# Mahjong

Yubo Sha 22207224
Yihan Yan 22207211
Tianrui Liu 2220721
Peiqi Tian 22207209

June 2024

# https://github.com/YolandaHarp/MahJong

| Name | GitHub name |
|---|---|
| Yubo Sha | SYB1103 |
| Yihan Yan | YolandaHarp |
| Tianrui Liu | lvtiry |
| Peiqi Tian | tianpeiq |

# Contents

# 1 Introduction

Mahjong is a four-person game that originated in China, it is an entertainment that is generally made of bamboo bone or plastic made of small rectangular pieces, engraved with patterns or words. It contains 136 tiles. The way mahjong is played varies depending on the region and cultural background.

Tile type: Mahjong tiles are divided into Dot, Bamboo, and Character suits, each suit has 1 to 9 nine numbers, and each tile has four. In addition, there are some special Honor tiles, such as east, south, west, and north tiles.

The goal of the game: The player's goal is to form the tiles in his hand into specific tile types, such as pairs (two identical tiles), triplets (three identical tiles), barbs (four identical tiles), and sequences (three tiles in a row).

Winning rules: When playing, players need to have a pair of tiles (pairs) and three sets of sequences or triplets. However, there are other special ways to win—for example, 7 couples of identical tiles and other traditional ones which may require certain lucky values.

Special rules: Chow, Pong, Kong. The last house plays tiles, and the next house tiles just form a pair of sequences, he can Chow. Someone else plays a tile, and if he has two identical tiles in his hand that make up exactly one set, he can Pong them. Bars are divided into exposed Kong and concealed Kong, other people play a tile, and their hand has three identical tiles, called exposed Kong. There are three identical tiles in the hand, and caught an identical tile, called the concealed Kong, others do not know what tile. When you have all the tiles in your hand together into a useful tile, just add the last one can be and tile, and you can enter the stage of ready hand. Win by others means that you play a tile that just lets the other side have a tile, which means that you let the other side win.

## 1.1 Version

The version of this project is the **Beijing Version** it has some different rules than other versions.

### 1.1.1 Tiles different:

In the Beijing Version, the tiles named "Bonus" do not exist. The amount of tiles in one game is 136.

#### 1.1.2 Joker

In each game, when all four players have drawn their cards the first card in the will be face upward as a sign to show the Joker which is the next order after the sign. For example, if the sign tile is bamboo 2 then the joker will be bamboo 3, by parity of reasoning, the "Honors" have an order as "East Wind", "South Wind", "West Wind", "North Wind", "Red Dragon", "Green Dragon", "Write Dragon". Joker means the type that in the next order after the sign, is not a tile but 4 tiles. It can be used to represent all types of tiles. However, it cannot be used in Kong, neither exposed Kong nor concealed Kong.

#### 1.1.3 Rules different

The rules to win a game have some differences a person can wait for the one necessary tile to win if and only if the player didn't do any actions like "Chow", "Pong" and "Kong" which means the player keeps 13 tiles in hand but not show some of the tiles facing up to other players. If a player has actions then this player can only win by drawing a tile by himself to win. Actions do not have limits which means players can do as many actions as they want.

# 2 Game Requirements

## 2.1 User requirements:

- 1. Allow users to check the rules of the game.

**System requirements:**

1.1 Contain one option in the main menu to show the rule.

- 2. Allow the user to start the game.

**System requirements:**

2.1 Contain one option in the main menu to start the game.

- 3. Allow the user to end the game.

**System requirements:**

3.1 Contain one option to end the game during the game.

3.2 The system can determine whether the game is over after one player finishes his round.

3.3 The system can determine whether the game is over after one player wins.

- 4. Display the pattern of each tile in the player's hand and the discard pile.

**System requirements:**

4.1 The system should connect which pattern represents which type of tiles.

4.2 The system should store where the tile is (In the wall, in the player's hand or the discard pile).

4.3 The system should show the tiles as the pattern in the player's hand, discard pile and some special places.

- 5. Allow the user to get one tile at the beginning of his round.

