ECM2414 Software Development CW Project Report

1 Cover Page

Students	730047916	730020353
Weight	50%	50%

1.1 Development log

Date	Time	Duration	730047916	730020353
22/10/24	19:00	1h 30min	Planning	Planning
24/10/24	21:30	2h	Planning	Planning
04/11/24	18:00	2h 30min	Observer	Driver
06/11/24	20:30	3h 30min	Driver	Observer
09/11/24	19:00	1h 30min	Observer	Driver
15/11/24	17:30	2h 30min	Driver	Observer
19/11/24	21:00	2h	Observer	Driver
26/11/24	20:00	3h	Driver	Observer
28/11/24	19:30	2h 30min	Observer	Driver
02/12/24	20:30	3h	Driver	Observer
04/12/24	21:00	1h 30min	Documentation	Documentation
08/12/24	16:30	10h	Documentation	Documentation

2 Architectural design choices

The architecture of this software is designed around modularity and thread safety ensuring a robust design. Players interact with shared resources like CardDeck instances under strict synchronisation maintaining data integrity. Each class has a specific role: Card class represents the fundamental game element, CardDeck class manages the shared card queues with thread-safe locking mechanisms and the Player class handles player specific logic such as card management and gameplay. The CardGame class acts as a central hub for the game initialising players and decks. The Utils class provides reusable methods for file handling and data formatting. The design combines modularity and concurrency, ensuring each component interacts seamlessly while maintaining thread-safety.

Class Report

1.1 Card Class

- Each instance of the Card class represents an individual card in the game held by either a player or a deck. The class has two attributes: value and age. Value is immutable while age can be incremented. While the class itself isn't thread safe as it allows for concurrent modification of the age field without any safeguard, it relies on external control from other classes to ensure no concurrent access using locks.

1.2 CardDeck Class

- The CardDeck class manages the decks placed between players, ensuring thread-safe operations during card draws and discards. Thread safety is ensured by controlled access and modification through a ReentrantReadWriteLock.WriteLock. The cards in the deck are managed using a queue, that follows **FIFO (First In, First Out)** order, allowing players to add, remove or log cards while preventing concurrent access. The class provides methods to add, retrieve, and manipulate cards within the deck, and it outputs the deck's contents to a file when the game ends. This design ensures synchronisation, safe interactions in a multi-threaded environment.

1.3 Player Class

- The player class is a participant in the multi-threaded card game designed to interact with CardDeck instances in a thread safe manner. Thread safety is achieved by using exclusive locks on the CardDeck objects during the draw and discard operations ensuring no two players access the same CardDeck object simultaneously. Every player prefers cards that match their index (e.g.: player 1 will prefer cards with value of 1, etc). To avoid cards remaining in a player's hand indefinitely, the class tracks the age of each card, incrementing it after every round. The class then prioritises removing the oldest non-preferred card from their hand. Players are notified of a win event when another player wins the game, and the winning player exits the game. During this process logs are generated, capturing actions such as drawing and discarding cards and final game states. These logs are written to individual player specific files using the Utils class.

1.4 CardGame Class

- The CardGame class is made up of entirely of static methods and fields designed to serve as the central hub **(The Main Class)** for the game. It is responsible for managing user input and initialising the game. The class stores all players and decks involved in the game and its primary role includes preparing the game environment by loading the card pack assigning cards to players and decks and setting up players with their respective draw and discard decks. The CardGame class handles launching the game by starting the player threads and enabling each player to execute their gameplay logic concurrently. Players use a synchronised callback method to signal when they win the game. This ensures thread-safe handling of the win event and prevents double outputs if multiple players attempt to win at the same time. The win event triggers the player to exit the game and for the rest of the player to carry on until all players achieve a win condition. The CardGame class provides a clear and structured framework for managing the multi-threaded game.

