Part 1: Production Code Design Choices

1.1 Overall Architecture Design

Our card game implementation follows a modular, object-oriented design with four core classes: Card, CardDeck, Player, and CardGame. It was built with thread safety in mind in order to simulate a concurrent card game and meet the requirements laid out.

Key Design Principles:

- Class separation: Each class has a single responsibility
- Thread Safety: Shared resources are protected using synchronisation mechanisms
- Immutability: immutable classes to prevent concurrent modification issues
- Dual player actions: Players act as both producers and consumers of cards through deck operations

1.2 Card Class Design

The Card class implements an immutable value object pattern. We chose immutability to remove thread safety concerns when cards are shared between multiple threads. The class includes proper equals() and hashCode() implementations to support collections.

Design Rationale:

- Immutable fields prevent accidental modification during concurrent access
- Keeping it simple - integer denomination for easy comparison in game logic
- Use of hashcodes as we override equals

1.3 CardDeck Class - Thread Safety Strategy

The CardDeck was the most complex design challenge due to concurrent access requirements. We use a synchronisation approach by both ConcurrentLinkedQueue and ReentrantLock.

Synchronisation Strategy:

- ConcurrentLinkedQueue Provides a thread safe FIFO operations for the card storage
- ReentrantLock: ensures atomic operations
- Lock acquisition follows a same pattern throughout in order to prevent deadlocks

1.4 Player Class - Threading Model

Player extends the thread and implements the complete game strategy as an independent concurrently on its own. The class manages its own hand, game state and file output while managing with the shared resources like deck.

Key Design Features:

- Hand Management: Thread safe hand operations by using ReentrantLock
- Game Strategy: Prefer to keep cards of the player’s current number
- Atomic Game State: Prevent race conditions
- File IO: Individual output files to stop hassle between players

Threading Considerations:

- Each player thread operates independently apart shared deck access
- Win condition uses compare and set to handle simultaneous wins
- Graceful shutdown when game ends

1.5 CardGame Class - Main layer

The CardGame class acts as the primary coordinator by handling initialisation, input validation, and game management. This design keeps robustness and makes the control centralized.

Input Validation Strategy:

- Check pack files are valid
- Interactive input using retry mechanism for invalid data
- Error handling with informative messages so mistakes are clear

Game Flow Management:

- Round robin card distribution to keep game fair
- Ring topology for deck access
- Proper thread management using join operations

1.6 Known Performance Issues

Contention Points:

1. Deck Lock Contention: At high player count this may cause bottlenecks at popular decks
2. File I/O Blocking: Player output writing can cause stalls
3. Memory Usage: Large pack files consume significant heap space while being initialized

Mitigation Strategies:

- Minimal sections in deck operations
- Using buffered writers for output files
- Short sleep intervals

Scalability Limitations:

The current design scales well up to approximately 50-100 players before lock contention becomes significant. Beyond this, a lock-free approach or partitioned deck strategy would be necessary. A horizontal scaling approach would increase scalability but would be significantly more complex to implement.

Part 2: Test Design Choices

2.1 Testing Framework Selection

Framework Used: JUnit 5

Rationale: We chose JUnit 5 as our testing framework for several compelling reasons:

1. Industry Standard: JUnit 5 is the most widely used testing framework in Java development
2. Modern Features: advanced annotations, well parameterised tests.
3. IDE Integration: Excellent support in all major IDEs for running and debugging tests
4. Comprehensive Documentation: As it's so widely used there are many resources
5. Easy install: Via extensions or built in support.

2.2 Test Architecture and Organisation

Our tests follows a hierarchical structure with four main test classes and a comprehensive test runner using JUnit 5:

Test Class Hierarchy:

- CardTest: Tests for Card class functionality
- CardDeckTest: Concurrency tests for CardDeck operations
- PlayerTest: Player behavior and threading tests
- CardGameTest: Tests for complete game scenarios
- CardGameTestSuite: Suite that orchestrates all tests and reporting

2.3 Unit Testing Strategy

2.3.1 Card Class Testing

Coverage: 8 comprehensive test methods covering:

- Verify boundary value are followed (non negative)
- Immutability verification
- Equality and hash code consistency
- String representation accuracy
- Exception handling for invalid inputs

2.3.2 CardDeck Class Testing

Coverage: 6 test methods addressing:

- FIFO operations
- Thread safety width concurrent access
- Empty deck behavior
- File output format verification
- Null handling and error conditions

2.3.3 Player Class Testing

Coverage: 8 test methods covering:

- Player initialisation and state management
- Hand managing and its capacity limits
- Win conditions

- Threads and termination
- File output generation and format
- Game strategy working

2.4 Integration Testing Approach

Our integration tests in CardGameTest simulate complete game scenarios to verify system-wide behavior:

Test Scenarios Covered:

1. Two Player instant win: Checks immediate win detection
2. Four Player long game: Tests strategy over longer game
3. Invalid Input Handling: Pack file validation
4. File Output Verification: Ensures all output files are correct
5. Niche Edge Cases: Empty decks, simultaneous actions for thread coordination

File-Based Testing:

Tests to create temporary pack files and check output file generation:

- Temporary pack files for controlled scenarios
- Output file content verification
- Automatic cleanup of test packs

2.5 Concurrency Testing Strategy

Thread Safety Verification:

Our concurrency tests use several approaches to verify thread safety:

1. Stress Testing: High volume of concurrent operations to verify no race conditions
2. Atomic operations verification: To ensure that operations remain atomic
3. State consistency checks: Verify object state remains consistent under concurrent access
4. Deadlock Detection: Longer tests to identify potential deadlocks

JUnit 5 Concurrency Features:

- We utilize the @Timeout annotation to prevent infinite test execution
- It has built in support for the ExecutorService testing patterns
- Parallel tests with @Execution(CONCURRENT)
- exception handling for multi-threaded scenarios

2.6 Test Data Management

Test Data Strategy:

- Dynamic test generation: Most test data is generated in code for reliability
- Temporary Files: Integration tests create temporary pack files with content we know
- Cleanup Procedures: Automatically cleanup of all test files
- Isolation: Each test method operates on newly instantiated objects to prevent interference

2.7 Test Coverage and Metrics

Coverage Statistics:

- Total Test Methods: over 30 tests cases across all classes
- Code Coverage: Most code is tested
- Exception Paths: All exception mostly checked

JUnit 5 Test Execution:

- Execution Time: 3-5 seconds for complete suite
- Success Rate: 100% on our tests
- Concurrency Tests: 10 threads performing 100 operations each.

Our testing suite provides confidence in the implementation's correctness. Which helps us confirm accuracy under various operating conditions, using essential industry tools like JUnit that help us spot errors efficiently.

Part 3: Development Log - Pair Programming Log Summary

3.1 Pair Programming Approach
We utilised a collaborative approach in which we both contributed to most components while frequently swapping between driver and navigator in several of our sessions which helped us both understand the codebase more thoroughly.
29/09, 10:00–13:00 (3h) *(Ben: Driver, Henry: Navigator)*
30/09, 14:00–16:00 (2h). *(Henry: Driver, Ben: Navigator)*
Signed: B.Pritchard / H.Alexander

3.2 Card Class Development
Henry implemented the hash checking, as this class is relatively simple and easy to understand. We moved on from this quickly.
02/10, 13:00–15:00 (2h) *(Henry: Driver, Ben: Navigator)*
Signed: B.Pritchard / H.Alexander

3.3 Card Deck Development
As this class is utilised more thoroughly through the program and requires thread safety it was more important to ensure proper implementation. Ben led the development by building the main parts of the class and methods, while Henry implemented the writeToFile and custom toString methods.
05/10, 10:00–13:00 (3h *(Ben: Driver, Henry: Navigator)*
06/10, 14:00–16:00 (2h) *(Henry: Driver, Ben: Navigator)*
Signed: B.Pritchard / H.Alexander

3.4 Player Class Development
Thread safety is a key concern so it was essential this was done correctly. Henry focused on implementing the initialization, file writing, adding and removing cards from hand while Ben handled winning, gameover and performing a turn.
10/10, 11:00–14:00 (3h) *(Henry: Driver, Ben: Navigator)*
12/10, 13:00–16:00 (3h) *(Ben: Driver, Henry: Navigator)*
Signed: B. / H.Alexander

3.5 Card Game Development
The class serves as the main coordinator for the games flow. Ben handled pack validation and game termination logic. While Henry focused on card distribution, player count verification, filling decks, and starting the game.
18/10, 09:00–12:00 (3h) *(Henry: Navigator, Ben: Driver)*
19/10, 09:00–12:00 (3h) *(Ben: Navigator, Henry: Driver)*
Signed: B.Pritchard / H.Alexander

3.6 Testing Suite Development
These were made together as it was important to be creative and make sure we didn't miss any edge cases. More complex testing involving thread safety was done by Ben, while Henry created the main scenario tests for concurrency.
24/10, 14:00–17:00 (3h) *(Henry: Navigator, Ben: Driver)*
25/10, 14:00–17:00 (3h)(*Henry: Driver, Ben: Navigator*)
Signed: B.Pritchard / H.Alexander

3.7 Debugging and Optimisation
After we had implemented the main logic, we had a dedicated debugging and optimization session to make our program run efficiently and fix potentially overlooked errors. Our largest challenge was resolving deadlock issues in deck access and race conditions in win detection. Ben addressed the thread synchronization problems by implementing proper locking mechanisms. Henry optimised performance through reducing unneeded thread sleep times.
29/10, 15:00–18:00 (3h) *(Ben: Driver, Henry: Driver)*
Signed: B.Pritchard / H.Alexander

Part 4 Summary and Reflection

As neither of us had large amounts of java or OOP experience before university, developing thread safe programs has taught us about handling issues that we hadn't encountered as most code we've written was mostly single threaded. Combining this with collaborative programming in an unfamiliar language helped us get a deeper understanding of java and furthered our understanding of OOP which is a critical skill to be knowledgeable in. Implementing testing scenarios with JUnit also taught us the importance of test driven development and helped us uncover slight oversights that would have been missed otherwise.