**System requirements:**

5.1 The system should have a queue to store all tiles in the game (represents the wall).

5.2 The system should connect which pattern represents which type of tiles.

5.3 The player should have a place to store 14 tiles (player's hand).

5.4 The system should remember which player can have action.

5.5 The system should give the head of the queue to the player.

5.6 The player can add the tiles into the player's hand.

5.7 The system should store where the tile is (In the wall, in the player's hand or in the discard pile).

- 6. Allow the user to throw one tile into the discard pile to end his round.

**System requirements:**

6.1 The system should connect which pattern represents which type of tiles.

6.2 The player should have a place to store 14 tiles (player's hand).

6.3 The system should remember which player can have action.

6.4 The player can throw the tiles from the player's hand.

6.5 The system should allow other players to respond to the tiles this player throws.

6.6 The system can store the tiles in the discard pile.

6.7 The system should store where the tile is (In the wall, in the player's hand or in the discard pile).

- 7. Allow user Pong, Kong, Chow, Exposed or Win after another player throws a tile.

**System requirements:**

7.1 The system should connect which pattern represents which type of tiles.

7.2 The system should remember which player can have action.

7.3 The system should allow other players to respond to the tiles this player throws.

7.4 If legal, the system should allow the player to get the tiles and do further action (Pong, Kong, Chow, Exposed or Win).

7.5 The system can move the tiles this player throws to the other player's hand.

7.6 System should store where the tile is (In wall, player's hand or discard pile).

- 8. Allow the user to see how many tiles are left in the queue.

**System requirements:**

8.1 The system should have a queue to store all tiles in the game (represents the wall).

8.2 The system should store where the tile is (In the wall, in the player's hand or in the discard pile).

8.3 The system should remember how many tiles are left in the queue (the wall).

- 9. Allow the user to see which type of tile is Hun (which can represent every type of tile).

**System requirement:**

9.1 The system should connect which pattern represents which type of tiles.

9.2 The system should store which type of tile is Hun.

9.3 The system should show which type of tile is Hun.

- 10. Allow 4 users to join in the game on 4 different computers.

**System requirements:**

10.1 The system should contain a protocol for multi-computer interaction.

10.2 The system should allow data transfer from one player to another.

Data: such as tiles from the queue, tiles throw, whose turn ...

10.3 One computer serves to work as the host another computer serves to access the host to get data.

10.4 Every player should see the same discard pile and tiles in their hands.

## 2.2 Functional Requirements:

1. The user shall be able to do an action (throw tiles, Pong, Kong, Chow, Exposed or Win) by clicking the mouse.

2. The system should display a game scream where the player playing the game.

3. The system should respond to user requests within an acceptable time.

4. The system should make sure all players have data synchronization.

5. The system should be robust.

6. Let 4 players join the game online.

7. The player can quit the game whenever they want and they can return to the main menu and start a new game.

8. When someone is unconnected to the game, all four players who play together will be logged out.

## 2.3 Non-Functional Requirements:

1. Usability requirements:

User interface design should be simple.

Users can do the action easily.

2. Performance requirements:

The system should response time should be as fast as it can.

The system should maintain high resource utilization.

3. Reliability requirements:

The system should run according to plan no matter what the player clicks on the game scream.

4. Scalability requirements:

The system should allow 4 users to access the system.

5. Compatibility requirements:

Ensure the software can run properly on different computers.

