The Old Maid Card Game

Assignment 6

Name: Mohammad Al-Qaisy

Date: 6th Aug 2023

Table of Contents

Introduction

This assignment delves into the world of Java Multithreading to create a simulation of the Old Maid card game. In this captivating endeavor, each player is represented as a separate thread, interacting to play this card game in a digital realm.

The game's foundation rests upon a standard 52-card deck, supplemented by an additional wildcard, the Joker. Participants can range from a minimum of two players to a larger group. The game begins with an equitable distribution of the deck among the players, with slight disparities in card numbers tolerated. The primary objective of the game revolves around discarding cards strategically to eliminate the deck from their hands.

The strategic prowess required for victory revolves around recognizing and discarding "matching pairs" of cards. These pairs consist of cards that share not only the same value but also the same color. It's essential to remember that Spades (♠) must match with Clubs (♣), and Diamonds (♦) must find their counterparts in Hearts (♥). Interestingly, the Joker assumes a unique role, as it lacks a pair. Consequently, the player possessing the Joker card at the culmination of the game will bear the unfortunate distinction of losing the match.

The initial player selects a card from the second player, striving to create a matching pair that can be discarded. Following this, the second player advances the process by selecting a card from the third player's hand and subsequently discarding any pairs. This sequence continues as the third player acquires a card from the fourth player, the fourth player receives a card from the first player, and the cycle repeats. This circular progression persists until all participants, with the exception of the unfortunate holder of the Joker card, succeed in eliminating their cards.

The heart of this report explores the intricacies of employing Java Multithreading to craft a simulation of this captivating card game. By leveraging the power of concurrent execution, the simulation replicates the fluid interactions among players and cards, infusing the digital rendition with a sense of realism and excitement akin to the traditional analog experience. Through this endeavor, we gain insights into the intersection of game theory and multithreaded programming, delving into the dynamic interplay of chance and strategy that characterizes the Old Maid card game.

Object-Oriented Design

The implementation of the Old Maid Card Game simulation has been meticulously structured using Object-Oriented Design (OOD) principles. OOD offers a systematic approach to organizing code into modular and reusable components, enhancing clarity and maintainability. In this section, we will delve into the design aspects of key classes, their relationships, and the overall architecture of the simulation.

Card Class

The Card class forms the foundation of the simulation. It encapsulates the attributes that define a card - its Type, Color, and value. This class also provides methods to compare cards and determine if they form a matching pair. The Card class serves as the basis for creating the deck of cards that participants will interact with during the game.

SuitCards Class

The SuitCards class manages the deck of cards used in the game. It handles the creation of the deck, shuffling, and distributing cards among players. This class ensures fairness and randomness in distributing cards, essential to maintaining the game's integrity.

Player Class

Players are integral to the gameplay, and the Player class encapsulates their attributes and actions. Each player is associated with a unique player ID and maintains a hand of cards. This class provides methods to draw cards and discard matching pairs, facilitating player interactions with the game.

Game Class

The pivotal Game class orchestrates the entire gameplay experience. It manages the players and the deck, ensuring smooth progression through the game rounds. The Game class includes methods to initialize the game, execute rounds, check for

winners, and conclude the game. It encapsulates the game's core mechanics and serves as the central hub for interaction.

Play Class

Introducing a new dimension to the simulation, the Play class is designed to handle multi-threaded gameplay. It addresses the synchronization challenges associated with managing multiple players' turns. By meticulously orchestrating the sequence in which players take their turns and wait for others to complete their actions, the Play class creates a synchronized and harmonious gameplay experience. This class aligns with modern gaming standards and augments the simulation's interactivity.

Encapsulation and Modularity

The Object-Oriented Design of the simulation manifests in its exceptional encapsulation and modularity. Each class encapsulates specific functionalities, ensuring that responsibilities are segregated and promoting the maintenance of clean, focused code. This design choice contributes to the simplicity of testing, debugging, and future enhancement, as changes within one class are less likely to propagate throughout the system.

Incorporating Object-Oriented Design principles, the Old Maid Card Game simulation achieves a harmonious synthesis of structure and functionality. The careful delineation of classes, encapsulation of behaviors, and thoughtful management of interactions bring the classic card game to life in a digital landscape. As a testament to the elegance and effectiveness of Object-Oriented Design, the simulation showcases its adaptability, extensibility, and capacity to create a robust and immersive gaming experience.

Figure: Class diagram generated by IntelliJ.