1.5 Utils Class

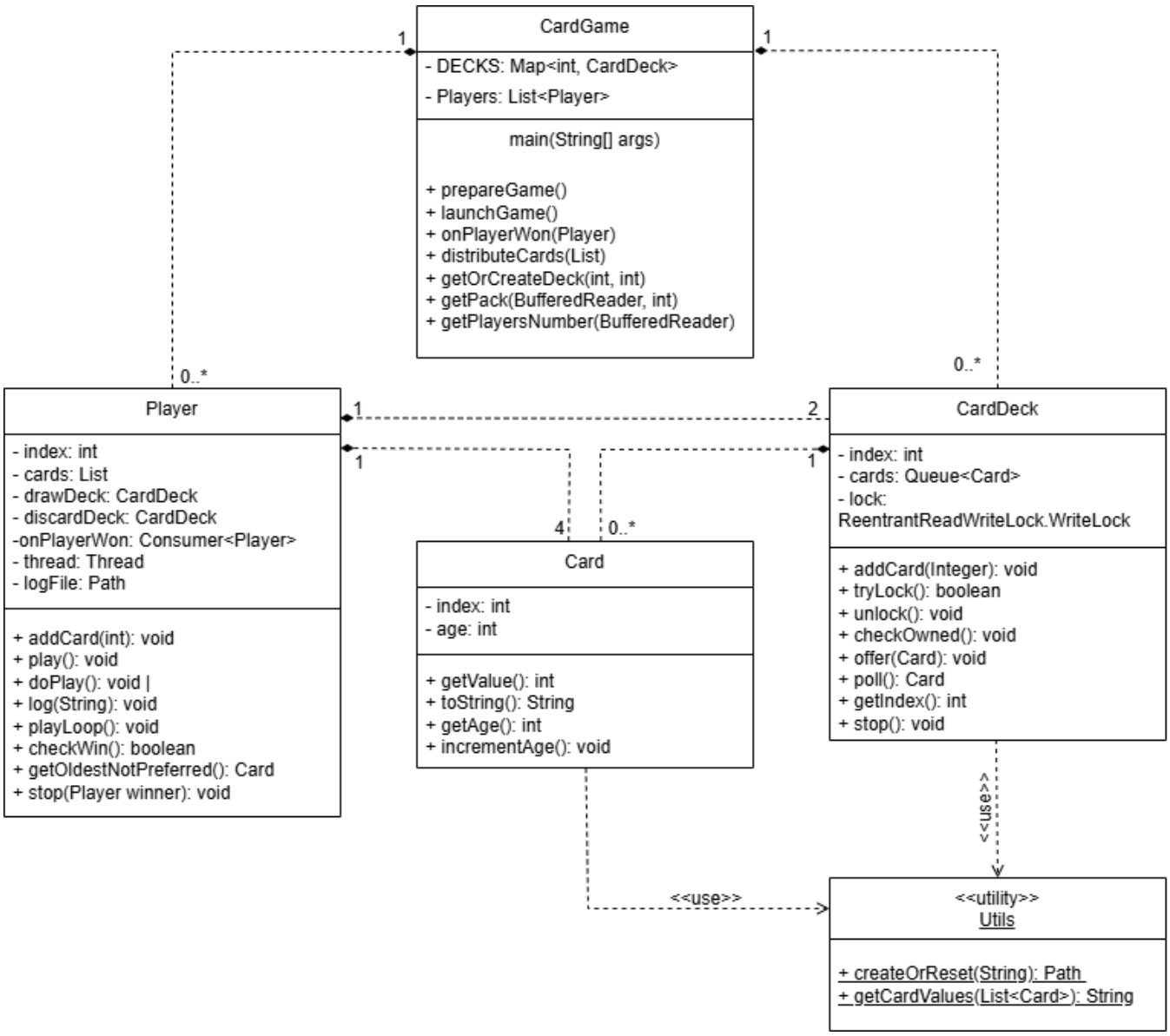
- The Utils class provides utility methods to simplify repetitive tasks across the program. It formats card values for logging and ensures log files are either created or reset at the start of a new game. These methods streamline file handling and data presentation. By centralising repetitive operations such as formatting card values and managing file operations, Utils ensures code reusability and clarity. The Utils class itself is a static

class (<<utility>>) as no Utils objects can be created and exists to serve as a utility class for the Player and CardDeck classes.

2 Performance issues:

The Java Virtual Machine has a limit on the number of concurrent threads capped at 256. As each player is represented by a separate thread, the maximum number of players that can play concurrently is 256. If this limit is exceeded, it could lead to thread contention or the game crashing.

UML Diagram



Testing Report

1 Framework Used: JUnit5

The Card Game project was tested using the JUnit 5 framework. Because of its flexibility, compatibility with Java 8 and subsequent versions, and modern features, JUnit 5 was selected. It is the perfect option for testing the logic, concurrency, and file-handling capabilities of our project because of its modular architecture and support for parameterised tests and assertions.

2 Automated Testing

We implemented automated tests to validate the correctness and functionality of the card game system. The tests targeted core classes like Card, CardDeck, CardGame and Player. Java reflection was used where necessary to access private methods and fields enabling in-depth testing of internal states and behaviours.