# 3 Structure of Project

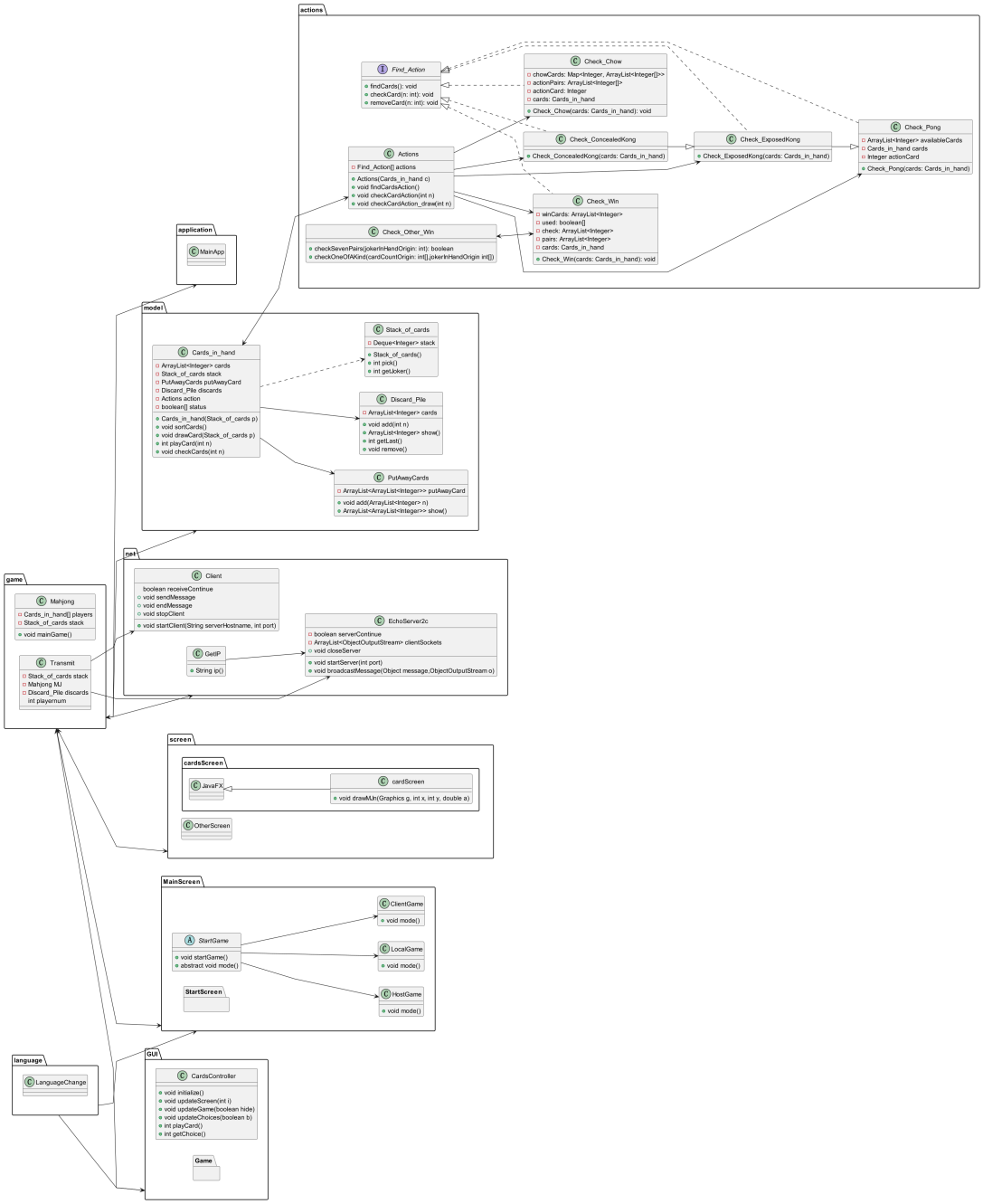



Figure 1: Class diagram

## 3.1 Structure

The file structure of this project. There are several packages: application, game, GUI, language, Main Screen, model, net, screen, and UMLs.

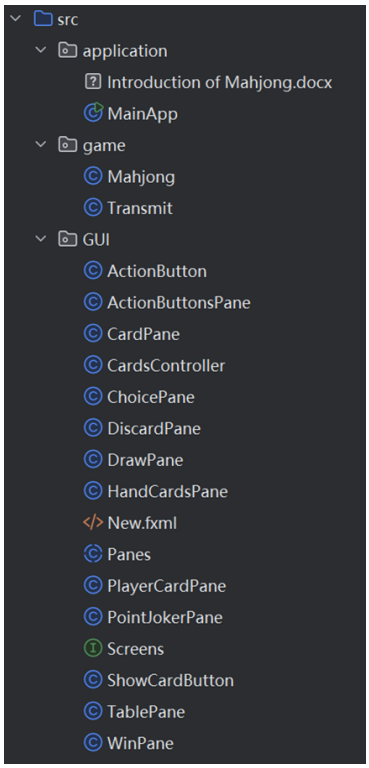



Figure 2: application, game, and GUI package

The application package contains the introduction to the game, and the MainApp class is used to enter the game. The Game pack contains a Mahjong class for the game's theme logic and a transmit class for sharing data with all players. The GUI package mainly contains the code for the interface and the buttons or patterns on the game interface.