Thread synchronization mechanisms

In the context of the Old Maid Card Game simulation, thread synchronization mechanisms are a pivotal aspect that ensures a smooth and orderly gameplay experience. The “Play” class, which implements the “Runnable” interface to represent player threads, employs thread synchronization techniques to orchestrate player turns and maintain a synchronized gaming environment. This section explores the thread synchronization mechanisms used within the simulation to achieve sequential player interactions.

Thread synchronization is crucial in multithreaded applications to prevent race conditions, data inconsistencies, and conflicts that may arise when multiple threads access shared resources concurrently. In the context of the Old Maid Card Game simulation, thread synchronization guarantees that players take their turns sequentially and do not interfere with each other's actions.

Mutex (Lock Object)

The primary synchronization mechanism utilized in the “Play” class is the Mutex (Mutual Exclusion) principle. This is achieved through the use of a “lock” object, instantiated as a static final object shared among all player threads. The “lock” object ensures that only one thread can access a critical section of code at a time, preventing concurrent access and potential conflicts.

Wait and Notify

Within the “run” method of the “Play” class, the “wait” and “notifyAll” methods are employed to manage the sequential execution of player turns. When a player's turn arrives, the thread associated with that player enters a synchronized block and waits for its turn. The “lock.wait()” call suspends the thread's execution until the condition specified in the “while” loop evaluates to “break”. Concurrently, the “lock.notifyAll()” method is used to wake up waiting threads and allow them to proceed once the current player's turn is completed.

Ensuring Sequential Player Turns

By utilizing the Mutex, “wait”, and “notifyAll” mechanisms, the “Play” class ensures that players take their turns in a sequential manner. The synchronization ensures that each player thread patiently waits for its turn and performs its actions within the synchronized block. This guarantees that players' interactions with the game, such as drawing cards and discarding pairs, occur in an organized and synchronized fashion.

Thread synchronization mechanisms are the backbone of the Old Maid Card Game simulation's multi-threaded gameplay. By employing Mutex, “wait”, and “notifyAll” techniques, the simulation achieves orderly player interactions and prevents conflicts arising from concurrent access to shared resources. This design choice not only enhances the reliability of the simulation but also contributes to an immersive and enjoyable gaming experience.

Clean Code Principles (Uncle Bob)

Throughout the development of the Old Maid Card Game simulation, I've integrated several key practices from "Effective Java" to enhance the codebase's quality, readability, and maintainability. Below are specific instances of these practices that have been incorporated into the game:

Item 2: Consider a builder when faced with many constructor parameters.

In my code, I do not need use more than one constructor

Item 6: Avoid creating unnecessary objects.

In my game, I've created objects only when they are needed and used. Every object in the game serves a purpose and isn't created unnecessarily.

Item 7: Eliminate obsolete object references.

Throughout the code, I ensured that I carefully manage object references and avoid holding onto objects that are no longer needed. This helps to prevent memory leaks and improves overall efficiency.

Item 8: Avoid finalizers and cleaners.

In adherence to best practices, I did not utilize any finalizers or cleaners in the code. These mechanisms are generally considered error-prone and are not necessary in the context of the game.

Item 10: Obey the general contract when overriding equals.

Given the need to compare Card objects, I overrode the equals method in the Card class to ensure it follows the general contract specified in Java. This ensures proper equality comparisons and consistency when comparing Card objects.

Item 11: Always override hashCode when you override equals.

In conjunction with overriding the equals method, I also appropriately overrode the hashCode method in the Card class and Player class. and make the equals method consider the game rule that is based on color and value.

Item 12: Always override toString.

In all classes within the game engine, I implemented the toString method to provide a meaningful and concise representation of the object's state. This aids in debugging and logging purposes.

Item 13: Override clone judiciously.

To avoid unnecessary complexity and potential issues, I consciously decided not to use or override the clone method in the codebase. This ensures simplicity and reduces the risk of unintended consequences related to cloning objects.

Item 26: Don't Use Raw Types

I've stayed away from using raw types in my code. Everything is well-typed and clear.

Item 28: Prefer Lists to Arrays

In managing collections, I've chosen ArrayList over arrays. This decision provides more flexibility and ease of use.

Item 34: Use Enums Instead of int Constants

I've used enums to represent card color and type. This ensures clarity and minimizes the chances of mistakes with integer constants.

Item 77: Don't Ignore Exceptions

Wherever exceptions might occur, I've incorporated proper exception handling. This ensures that errors are acknowledged and managed appropriately in the code.