical Complexity

Each draw or discard takes constant time (assuming there are no locked decks), so one full round scales somewhat linearly with the number of players. Total runtime varies with card distribution and turn order. Memory use also grows with player count, as each stores a small hand and two deck links. Actual speed depends on JVM thread scheduling, deck contention, and I/O performance.

Possible Improvements

- Use a built-in thread-safe queue (for example, BlockingQueue<Card>) instead of manually managing locks to simplify the code.
- Use a BufferedWriter or a background logger so the program writes several lines at once instead of one at a time, reducing file-writing delays.
- Use a profiler to see whether most of the time is spent waiting on locks or writing to files, and adjust the design based on that result.

2. Test Design and Strategy

2.1 Testing Framework

We used JUnit 5.14.0 for both unit and integration testing.
Tests are organized per class to keep them clear and focused. For example:

- CardTest verifies card creation, value access, equality, and immutability.
- DeckTest checks adding and removing cards and retrieving deck contents.
- PlayerTest ensures players draw and discard correctly, selecting cards according to their strategy, and correctly detecting a winning hand.
- PackReaderTest validates reading packs from files, including handling invalid input, incorrect file sizes, and non-integer values.

Each test checks functionality in isolation. For multiple class interactions, we used integration tests to verify components worked together correctly. This aims to ensure both individual components and the system as a whole behave as expected.

2.2 Unit Testing

Tests verify each class in isolation with defined inputs and expected outputs.

- ***CardTest Class***

Method	Input	Expected Output	Case
Card(int value)	5	Card created with value 5	Normal case
Card(int value)	0	Card created with value 0	Edge case
Card(int value)	-1	Throws IllegalArgumentException	Invalid input
getValue()	Card(5)	5	Returns card value
equals(Object obj)	Card(5) vs Card(5)	true	Equal cards
toString()	Card(7)	"7"	String representation of value

- ***DeckTest Class***

Method	Input / Scenario	Expected Output / Behavior	Note
Deck(int deckId)	Create a deck with ID 1	Deck created with ID 1	Constructor requires deck ID
addCard(Card) + pollFirst()	Deck empty, add Card(1), add Card(2), then pollFirst()	Returns Card(1), deck becomes [Card(2)]	Normal FIFO behavior
getContents()	Deck contains [Card(1), Card(2)]	Returns [Card(1), Card(2)]	Snapshot of deck
lock() / unlock()	Simulated concurrent add/removal by multiple threads	No cards lost or duplicated	Ensures thread safety and correct locking

- ***PackReaderTest Class***

Method	Input / Scenario	Expected Output / Behavior	Note
readPack(String path, int numPlayers)	File contains 0,1,2,...,7 repeated for 2 players (16 cards)	Returns list of 16 Card objects	Normal valid pack
readPack(String path, int numPlayers)	File contains negative number or non-integer	Throws IllegalArgumentException	Invalid card values
readPack(String path, int numPlayers)	File contains wrong number of cards	Throws IllegalArgumentException	Invalid pack size

- ***PlayerTest Class (unit aspects)***

Method / Feature	Input / Scenario	Expected Output / Behavior	Note
addCard() + handSnapshot()	Start with hand [1,2,1,1], call addCard(3)	handSnapshot() returns "1 2 1 1 3"	Tests adding a card to hand
Winning hand detection	Hand [3,3,3,3]	handSnapshot() shows "3 3 3 3"; player run loop declares win	Tests that player recognizes a winning hand using public methods only

2.3 Integration Testing

Tests confirm correct interaction between classes in game scenarios.