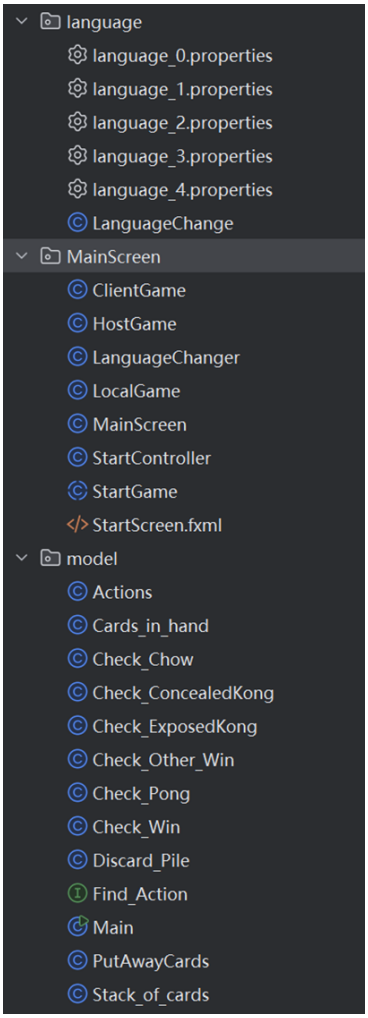



Figure 3: Language, MainScreen and model package

The language pack contains the game's four language modes, which can be switched between each other. The MainScreen package contains code to draw the game's main interface and the contents of the main interface. The model package mainly contains several operations that can judge the cards such as chow and pong, and judging concealed Kong and exposed Kong as well as joker and win.

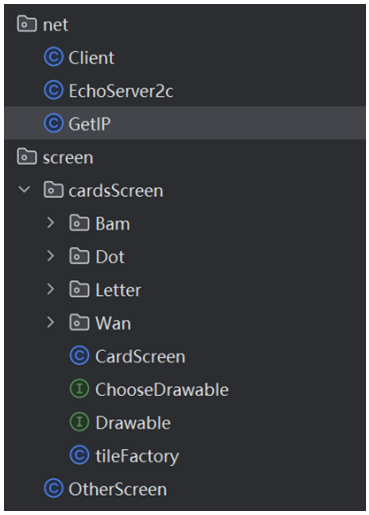



Figure 4: net and screen package

The net package contains code related to network connection, it contains three classes for the client service, and the host service, to get the IP of the current computer. Everything in our game is drawn in Java, no images are used. The drawing code for the pattern of the mahjong deck and the pattern of the interface button are placed in the screen package.

## 3.2 Front-End Code Implementation Method

The main structure of the game interface is constructed in game.fxml and managed through the CardsController class, which also provides external interfaces. The main components include:

ActionButtonsPane: A panel that manages multiple instances of ActionButton. It is used for displaying and managing the player's action selection buttons.

ChoicePane: A panel that manages the player's choice of actions. It displays the options available to the player for their actions.

DiscardPane: A panel that displays and manages the player's discarded cards. It handles discard operations and displays the discard area.

HandCardsPane: A panel that displays the player's hand of cards. It manages the display and interaction of the hand cards.

TablePane: A panel that displays and manages the playing table, including all the cards visible to each player.

WinPane: A panel that displays the player's winning card information. It handles the display and operations after winning a hand.

The main structure of the main interface is constructed in startScreen.fxml and managed by StartController. It primarily enhances the appearance of the interface and initializes and sets up various buttons on the main screen, including initializeGameButton(int n), initializeExitButton(), and initializeInfoButton().

# 4 Junit Test

Application of Junit4 testing framework to Mahjong

In this project, our team attaches great importance to Junit unit testing. It can be used to ensure code quality and stability. Junit4 provides an easy way for the team to run unit tests and find out the bugs in the code.

