

Faculty of Applied Sciences  
Bachelor of Science in Computing

**COMP221 Group Project  
Report**Academic Year 2021/22 Semester 2

|  |  |
| --- | --- |
| Online Turn-based Game | |
|  |  |
| Group Number: | 01 |
| Student ID: | P2010411  P2010556  P2010562 |
|  |
| Student Name: | Jing Yu Long, Grant  Liu Run Rong, Polo  Chen Ling Xiao, Veronica |
|  |
|  |
|  |
| Submission Date: | April 26, 2022 |

**Table of Content**

[1 Introduction 2](#_Toc101863537)

[1.1 Application Information 2](#_Toc101863538)

[1.2 Requirement Analysis 2](#_Toc101863539)

[1.3 Software Architecture 3](#_Toc101863540)

[1.4 How to Use 4](#_Toc101863541)

[1.5 Details / Demonstration 5](#_Toc101863542)

[2 Object Model 8](#_Toc101863543)

[2.1 Encapsulation 8](#_Toc101863544)

[2.2 Modularity 8](#_Toc101863545)

[2.3 Hierarchy 9](#_Toc101863546)

[3 Design Principle 10](#_Toc101863547)

[3.1 Don’t Repeat Yourself Principle 10](#_Toc101863548)

[3.2 Complete And Consistent Principle 11](#_Toc101863549)

[3.4 Single Responsibility Principle 12](#_Toc101863550)

[4 Design Pattern 13](#_Toc101863551)

[4.1 Mediator Pattern 13](#_Toc101863552)

[4.2 Singleton Pattern 13](#_Toc101863553)

[4.3 MVC Pattern 14](#_Toc101863554)

[5 Additional Features/Work (Optional) 15](#_Toc101863555)

# Introduction

We have learned how to write client and server in java last semester, so we want to apply this to our project.

## Application Information

The theme of our application is the Rock-Paper-Scissors Game. Our game has three modes to choose: one round, three round, and five round. We made this game that allows two players to play together through the computers.

## Requirement Analysis

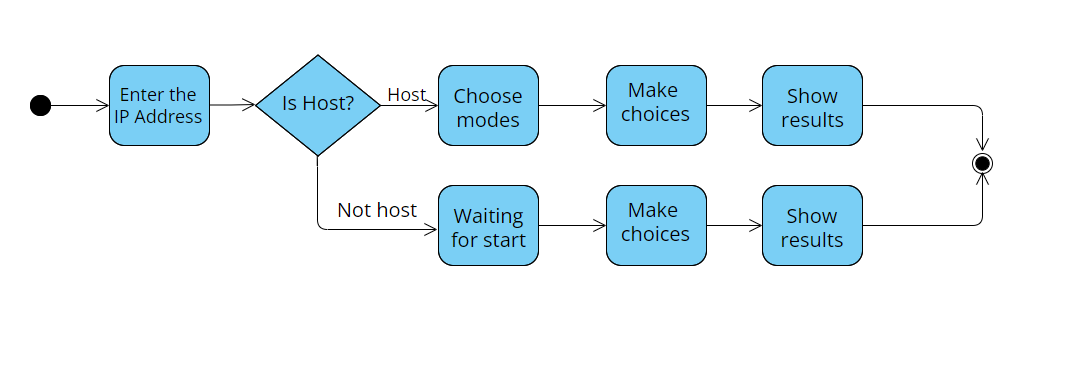
1. This game requires two players to play.
2. Every player needs to enter an IP address to enter the game.
3. The first player to enter will be the host by default, only the host has the right to choose the mode and start the game.
4. Mode cannot be selected when host is waiting for another player.
5. During the game, players only have 10 seconds to choose in each round, and 1 second to display the result of each round.
6. In each round of the game, the player can only choose once. If you choose multiple times, the system will use the first choice.
7. If the player does not make a choice within 10 seconds, the system will automatically choose ROCK for you.
8. During the game, when one round is over, it will go to the next round automatically.
9. When the player has made all the choices, the game will automatically jump to the end page and display all the results.
10. If one player quits mid-game, the other player will get a warning automatically and quit.

