Atypon Training Program 2023

JAVA MULTITHREADING AND OOP APPLICATION: COMPUTERIZED OLD MAID GAME

BY: YUSSIF ABDALLA, The University of Jordan

I. ABSTRACT

This report illustrates the solution of the Atypon's 2023 assignment. The assignment is based on the popular old maid card game, where each player is represented as a thread that plays the game. Java Multithreading and OOP is successfully used to implement the old maid game, which is played automatically by the computer (i.e. no human players are included).

The Java OOP Multithread program is designed such that each player is running as a separate thread in the main program. The main thread is used to start all player threads, divide the cards between them, and then report the game results at the end yet it is not used as one of the players. Finally, all player threads are put in the "wait" state when it is not their turn to increase program's efficiency. Special switching mechanism is used regulate the wait/notify mechanism between threads in order to simulate the game correctly while ensuring efficient use of the CPU.

II. INTRODUCTION

In the advent of computers with multiple CPUs, programmers may run concurrent processes and each one can handle a different task at the same time making optimal use of the available resources. These techniques are called Java multi-threaded programming language, which means we can develop multi-threaded programs using Java. A multi-threaded program contains two or more parts that can run concurrently [1].

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application [1][2].

A Java Multithreading and OOP to implementation of the old maid game that is played automatically by a computer, where no human players are included, is presented. The OOP multithreaded application is designed such that each player will be running as a separate thread in the program. Consequently, if there are N players, then there must be N threads in the game, plus the main thread. The main thread is designed to start all player threads, divide the cards between them, and then report the game results at the end. Finally, the main thread is not used as one of the players and the wait/notify mechanism is appropriately used the threads to simulate the game correctly while ensuring efficient use of the CPU. [3].

This report presents a detailed step-by-step problem-solving approach, outlining the different classes created and the relationships among them. In addition, the report summarizes the programming techniques that were implemented to achieve java multithreading and OOP application of a fully computerized old maid game.

III. THE OBJECT-ORIENTED DESIGN

Object-oriented design has been applied in the game development, assigning distinct responsibilities to individual classes, ensuring each class maintains a singular well-defined purpose. Refer to the figure 1 for an illustration of the game's structure:

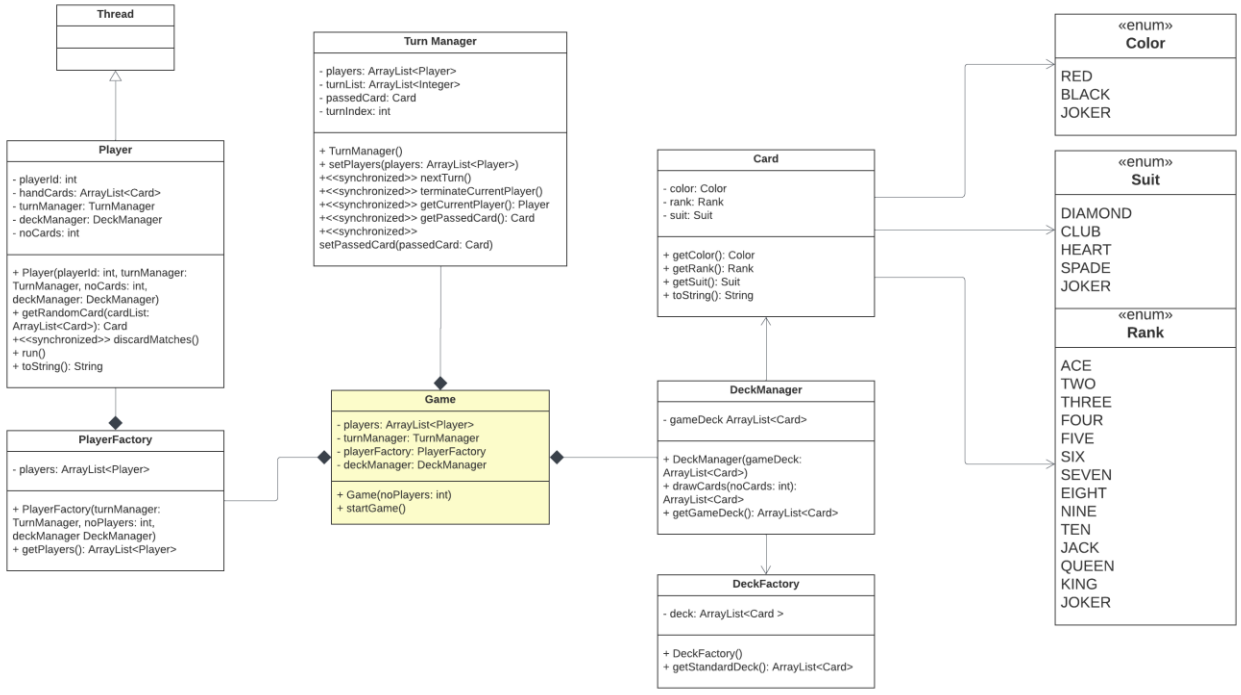

Figure 1: Game UML Structure

Here are the program's main classes:

The Game Class

The game class functions as the core of the game, integrating a turn manager, player factory, and deck manager, and then creating instances of these elements. When initializing the game class, the number of players is provided and passed to the player factory, which generates threads corresponding to each player within the game. Following the instantiation of the game class, the startGame() method is utilized to iterate through the players and initiate the threads.

Player Factory

The player factory class generates a list of players, assigning each player a unique ID and determining the number of cards they initially draw at the beginning of the game. It aims to distribute the cards equally among players; any remaining cards are automatically assigned to the last player. Subsequently, this class returns the list of players to the game class, allowing the game to start.

Player

The Player class extends the Thread class. Each player possesses a unique ID and maintains a hand of cards throughout the game. A turn manager facilitates the player's ability to conclude their turn. Additionally, the player interacts with a deck manager to draw cards at the game's outset. The number of cards drawn by a player at the beginning of the game is determined by the player factory during the player's creation.

Players have essential methods for the game, like drawing a random card to pass to the next player and discarding matching cards from their hands during their turns.

Turn Manager

The turn manager class oversees the game's progression, managing player's turns by maintaining a players list and an index that tracks each player's turn and the players remaining in the game. Players use the nextTurn() method to prompt the turn manager to adjust the turn index. When a player exhausts their hand, invoking the terminateCurrentPlayer() method removes the player's turn from the list and adjusts the index accordingly. The Turn Manager also facilitates obtaining the current user's turn.

Additionally, the turn manager features a passed card attribute, providing a location for players to place a card for the next player to pick up.

Deck Manager

The DeckManager receives a list of cards and offers players a drawCard() method for drawing cards during their initial turns. The game class can utilize DeckFactory.getStandardDeck() to obtain the original 52 cards along with an additional joker.

Deck Factory

The deck factory provides a getStandardDeck() method which games can use to obtain a standard deck of 52 cards, along with an extra joker. While other methods for creating customizable decks could have been included, I opted for simplicity.

The getStandardDeck() method populates the deck with cards that have predefined colors, ranks, and suits. Additionally, the cards are shuffled every time the method is invoked, ensuring that different games can experience unique shuffling patterns.

Card

The card class serves as a representation of the playing cards used within the game. It encapsulates essential attributes like color, rank, and suit, which are established using the Color, Rank, and Suit enumerations.

Colors are categorized as red and black.

Ranks encompass Ace, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, and King.

Suits consist of Hearts, Diamonds, Clubs, and Spades.

IV. FLOW CHART AND SYNCRONIZATION MECHANISMS

Synchronization mechanisms have played a vital role in the game, ensuring the smooth and orderly execution of threads while enabling players to seamlessly take their turns. A clearer understanding of these synchronization mechanisms can be achieved by referring to the flowchart depicted in Figure 2.

To maintain thread synchronization, threads progress through three distinct stages throughout the game. Here are the stages:

Stage 1: Drawing Cards

In this stage, threads determine their turn by using the getCurrentPlayer() method from the turn manager. If it's not their turn, they enter a waiting state by invoking the wait() method. If it is their turn, threads proceed to draw cards using the deck manager and call the nextTurn() method to modify the turn order. Afterward, the thread triggers notifyAll() to awaken other threads, prompting them to reevaluate their turn status.

This stage continues until all threads have completed it, and players have successfully drawn their cards.