Because of program encapsulation, many important data is private. So, the test is stored in the action package. Our tests aim to test the importance of logic in the game. We test the logical judgment of Chow, Pong, Kong, and Win.

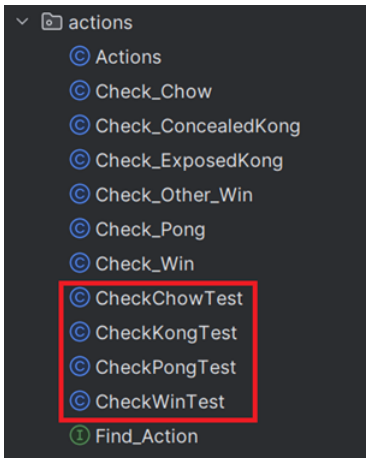



Figure 5: logic tests

Also, we test the net to make sure the client can connect to the host successfully.

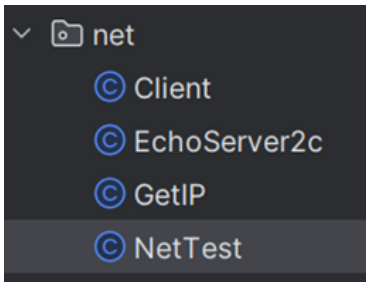



Figure 6: Net Test

In the test code, we use the normative Junit4 language and comments are used to ensure readability:

'@Before': A method used to execute before each test method.

'@Test': Used to mark a test method.

The test code makes use of Assert methods to evaluate the execution of the code under different conditions, such as:

assertEquals(expected, actual): Checks whether two values are equal.

assertTrue(condition): This verifies whether the condition is true.

assertFalse(condition) - Checks if the condition is false.

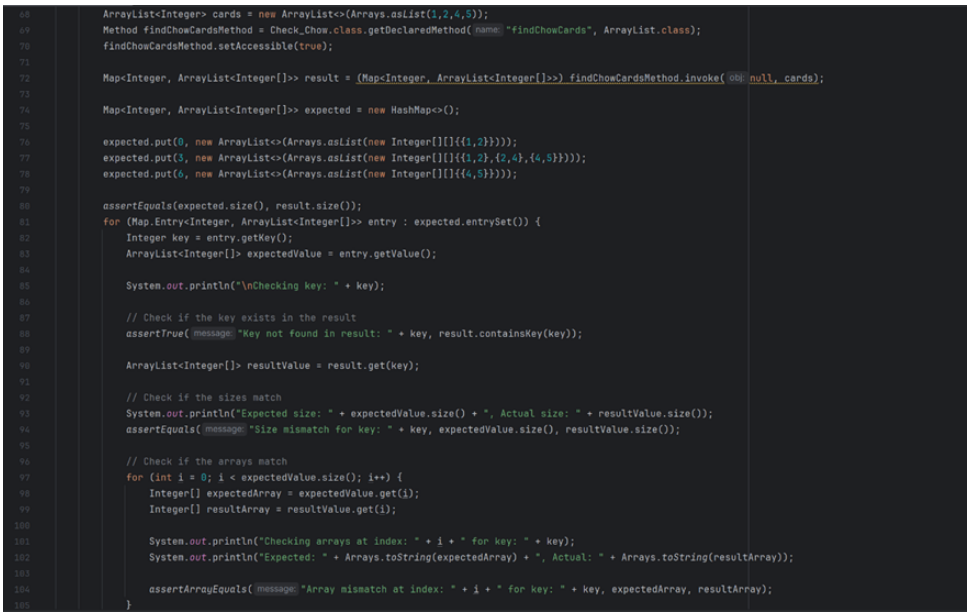
assertNull(object): This validates whether an object is null.

assertNotNull(object): Verifies if an object is not null.

assertSame(expected, actual) - Verifies if two objects are the same reference.

assertNotSame(expected, actual) - Verifies if two objects are not the same reference.