## Software Architecture

The following diagram shows the structure of the application:

  
Figure-1: UML Structural diagram (package diagram)

The following diagram shows the behaviour (flow) of the application:

  
Figure-2: UML Activity diagram

## How to Use

First Step: Enter the IP Address

When players enter the game, players need to enter the IP address, as shown in the picture below.

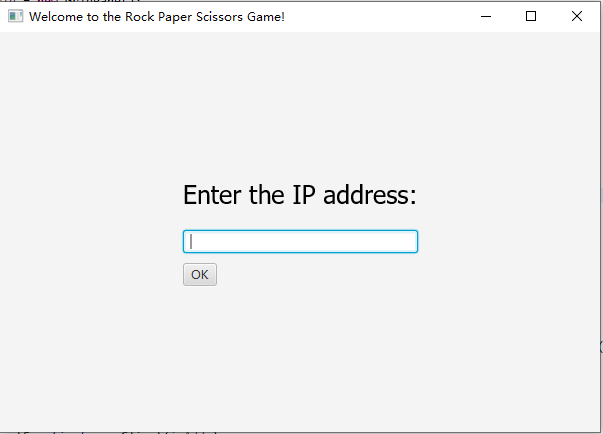


Figure-3: User Interface

Second Step: Choose the Modes(host) and Waiting(non-host)

After the player enters the IP address, the player needs to click OK to enter the game. The player who firstly enter the game will become the host, and he has three modes (corresponding to three buttons) to choose. For non-host player, he can only wait for host to start.

Third Step: Make Choice

When players enter the selection interface, there are three buttons corresponding to Rock, Paper and Scissors. They can make choice by clicking the button.

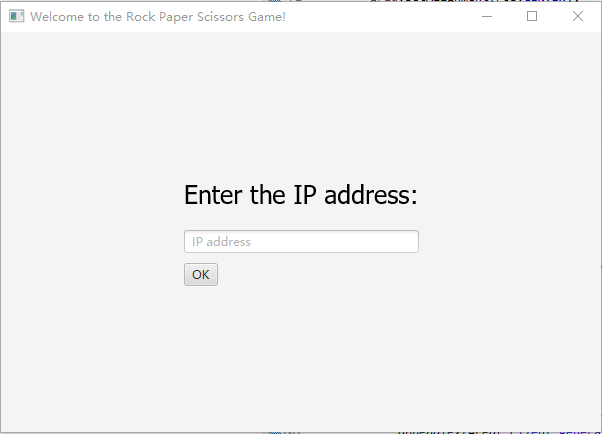
Fourth Step: Result and Exit

When the player has made all the choices, the game will automatically jump to the end page and display all the results. If you want to exit the game, please directly click exit in the upper right corner of the window.

## Details / Demonstration

This application has 5 pages. Page one is for players to enter the IP address.

The following figure shows the screenshot of IP address page(scene). Players can enter IP address in the input box.

  
Figure-4: IP Address Page

The following figures are screenshots of choosing modes and waiting pages, for the host, there are three buttons that you can click to choose the modes.



Figure-5: Choose Modes Page (For host)

For non-host, player can only wait for the host to start the game.

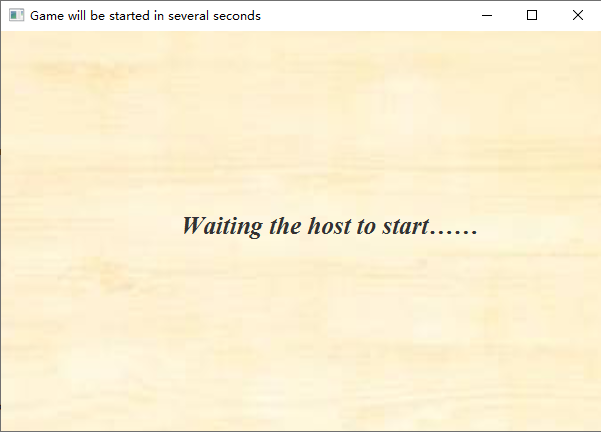


Figure-6: Waiting Page (For non-host)

The following figure is screenshot of making a choice page, the clock icon in the upper right corner of the picture is a countdown. Below the picture there are three buttons corresponding to Rock, Paper, and Scissors.