Stage 2: Discarding Matches

Threads progress to this stage only after all of them have successfully completed stage one.

In this stage, threads make check for their turn by utilizing the getCurrentPlayer() method from the turn manager. If it's not their turn, they enter a waiting state by invoking the wait() method. On the other hand, if it is their turn, they proceed to discard matching cards from their hands, followed by invoking the nextTurn() method to modify the turn order. Subsequently, the thread employs notifyAll() to awaken all threads, prompting them to recheck their turn status. This stage endures until all threads have completed it, and players have successfully discarded matching cards from their hands.

After the players discard the matches, the thread checks if their hand is empty. If the hand is indeed empty, the thread will terminate using the terminateCurrentPlayer() method.

Stage 3: Gameplay

Threads move to this stage only after successfully completing the previous stages. This stage represents the heart of the gameplay, involving player decisions and interactions.

In this stage, threads identify their turn by using the getCurrentPlayer() method from the turn manager. When it's not their turn, they enter a waiting state using the wait() method. Conversely, when it is their turn, they proceed to receive a card passed from the previous player, followed by removing matching cards from their hand. Furthermore, they check if their hand is empty; if so, the thread ends by using the terminateCurrentPlayer() method.

If the player's hand still has cards, they should proceed by selecting a random card to pass to the next player using the setPassedCard() method within the turn manager class. Subsequently, the player should check if their hand is now empty. If it is indeed empty, their turn ends. However, if the player's hand consists of a single card, specifically the Joker card, and this player happens to be the last remaining participant in the game, an announcement is triggered, designating this player as the "old maid," followed by termination of their participation. Once these steps are concluded, the nextTurn() method comes into play to appropriately adjust the turn sequence. To synchronize the threads, the notifyAll() method is invoked, awakening all threads and prompting them to re-evaluate their respective turn statuses.

This stage continues until all threads terminate and the announcement of old maid is made.