In our test, we use assertEquals() to find whether the result is the same as the expectation. Use assertTrue() and assertFalse() to test that the Boolean type data is changed correctly. We try our best to include all possible situations in the test.



```
ArrayList<Integer> cards = new ArrayList<>(Arrays.asList(1,2,4,5));
Method findChowCardsMethod = Check_Chow.class.getDeclaredMethod( name: "findChowCards", ArrayList.class);
findChowCardsMethod.setAccessible(true);

Map<Integer, ArrayList<Integer[]>> result = (Map<Integer, ArrayList<Integer[]>>) findChowCardsMethod.invoke( obj: null, cards);

Map<Integer, ArrayList<Integer[]>> expected = new HashMap<>();

expected.put(0, new ArrayList<>(Arrays.asList(new Integer[]{{1,2}})));
expected.put(3, new ArrayList<>(Arrays.asList(new Integer[][]{{1,2},{2,4},{4,5}})));
expected.put(6, new ArrayList<>(Arrays.asList(new Integer[]{{4,5}})));

assertEquals(expected.size(), result.size());
for (Map.Entry<Integer, ArrayList<Integer[]>> entry : expected.entrySet()) {
    Integer key = entry.getKey();
    ArrayList<Integer[]> expectedValue = entry.getValue();

    System.out.println("\nChecking key: " + key);

    // Check if the key exists in the result
    assertTrue( message: "Key not found in result: " + key, result.containsKey(key));

    ArrayList<Integer[]> resultValue = result.get(key);

    // Check if the sizes match
    System.out.println("Expected size: " + expectedValue.size() + ", Actual size: " + resultValue.size());
    assertEquals( message: "Size mismatch for key: " + key, expectedValue.size(), resultValue.size());

    // Check if the arrays match
    for (int i = 0; i < expectedValue.size(); i++) {
        Integer[] expectedArray = expectedValue.get(i);
        Integer[] resultArray = resultValue.get(i);

        System.out.println("Checking arrays at index: " + i + " for key: " + key);
        System.out.println("Expected: " + Arrays.toString(expectedArray) + ", Actual: " + Arrays.toString(resultArray));

        assertArrayEquals( message: "Array mismatch at index: " + i + " for key: " + key, expectedArray, resultArray);
    }
```

Figure 7: Test Example

The code in the picture is one test Chow situation. In this situation, the player has 4 cards in hand without any joker (34). There are three situations when Chow. It can Chow if the previous player plays 0, 3 or 6.

# 5 Design Pattern

## 5.1 Singleton pattern

Singleton pattern is used in many parts of the project, such as model, net, and GUI. Most of the categories in these three packages use this design pattern. It ensures that there is only one instance of these classes and it provides a global access point. In a multithreaded environment, it ensures that there is only one instance globally that can avoid problems caused by the inconsistent state of multiple instances, and it can ensure the uniqueness and thread safety of instances in a multithreaded environment. A single instance can be shared by all threads, which greatly reduces resource consumption and the overhead of repeated initialization. In addition, this pattern enables centralized configuration and management of services and components shared across modules. Meanwhile, global access points also simplify the structure of the code, making the process of obtaining instances more intuitive.

## 5.2 Composite pattern, Facade pattern and Proxy pattern

We use a combination of Composite pattern, Facade pattern and Proxy pattern in the mahjong behaviour checking section and the Javafx front-end controller section. Taking checking the behaviour of mahjong as an example, the Actions class combines five Find_Action classes: Check_Chow, Check_ConcealedKong, Check_ExposedKong, Check_Pong and Check_Win, making the main program of the game to use these objects with consistency. At the same time, the Actions class wraps the behaviours in each Check class, providing a simple and unified interface for the main program of the game. The main program can check all operations in the same method, such as findCardsAction() and checkCardAction(). This not only simplifies the interaction between the system but also reduces the dependency between the customer and the subsystem. Moreover, this greatly improves the maintainability and scalability of the system. In addition, in the special case where a player has multiple options to choose from, such as eating, the main program can retrieve all options directly from Check_Chow via the getChowChoice() method in Actions, where the Action class acts as an intermediary. This gives the Actions class the ability to intelligently handle requests for special cases and the flexibility to add additional functionality or controls.

## 5.3 Observer Pattern

We use the observer pattern for the game's networking logic and front-to-back interactions. sendMessage () and receiveMessage () in the Client class will notify all other players to update the data after each player performs any operation, the front-end game interface is updated as well using the upDate () class, which ensures that every player has exactly the same data. It also ensures that the front-end display picture is the same as the real-time data at the back end and it ensures the low coupling and high collaboration between objects.