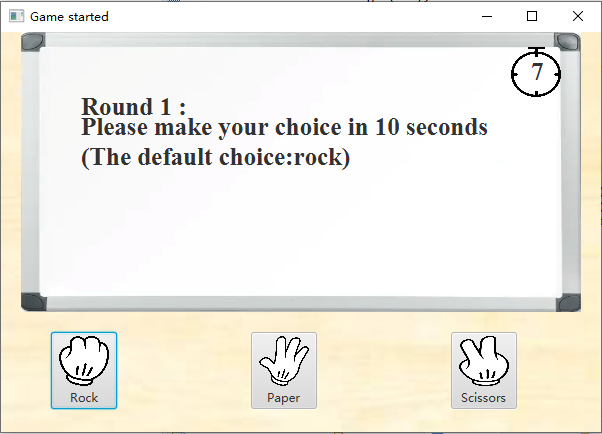


Figure-7: Make a Choice Page

The following figures are screenshots of result pages, the first row in pink is the total score between you and your opponent (You – You Opponent). The second row is the result, under the result are the result of each round. If you win, the font colour is green, if you lose, the font colour is red. In the case of the tie, the font colour is purple.

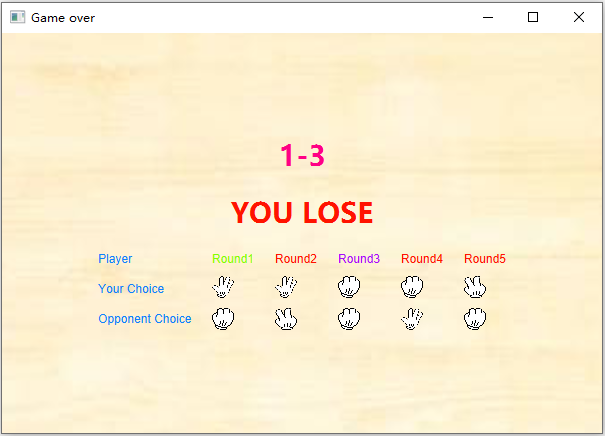
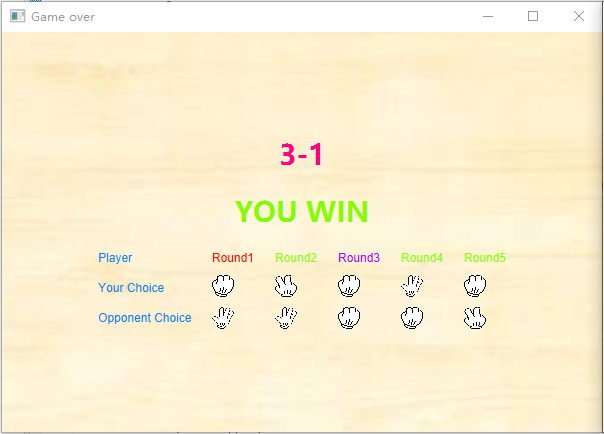