2.1 CardTest

- The CardTest class ensures the core functionality of the Card class is reliable by testing its key methods. The **getValue()** test verifies that the card's value is correctly set and retrieved, while the **getAge()** test confirms the initial age is 0. The **incrementAge()** test validates that calling the method increases the card's age by 1. A **@BeforeEach** setup initialises a card with a value of 10 before each test, ensuring consistency and isolation. Using clear assertions, the test class effectively validates the Card class's behaviour, ensuring its properties and methods work as intended.

2.2 CardDeckTest

- The CardDeckTest class provides comprehensive validation of the CardDeck class, covering functionalities such as locking mechanisms, card operations, and file logging. **@BeforeEach** initialises a new deck object with index 1 and a new card object with value 5 before each test to ensure consistency and isolation with each test. Tests like **addCardTest()**, **pollTest()**, and **offerTest()** ensure proper handling of cards when the deck is locked, while **tryLockTest()** and **unlockTest()** validate thread-safe access. Edge cases, including unauthorised operations (**pollWithoutLockTest()**) and empty deck scenarios, are thoroughly tested. File logging is verified through tests like **stopTest()** and **playerDeckContentTest(),** which confirm that deck contents are correctly logged. The use of reflection in **checkOwnedTest()** ensures private methods function as intended. With its focus on edge cases and error handling, the test class ensures the CardDeck is robust and reliable, though mocking file operations and parameterised tests could enhance efficiency and scalability.

2.3 PlayerTest

- The PlayerTest class provides comprehensive testing for the Player class, It focuses on the gameplay mechanics, synchronisation and threading with the CardDeck objects. **@BeforeEach** initialises two new card decks, one discard deck and one draw deck, and creates a player instance with those two decks as that players draw and discard deck. The program then creates a log file for this player instance. This ensures the tests are consistent. The **playCheckWinTest()** function simulates a win condition by adding 4 identical cards to the players hand and checks if the OnPlayerWon callback is called. The **addCard()** test checks to see if cards get correctly added to the players hand and **playTest()** checks that the players thread initialises properly and interacts with the decks during gameplay. The threads behaviour is further checked in **playLoopInterruptedTest()** which makes sure that the players thread responds to stops and interruptions properly. The **playerWaitUntilDrawDeckUnlocked()** test tests the synchronisation of the class by checking if the player waits appropriately when a drawDeck is locked. Finally, The **stop()** test checks that the game correctly logs deck contents to a file upon termination. The PlayerTest class ensures robust functionality and reliability for the Player class in a multi-threaded environment.

2.4 UtilsTest

- The UtilsTest class verifies the functionality of the Utils class, focusing on card value formatting and file handling. The **getCardValues()** test ensures that a list of Card objects is correctly converted into a space-separated string of values for logging or display. The **createOrReset()** test confirms that the method creates a new file or clears the contents of an existing one, ensuring the file is empty afterward and the returned Path matches the input.

2.5 CardGameTest

- The CardGameTest class focuses on testing input validation and error handling during the initialisation of the CardGame. The primary test, **PlayerCountMismatchTest()**, simulates a scenario where the number of players does not match the size of the provided card pack. It redirects System.in to simulate user input and captures System. Out to validate that the program outputs the expected error message ("Invalid file"). The test ensures the game enforces its rule that the card pack size must equal 8 * n of players. By verifying file existence and program output, it highlights the program's robustness in handling invalid input. Improvements could include dynamically creating test files and expanding test scenarios to cover more edge cases.

2.6 ReflexionUtils

- The ReflexionUtils class provides utility methods for accessing private fields and methods using Java Reflection, making it invaluable for testing and debugging. The invoke method dynamically locates and executes private or protected methods, while getValue retrieves the value of private fields by temporarily overriding access controls. These features enable thorough testing of internal states and behaviours, such as inspecting a player's cards list or invoking private methods like checkOwned in CardDeck. It offers flexibility for various use cases. Its usage should be restricted to testing scenarios to preserve encapsulation in production code.

3 Manual Testing:

To test the round robin distribution of cards, a card pack file was created which intentionally makes player two win (**2_win.txt**). This file contains the same value card every 4 cards. This ensures that 1 of the players gets the same 4 cards in their hand if the cards are correctly distributed as per the specification. If the distribution method is correct then player 2 should win straight away which it does, confirming that it has passed the test. Larger card packs were made to test if the program could handle different sizes of players and card decks. The tests also checked if the program worked properly with different numbers of threads by using various player counts.