## 5.4 Memento pattern

We use the memo pattern in the card-checking part of the game, the Check_Win class. In each check class, it will first save the original state of the hand, then make changes to the opponent cards, and finally check whether the hand has been all removed and reached the win condition, if not, it will restore the hand to the initial state and then carry out the next possibility check. CardCountOrigin and jokerInHandOrigin are used to save the state and restore it. This allows the program to easily reset the state without having to do multiple retrieves.

## 5.5 Factory pattern

The factory pattern provides a way to create objects. It eliminates the need to specify the specific class to be created, separating the process of creating objects from the process of using them. We used factory mode for the painting of various cards in the game. In the tileFactory class, other modules only need to enter the corresponding parameters and it can output the corresponding deck. It encapsulates the object creation logic in a factory class, rather than instantiating the object directly in the client code, which improves the maintainability and extensibility of the code.

## 5.6 Transfer object pattern

We use the transfer object pattern in the Transmit class, which packages all the data of the game at one time and sends it to the server. After receiving the data, the client can use a series of get methods to obtain the data and update it. It reduces the number of network requests by transmitting multiple data at one time. At the same time, the transmission object hides the internal data structure as the

encapsulation carrier of data.

### 5.7 Decorator pattern

Decorator pattern allows you to add new functionality to an existing object without changing its structure. We use the Decorator pattern in MainScreen. Firstly, we create an abstract class, "startGame" which initializes the Javafx interface of the game and has an abstract method called mode, whose purpose is to launch the game in different modes. Three classes extend this class, LocalGame, HostGame, and ClientGame, which implement three modes for mahjong games. Moreover, the decorative class and the decorated class can change independently and do not affect each other. Not only that, it also reduces code duplication and improves code reusability.

## 6 Data Structure

### 6.1 Double-ended queue

In the draw pile section, a double-ended queue is used because the draw pile and the discard pile are similar in that only the card at the front is of concern, and it is removed when drawing a card. However, the draw pile differs in that, during initialization, it requires the removal of a joker indicator card and four corresponding joker suits. In the code, joker cards are treated as a separate type and are assigned a specific display style during initialization.

### 6.2 Stack

In the discard pile section, a stack is used because the most recently discarded card is of interest and may be removed (for chows or pongs by other players). For outputting the discard pile, an Iterator is used, as there is no interaction with the middle of the list. Using an Iterator to draw the discard pile on the front end sequentially is sufficient.

### 6.3 Array

In sections such as the player list and the action record list, fixed-length array data structures are used. Since their length is fixed, they can avoid certain issues and generally run faster than ArrayLists.

Excessive creation of ArrayLists could potentially slow down the performance.

## 6.4 Map

A data structure consisting of a map with integers as keys and ArrayLists of integer arrays as values is utilized. This structure records multiple chow possibilities for a player. For instance, if there are tiles 1, 2, 4, and 5, and a 3 is discarded, there are two chow possibilities: 123 and 345. This data structure effectively accommodates this characteristic.

After a move is made, the map is regenerated. Using the previous example, one of the keys in this map would be 3, indicating that the tiles can interact with the discarded 3. The ArrayList in the value stores two pairs of integers, 1,2 and 4,5, representing the possible actions. This structure also facilitates the front-end display of these choices, allowing the creation of an HBox for each integer array in the FXML then it can be seen by the player.

## 6.5 Array List

In the remaining parts of the code, such as recording players' hands and checking for Chows, Pongs, and Kongs, many ArrayLists are used. This data structure is versatile, providing numerous methods for the necessary searches and checks.