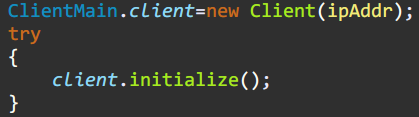
Figure-8: Result Pages

# Object Model

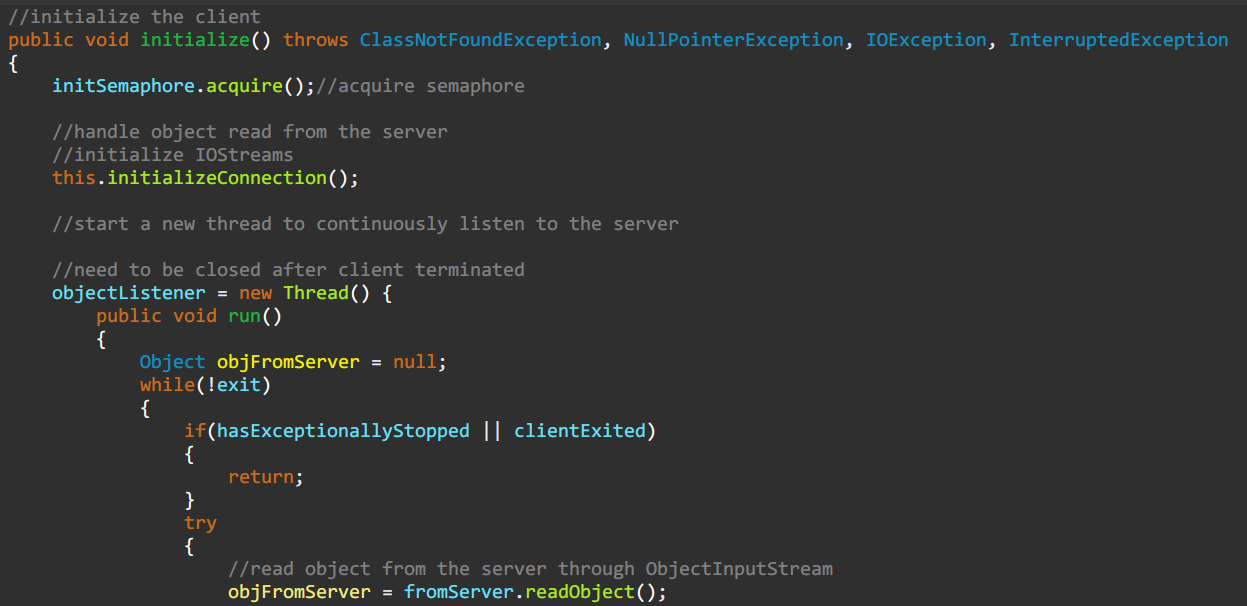
We have adopted the object model concept to develop the application.

## Encapsulation

For the model-controller, it provides a encapsulated method *initialize()* to let the view-controller to initialize an instance of model-controller.

****

*ClientMain (View-Controller)*

****

*Client (Model-Controller)*

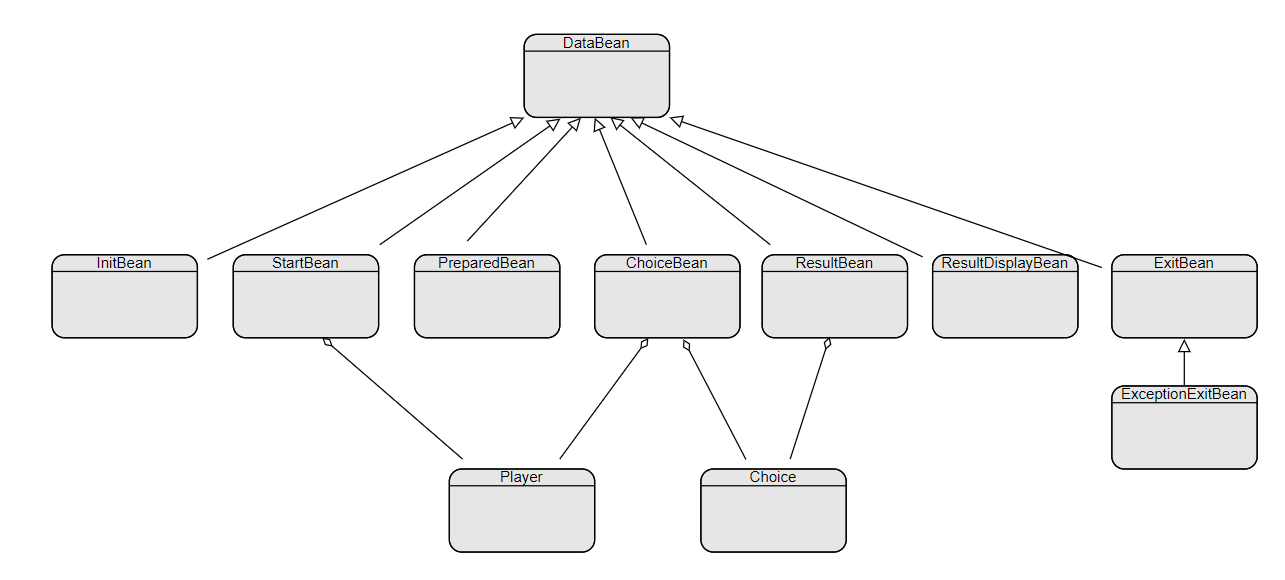
## Modularity

Our work is subdivided according to MVC pattern. Grant is responsible for developing the view: pages, cascading style sheet, pop-up windows etc. Veronica mainly focuses on view-controller, that is, creating and switching pages, restrict user’s behavior (making a choice), countdown timer etc. Polo makes contribution on model and model-controller such as data model (JavaBeans), socket connection, game logic etc.

