The Old Maid Card Game

1) Object-Oriented Desing

About my OOP design, I made 6 classes to create this game, I will walk through the classes in the best way to make the design easier for the reader.

The first class is the GameRunner class, a simple class that the end customers use to make a game and start it.

The Second class is the Game class, which is the main class here and consists of many players and a deck. When an object from this class is instantiated, a deck object will also be instantiated by calling the constructor of the Game class, this will lead me to talk about the third class in my design which is the Deck class and then back to the Game class.

The main responsibility of the Deck class is to create a list of cards and shuffle them so that the game class will distribute the card to the players.

The responsibility of the Game class is to start the game, distribute the cards to the players, make a thread for each player, start all threads and printing the name of the loser.

The fourth class is the Player class, each player will have a name and a list of cards. When the game class starting the threads, the run method will start working for all players and the player will start discarding the

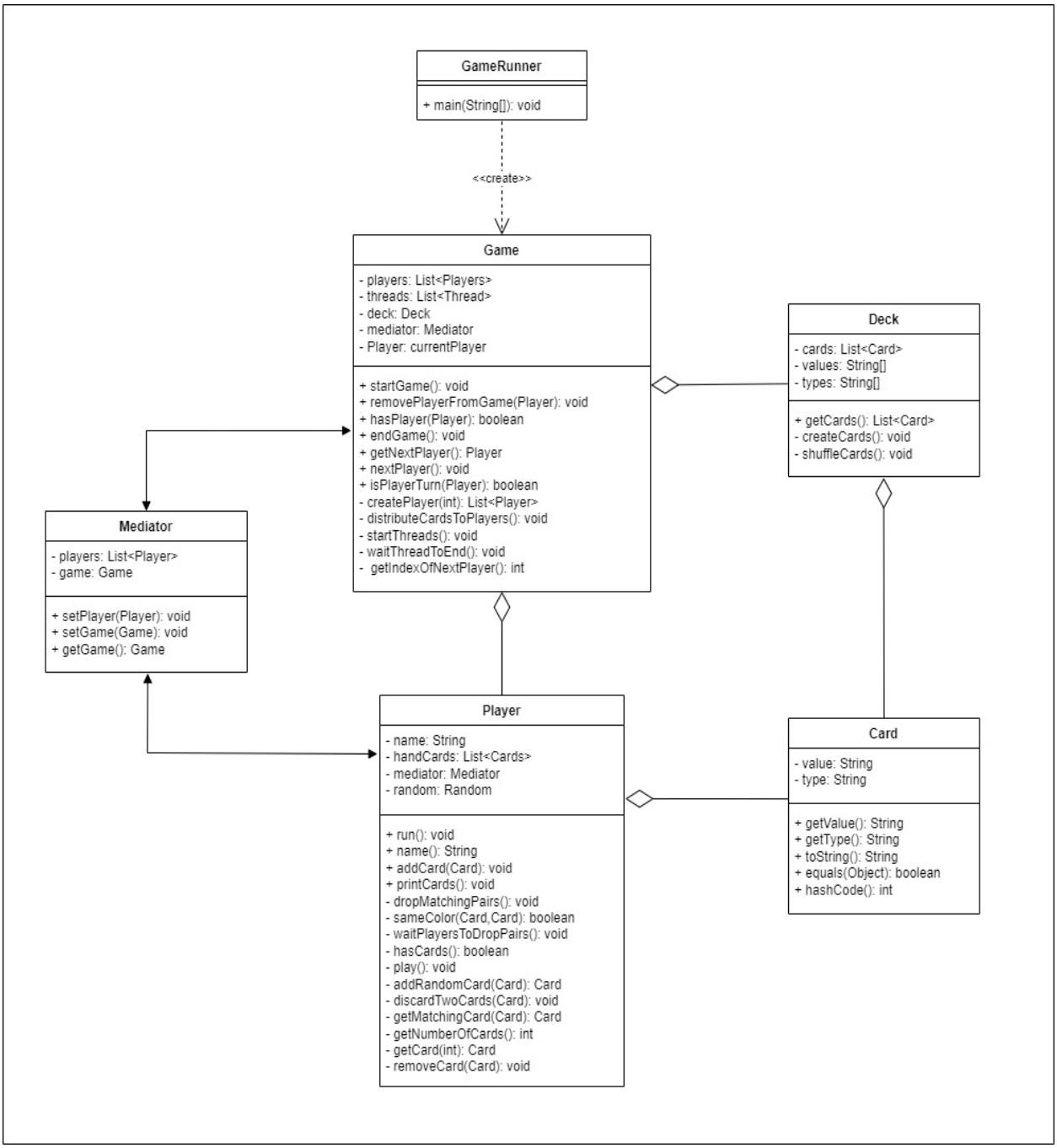
cards and drawing the cards from each other (I will mention more details in the second section of the report).

