Cover Page

Mark Split

50:50

Development Log

Date	Time	Duration of a session	Role 1	Signature	Role 2	Signature
31/10 /2023	17:00	5 hours	Driver	225743	Observer	195810
03/11 /2023	18:00	6 hours	Driver	225743	Observer	195810
07/11 /2023	16:00	7 hour	Observer	225743	Driver	195810
07/11 /2023	20:00	4 hours	Observer	225743	Driver	195810
10/11 /2023	20:30	2 hours	Driver	225743	Observer	195810
13/11 /2023	18:00	4 hours	Driver	225743	Observer	195810
20/11 /2023	13:00	8 hours	Observer	225743	Driver	195810
25/11 /2023	13:00	7 hours	Driver	225743	Observer	195810
29/11 /2023	13:00	7 hours	Observer	225743	Driver	195810

Card Game Design Documentation

Introduction.

The architecture and design of classes play an important role in the game development and overall performance of the application. This report explains how classes such as Card, CardDeck, CardGame, GameActionsLogger, GamePlayException, GameTools and Player have been designed to maximise the performance of the CardGame application. It will dive into each class and demonstrate the reason of how the class was structured, methods it provides as well as its contribution to the game's functionality and common performance issues it may face.

Design Choices

1. Card Class

The class represents a unit of the game which is a playing card. The only private field it has is its denomination. The class has methods that are directly related to the card's denomination such as accessing the information about it. The design of the class is simple therefore the usage is very straightforward and it ensures low usage of memory. It is a fundamental element of the game and it is used to create decks of cards. It is a lightweight class, therefore it works efficiently and minimises any performance issues.

2. CardDeck Class

The class represents a unit of the game which is a deck of cards. The deck of card is basically a collection of Card objects in the list. The List<Card> was chosen for efficient management of cards in a deck. The methods of the class include main methods which are drawing and discarding cards from a deck as well as managing a deck's state. During the game more than one decks of cards are used therefore synchronised methods were implemented. It ensures thread safety and would not allow access to decks simultaneously.

3. CardGame Class

The class represents an actual process of the game. It includes attributes and methods to manage the game's process, players and decks. It uses List<Player> and List<CardDeck> for efficient management. Synchronised methods have also been used in the CardGame class because it is vital for the multithreaded environment. It controls actions of players in the gameplay as well as conditions of decks. However, in the case of CardGame class, synchronised methods may lead to performance bottlenecks, particularly if the number of decks and players is scaling up.

4. **GameActionsLogger Class**

The class represents functions of logging to track and record actions in the game. Methods of the class primarily aim to write logging actions and print them to console. It uses a List<String> to store actions in the game. The GameActionsLoggerClass is separated from the main game functions for efficient debugging and code maintainability. However, extensive logging of actions in the game may impact the performance.

5. **GamePlayException Class**

The class represents functions for handling game-related errors and extending RuntimeException. It is used to effectively demonstrate issues that could happend during the game. It provides clear specific to the context error handling.

6. **GameTools Class**

The class represents tools that are necessary for a smooth gameplay. It provides methods such as random number generation and use of Input/Output files. The class centralizes all utility methods at one place. Nevertheless, Input/Output files have to handled asynchronously to avoid performance bottlenecks.

7. **Player Class**

The class represents a player in the game. It has List<Card> that demonstrates a hand of the player. The methods are used for specific actions that a player can perform in the game. It also utilizes GameActionsLogger for action tracking. The Player Class uses synchronized turn method to ensure safety in the multithread environment but could be a bottleneck with increased number of players.

Conclusion.

Overall, the design of the classes in the card game are well structured for best performance of the game, considering potential bottlenecks in the multithreaded environment. Each class meets specific needs of the game from the simple Card Class to more complex CardGame class. GameActionsLogger and GameTools classes demonstrate focus on the important aspects of the game such as logging and Input/Output files. Also, GamePlayException class shows error handling in the game.

Card Game Test Documentation

Overview

The card game presented is a Java based application that simulates the card game environment. Object-oriented design principles such as modularity, separation of concerns, etc. are used throughout the development process. This document focuses on the design decisions made during the development process, especially regarding testing using JUnit framework (5.x), highlighting specific examples, and expanding on error handling.

Design Choices

Testing Strategy Using JUnit 5.x

1. Framework Choice: JUnit 5.x

- Chosen for its powerful features and fluent API, improving readability and maintainability of tests. *Example: assertEquals(4, player.getCardCount());* .
- Enhanced support for parameterized tests and dynamic tests.

2. Test Classes Structure

- **CardDeckTest**: Tests CardDeck operations like drawing and discarding cards.
- **PlayerTest**: Focuses on testing player actions, wins, and card management.
- **CardGameTest**: Integrates testing of the overall game mechanics.

3. Test Case Design

- Each test case is designed to cover both typical use cases and edge cases.
- Assertive methods from JUnit 5.x (like assertEquals, assertTrue, assertThrows) are used to validate expected outcomes.
- Testing for exceptions (using assertThrows) to ensure proper error handling.
- Example: Testing card drawing when the deck is empty.

4. Mocking and Dependency Injection

- Although it is not widely used in current testing, the design makes it possible to easily integrate mocking frameworks such as Mockito for more private testing. For example, you can use objects like ‘Player’ without needing ‘CardDeck’ objects.

5. Continuous Integration

- The test suite is designed to be easily integrated into CI/CD pipelines for automated testing.

6. Documentation and Readability

- Test methods are well-documented, explaining the purpose and expected outcomes.

7. Performance Testing

- Although not currently implemented, the design allows for easy integration of performance tests, given the modular structure.

Conclusion

The card game is designed with object-oriented concepts, modularity and maintainability in mind. JUnit 5.X testing is an essential part of the game development process to ensure robustness and reliability. In the future, we can expect to see more use of mocking, performance testing and integration with a Graphical User Interface.