## Hierarchy

Class Structure

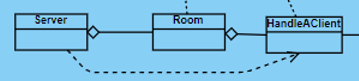
Data is encapsulated in the form of JavaBean, including logical Player and Choice that the player has chosen.

****

*Model Hierarchy*

Object Structure

Aggregation is applied in the server-side.



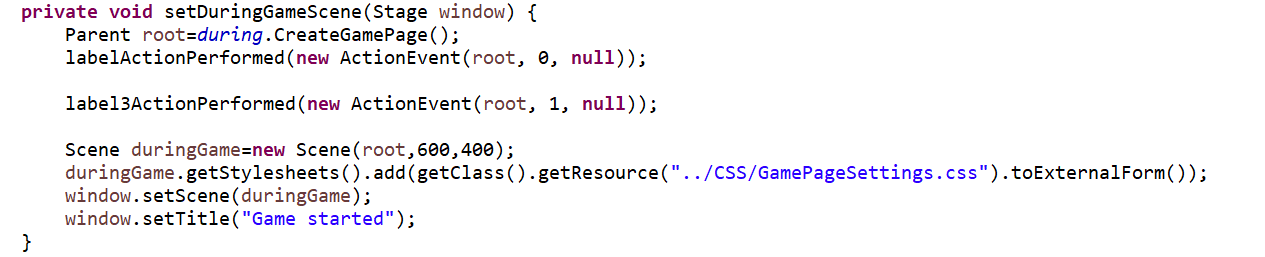
*Model-controller Hierarchy*

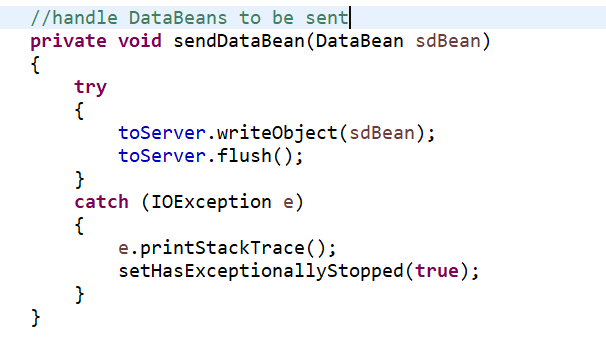
# Design Principle

We have followed the design principles to develop the application.

## Don’t Repeat Yourself Principle

The definition of DRY principle is about abstracting out the duplicate codes. It looks pretty straight-forward but turns out to be critical in coding for easy to maintain and reuse. This principle has two purposes. The first one is to have each piece of information and behaviour in a single sensible place of the system. The second one is to centralize the duplicate codes in a unique place for integrity and apply changes for all.

Here we have two examples. The first one is setDuringGameScene method (In ClientMain class). Because the welcome page has 3 buttons, these buttons are set on the same action which is turn to the DuringTheGame page but with different mode number. We abstract these codes out into one method.