The fifth class is the Card class, a simple class that stores the value and type of each card and has some method for getting information and comparison.

The sixth class is the Mediator class, this class solve a real problem for me, because of the using synchronization, there will be a high coupling between the Player class and the Game class, as they need to communicate between each other, so in the scenario where I don't have the Mediator class, I will end up with circular dependency; this will make the design more complex, difficult to manage and it will affect the code readability and maintainability.

So, I came up with this class for managing the interaction between the Player class and the Game class.

"Mediator Pattern"



class diagram

2) Thread Synchronization Mechanisms

In the Game class, n threads will be instantiated to the players object and start, when all threads start, the run method in the Player class will start for all players concurrently.

```
1 usage
private void startThreads() {
    for(Player player:players) {
        Thread thread = new Thread(player);
        threads.add(thread);
        thread.start();
    }
}

1 usage
private void waitThreadsToEnd() {
    for (Thread thread:threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The first thing that all players will do is to discard the matching card from their hand concurrently, and then there will be a 500 ms sleep to make sure that all players are done with discarding their matching cards.

Then only the players that they have a cards in their hand will enter the loop and exit it only if he has no cards left in his hand, then only one player will enter the synchronized block and the other player will wait,

the player that entered the synchronized block will facing a loop that check if it is his turn or not, if not he will enter the loop and wait for the notify, and the synchronized block will be open and another player enter the blocking and so on.

When the player that it is his turn and enters the synchronized block, he will not be facing the loop because it is his turn, he will be facing if statement for handling the scenario where is the game is already end.

```
@Override
public void run() {
    dropMatchingPairs();
    waitPlayersToDropPairs();
    while(hasCards()) {
        synchronized (mediator.getGame()) {
            while(!mediator.getGame().isPlayerTurn(this)) {
                try {
                    mediator.getGame().wait();
                    if(!mediator.getGame().hasPlayer(this)) {
                        return;
                    }
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            if(mediator.getGame().endGame()) {
                break;
            }
            play();
            if(!hasCards()) {
                mediator.getGame().removePlayerFromGame(this);
                return;
            }
        }
    }
}
```

So, let's assume that the game is still not ending, the player will call the play method, which does the following, it will get the next player and get a random card from him and then check if there is any matching pair if there the player will throw them.

After this work is done, the current player will be the next player, and the player will notify all players that are waiting in the synchronized that he is done.

The notify method will be calling from the nextPlayer method that is in Game class.

```
1 usage
private void play() {
    Player player = mediator.getGame().getNextPlayer();
    System.out.println(name + " draw card from " + player.name);
    Card card = addRandomCard(player);
    discardTwoCards(card);
    mediator.getGame().nextPlayer();
}
```

```
1 usage
public void nextPlayer() {
    int index = getIndexOfNextPlayer();
    currentPlayer = players.get(index);
    notifyAll();
}
```

The last thing in the run method is to check if the current player does not have any card that's mean he is not the loser, and we will delete this player from the game and print that he is out and end his thread.

This loop will continue if there is more than one player in the game. When there is only one player in game, the loop will end and there will be only one player that is not deleted from the game, and this player is the looser.

Important this to mention:

In the startGame method from the Game class, after starting the threads, there will be a method that makes any thread end earlier wait all threads to end and then will print the looser of the game.

3) Clean Code Principles

SOLID Principles:

1. Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one responsibility or job. As we see in the design every class has a single responsibility. (Card, Deck, Player…).

2. Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. Because of the simplicity of this game, this principle will not be used.

3. Liskov Substitution Principle (LSP): Objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program. In this solution we don’t have inheritance.

4. Interface Segregation Principle (ISP): A class should not be forced to implement interfaces it does not use, and clients should not be forced to depend on interfaces they do not use.

5. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

DRY (Don't Repeat Yourself):

This principle emphasizes that duplication in code (such as repeating the same logic) should be avoided. Instead, a single, authoritative source should exist for a particular piece of knowledge. In the Old Maid Card Game there is no code duplication, as every step that used more than one time is put in method and called by method name.

YAGNI (You Ain't Gonna Need It):

This principle suggests that you should not add functionality until it is necessary. Avoid speculative or premature features that are not currently required. In this code all functionalities written are used.

The Boy Scout Rule:

Leave the code cleaner than you found it. Whenever you make changes to the code, take the opportunity to improve it by refactoring or cleaning up any mess you encounter. As I am making this solution, I follow up this role, and do it more than once.

Composition Over Inheritance:

Favoring composition (building complex objects by combining simpler ones) over inheritance can lead to more flexible and maintainable code. I gave priority to composition over inheritance in my design, as this will make it simpler.