- ***Player Behavior (PlayerDeckIntegrationTest)***

Feature / Method	Input / Scenario	Expected Output / Behavior	Notes
Drawing and discarding	Player hand [Card(1), Card(2), Card(1), Card(1)], left deck [Card(3)], right deck empty	Player draws Card(3) from left deck, discards Card(2) to right deck, hand becomes [Card(1), Card(1), Card(1), Card(3)]	Tests round-robin draw/discard logic

- ***Game Flow***

Feature / Scenario	Input	Expected Output / Behavior	Notes
Small-scale 2-player game	Pack [0,1,2,3,0,1,2,3,4,5,6,7,4,5,6,7], 2 players, decks initialized empty	Each player draws and discards correctly in round-robin; output files contain lines like "player 1 draws a 0 from deck 1" and "player 1 discards a 1 to deck 2"; only one player wins ("player X wins"); all threads terminate and final hands are logged correctly	Confirms overall game logic, deck sharing, and proper thread termination
Winner declaration race	Two players both reach a winning hand around the same time	Only one player's tryDeclareWin() call succeeds; the other detects gameWon = true and exits gracefully	Verifies correct synchronization and atomic winner declaration

2.4 Concurrency Testing

Tests ensure thread safety, deadlock avoidance in tested scenarios, and proper game termination with multiple players.

Feature / Scenario	Input	Expected Output / Behavior	Notes
Simultaneous Player Actions	Two players access the same deck concurrently:• Player 1 hand = [1, 2, 3, 4]• Player 2 hand = [5, 6, 7, 8]• Left Deck = [9, 10], Right Deck = empty	Both players draw and discard without card loss or duplication; deck contents remain consistent; no deadlocks occur; both threads complete successfully.	Confirms deck locking and signalling work as intended under concurrency.
Deadlock Prevention when Draw/Discarding	Multiple players attempt to lock left and right decks simultaneously.	Lock acquisition order follows deterministic rule (lower deck ID first); game continues without deadlock, all threads eventually release locks and complete.	Verifies correct use of ordered locking (firstLock, secondLock) in Player.run().
Game Termination Under Concurrency	Two players reach a winning hand nearly simultaneously.	Only one player's tryDeclareWin() succeeds; shared gameWon flag set; other players detect game end and terminate; output files show consistent final hands and exits.	Confirms atomic win declaration and proper shutdown of all threads.

2.5 I/O Testing

Tests check that player and deck output files are correctly formatted, and reflect game actions.

Feature / Scenario	Input	Expected Output / Behavior	Notes
Initial Hand Logging	Player starts with hand [Card(1), Card(2), Card(3), Card(4)]	Output file begins with:player 1 initial hand 1 2 3 4	Confirms initial hand state is logged before any gameplay actions.
Deck Output Format	Game ends and deck contents are written to file	Output file for each deck includes:deck2 contents:*cards*	Ensures proper final deck logging format and readability.

3. Development Log

Dev A = Ben Lloyd   Candidate Number: 312807
Dev B = Sylvester Koroma   Candidate Number: 312402

Date	Time	Activities	Roles	Signatures
01 Oct 2025	14:00 - 17:00	Initial project setup. Planned architecture for Card, Deck, Player, and CardGame. Created repository and base package structure.	Dev A – Driver Dev B – Navigator	Ben L 312807 [illegible] 312402
06 Oct 2025	13:00 - 16:00	Implemented Card and Deck classes. Planned locking and condition logic.	Dev B – Driver Dev A – Navigator	Ben L 312807 [illegible] 312402
12 Oct 2025	10:00 - 13:00	Completed PackReader with input validation. Planned initial tests	Dev A – Driver Dev B – Navigator	Ben L 312807 [illegible] 312402
18 Oct 2025	14:00 - 17:00	Implemented main Player logic: draw/discard loop, hand management, and file logging	Dev B – Driver Dev A – Navigator	Ben L 312807 [illegible] 312402
25 Oct 2025	11:00 - 14:00	Integrated all components in CardGame. Finalised and ran all tests.	Dev A – Driver Dev B – Navigator	Ben L 312807 [illegible] 312402
29 Oct 2025	09:00 - 11:30	Final debugging, output verification, and report preparation ready for submission.	Both collaborated equally	Ben L 312807 [illegible] 312402