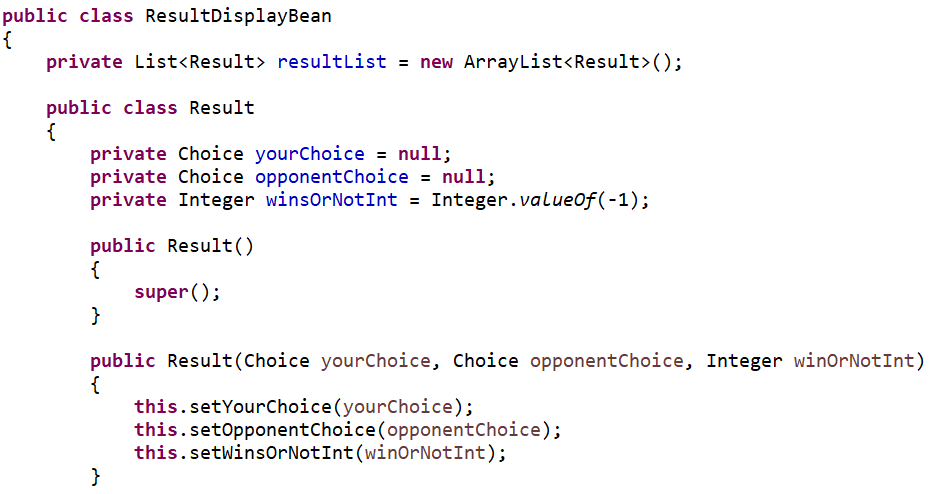
The second one is sendDataBean method (In Client class). Because this game is an online game, client needs to send different kinds of data to the server. Notice that we have one abstract class called DataBean. And there are a lot of concrete classes extending it such as ChoiceBean, StartBean and so on. So, we simply abstract the duplicate codes into one method. When the client needs to send data to the server, we can simply call the sendDataBean method.

## Complete And Consistent Principle

The C&C principle has two parts: Complete and Consistent.

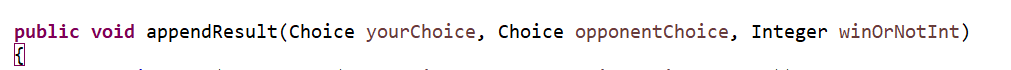
The definition of Complete is that each class should have exactly one role but similar type of behaviours can also add to that class to make it more complete. Therefore, there are always related behaviours that they will exist together in nature. In this game, for client we have initialize( ) and terminate( ) method. For server, we have clientRegister( ) and clientDeregister(roomNo: int, uuid: UUID).

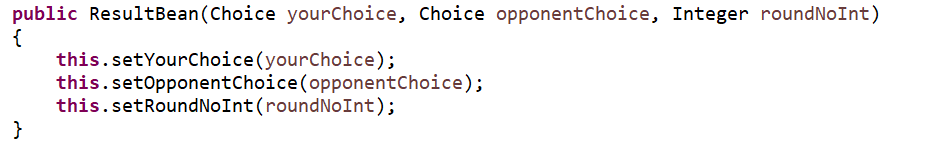
There are four rules for Consistent part.

The first one is the naming convention of the methods should be in the same manner. For the client side, we have the ResultDisplayBean class. For the server side, we have the ResultBean class. It’s not hard to see they are all in the same manner.

The second one is the parameters the methods take should be in the same order. Below are the examples I list. We can discover that the first parameter is always your choice and the second parameter is always the opponent’s choice.

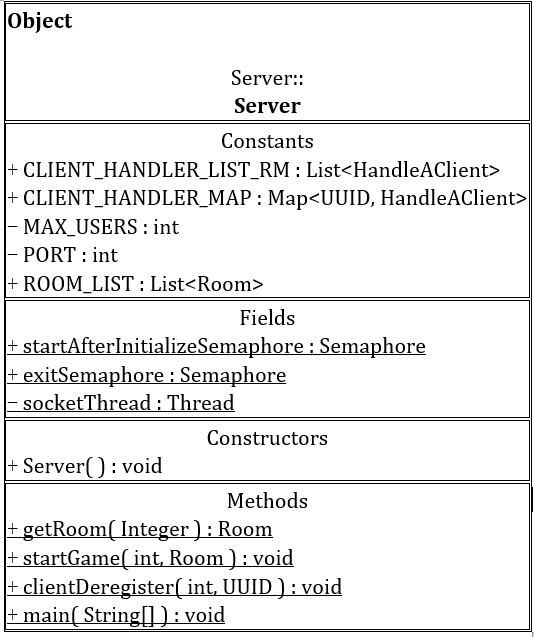
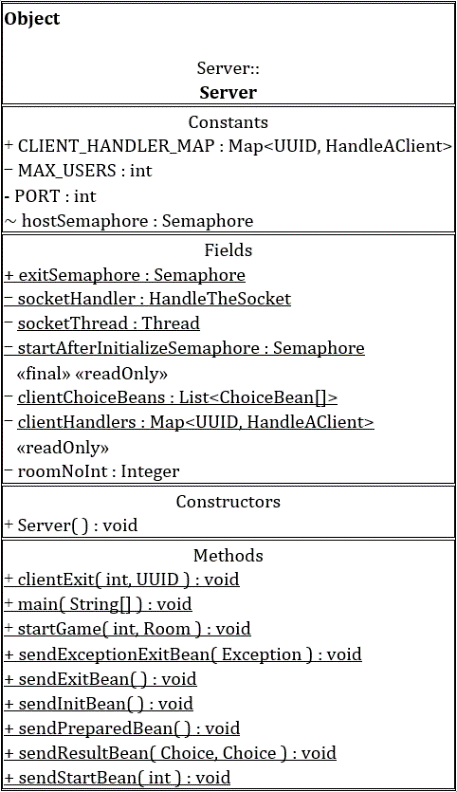