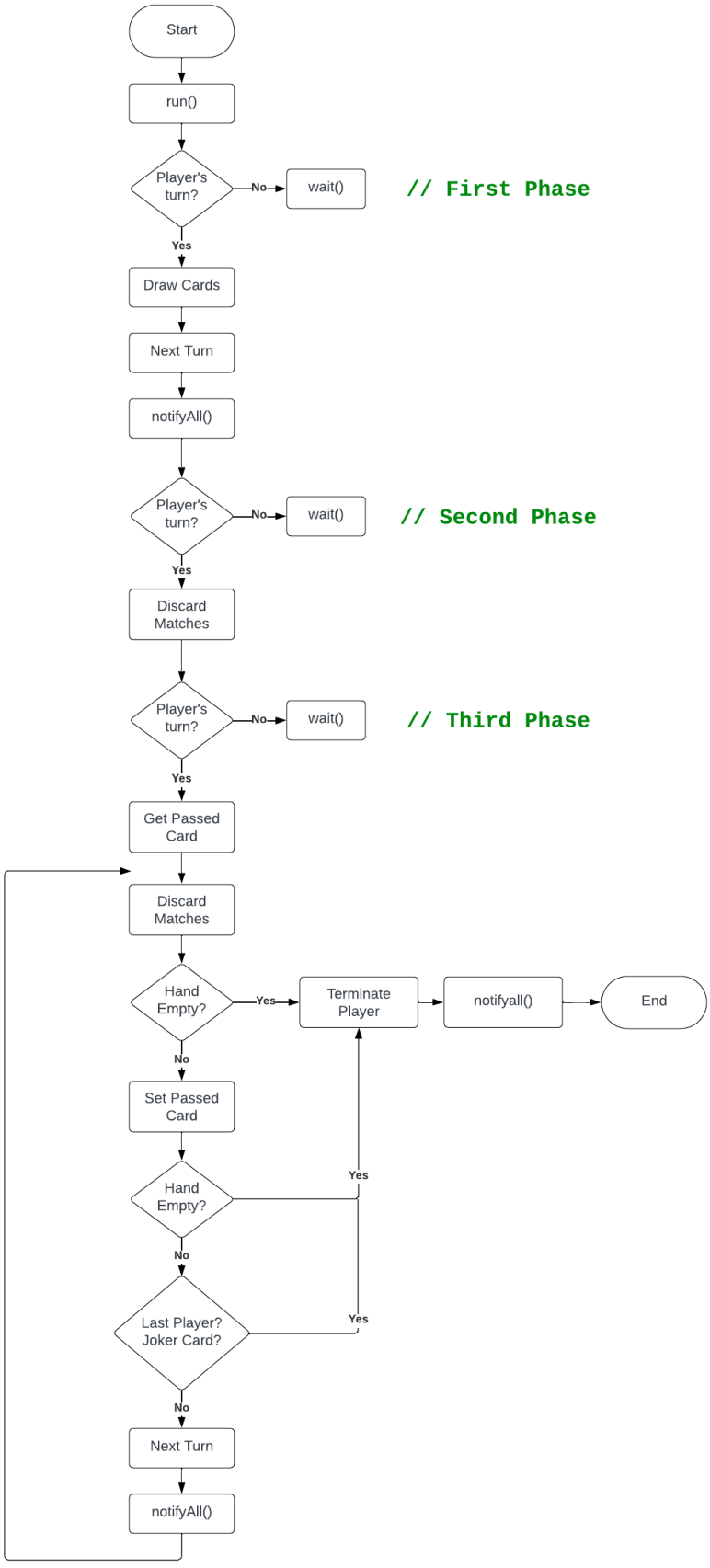

Figure 2: Synchronization Mechanism Flowchart

V. CLEAN CODE PRINCIPLES

Clean code principles make sure that you have a well-structured, readable, and maintainable code that is easy to understand and modify. It focuses on making the code more understandable, reducing complexity, and minimizing potential for errors and bugs.

The provided code adheres to the clean code principles outlined by best practices. Here are a few examples:

Naming

- Class names such as Game, DeckFactory, Player, TurnManager, and DeckManager are clear and indicative of their purpose.
- Variable and method names, like drawCards(), 'terminateCurrentPlayer()', and 'getPassedCard()', are descriptive and convey their intentions.

Functions/Methods

The code is divided into methods, improving readability and maintainability. Functions and methods have clear and focused purposes, such as ‘startGame()', responsible for starting the game, and 'terminateCurrentPlayer()', responsible for terminating the current player.

Comments

Well-placed comments effectively explain the purpose, behavior, and complex logic of certain components and algorithms. You can refer to Appendix A for examples of well-placed comments in the code.

Formatting

- The code follows consistent formatting conventions and maintains a clear and readable structure.
- Indentation and spacing are used consistently throughout the codebase, enhancing readability and maintainability.

Class Organization

- The classes in the codebase are organized and grouped based on their responsibilities.
- The Single Responsibility Principle and object-oriented design principles guide the class organization.

Error Handling

The code implements proper error handling mechanisms, such as try-catch blocks, and provides meaningful error messages and fallback mechanisms. You can refer to Appendix B to explore specific examples of how errors were handled.

VI.REFERENCES

[1] Java - Multithreading https://www.tutorialspoint.com /java_multithreading.htm / Aug.8, 2023

[2] Java On-Line: https://www.tutorialspoint.com/java_online_training/index.asp, Aug. 2023.

[3] The Old Maid Card Game, Dr. Fahed Jubair, Atypon Training Program 2023.

VII. APPENDICES

Appendix A

```
// All players get the same number of cards unless there's leftover then it goes to
the last player
for (int i = 0; i < noPlayers - 1 ; i++) {
        players.add(new Player(i + 1, turnManager, singlePlayerCards, deckManager));
}
         players.add(new Player(noPlayers, turnManager,
                 singlePlayerCards + leftOverCards, deckManager));
}
```

Appendix B

```
// Thread has been interrupted at sleeping or waiting
catch (InterruptedException e) {
        System.out.println("Thread was interrupted!");
        System.out.println(e.getMessage());
}


try{
        int noPlayers = scanner.nextInt();
        while(noPlayers > 12 || noPlayers < 2){
                System.out.println("Invalid number of players");
                noPlayers = scanner.nextInt();
        }

        Game game = new Game(noPlayers);
        game.startGame();
} catch (InputMismatchException e){
            System.out.println("Incorrect number format");
}
```