# 7 Refactoring

## 7.1 Improved Code Readability and Maintainability

```
public void startGame(){
    initializeGame();
    boolean win=false;
    while(!win){
        if(Stack_of_cards.getStack().remainCardNum()==0){
            Platform.runLater(() -> {
                FinishPane.getFinish().updateNoCards();
            });
            break;
        }
        boolean waitToDraw = false;
        if(findNextPlayer()){
            updateGame();
            Platform.runLater(() -> {
                ActionButtonsPane.getButton().updateCanvases();
            });
            int result=ActionButtonsPane.getButton().getChoice();
            if(result==4){
                playerList[nowPlayer].clearStatus();
                continue;
            }else if(result==0){
                win=true;
                Platform.runLater(() -> {
                    FinishPane.getFinish().updateCanvases();
                });
            }else if(result==1||result==2){
                playerList[nowPlayer].getAction().removeAction(result, n: 0);
                if(result==1){
                    waitToDraw=true;
                    Stack_of_cards.getStack().setEnd();
                }
            }else if(result==3){
                int chowResult=0;
                if(playerList[nowPlayer].getAction().getChowChoice().size()!=1){
                    Platform.runLater(() -> {
                                ChoicePane.getChoicePane().updateChowChoices(playerList[nowPlayer].getAction().getChowChoice());
                            });
                    chowResult=ChoicePane.getChoicePane().getChoice();
                }
                playerList[nowPlayer].getAction().removeAction(result,chowResult);
```

Figure 8: Before Refactoring

```
public void startLocalGame() throws IOException {
    initializeGame( online: false);
    while(!win&&!getStop()){//Quit the game when game is running
        if(Stack_of_cards.getStack().remainCardNum()==0){
            updateScreen( i: 6);
            break;
        }
        boolean haveAction=findNextPlayer( online: false);
        boolean canPlay=turnStart(haveAction, online: false);
        drawStage( online: false);
        if(win){
            updateScreen( i: 5);
            break;
        }
        if(canPlay) {
            playStage();
        }
    }
}
```

```
private void drawStage(boolean online) throws IOException {
    //The player's actions after drawing a card
    while(playerList[nowPlayer].getStatus( n: 0)||playerList[nowPlayer].getStatus( n: 4)) {
        updateScreen( i: 1);
        int result = getChoice();
        if (result == 4) {//player cancel
            playerList[nowPlayer].clearStatus();
            break;
        }else if (result == 0) {//player  win
            win = true;
            break;
        }else if(result==1){//player kong and draw card again
            int cKongResult=0;
            if(playerList[nowPlayer].getAction().getCKongChoice().size()!=1){
                cKongResult=updateChoices( b: false);
            }
            playerList[nowPlayer].getAction().removeAction( action: 4,cKongResult);
            playerList[nowPlayer].drawCard();
            Stack_of_cards.getStack().changeEnd();
            updateGame( hide: false);
            if(online){
                Client.getClient().sendMessage();
            }
        }
    }
}
```

Figure 9: some method after Refactoring

This project has been refactored that jumbled methods have been divided into small methods. For example, the methods in the mahjong class were very long which contained the logic of the game. It has been refactored to be some small method.

### 7.1.1 Readability

Smaller methods are generally easier to understand. Each small method handles a specific task, making the code's intention clearer.

#### 7.1.2 Maintainability

: Smaller methods are easier to test and debug, reducing the likelihood of errors. When issues do arise, they are easier to locate and fix.

#### 7.1.3 Reduction of Duplicate Code

By extracting repeated code fragments into separate methods, you can avoid code duplication, improving code maintainability and consistency. Simplified Complexity:

### 7.2 Enhanced Testability

Smaller methods are easier to unit test because each method deals with a specific task, allowing test cases to be more focused and precise. This improves code reliability. Promotes Code Reuse:

By extracting commonly used functionality into independent methods, these methods can be reused throughout the project, reducing code duplication and increasing development efficiency.

### 7.3 Increased Flexibility

Smaller methods make the code easier to modify and extend. When new features need to be added or existing features modified, changes can be made more easily without affecting other parts of the code.

### 7.4 Improved Design

Refactoring makes the code structure clearer, helping to optimize the design of the code and improve the overall quality of the system

## 8 Team Member Contribution

### 8.1 Division of Team Member

- Tianrui Liu: Participate in testing, implement backend logic for chi, peng, gang, and hu, and write the report.
- Peiqi Tian: Design card faces, implement frontend layout and components, and write the report.
- Yubo Sha: Main game program, network setup, connect frontend and backend, write the report.

- Yihan Yan: Participate in testing, implement multilingual support, assist in frontend component implementation, and write the report.