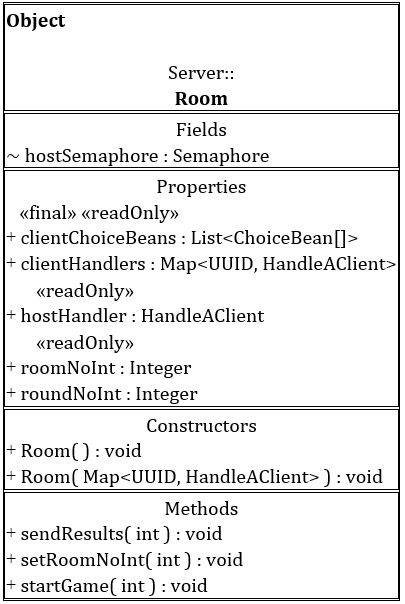
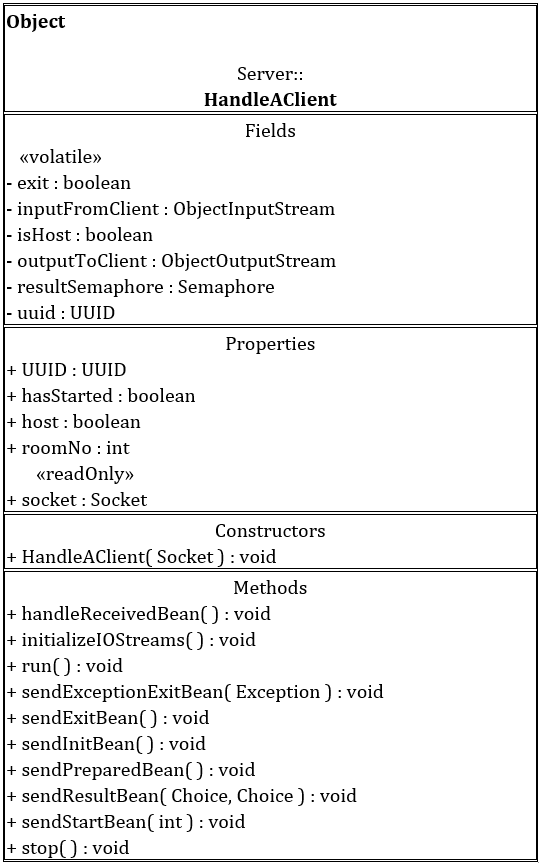


The third one is the naming for methods and variables should be meaningful. Below pictures shows us the sendResultBean method and ResultBean constructor. Their names are quite meaningful for us to get the information directly about what these methods do for us.

## 3.4 Single Responsibility Principle

The definition of SRP is that: a class should have only one reason to change. We’ve applied SRP to the Server class because it was a large, cumbersome class containing too many methods and attributes. What’s more, the former Server class was designed for handling only 2 players, which is not reasonable.

Therefore, for maintainability and scalability, we’ve separate responsibilities from the former Server class. For example, the duty of sending data to client would be done by HandleAClient class. So, the maintainability increases due to much slimmer Server class acting as a role of a stateless class containing several static methods and some constants. As for the new Room class, it can handle one game of each 2 players, which can be scaled by incrementing the room number. Theoretically, the Server allows multiple clients to connect and play games on, and there is no interference among rooms. These are benefits from applying single responsibility principle.



*After*

*Before*

# Design Pattern

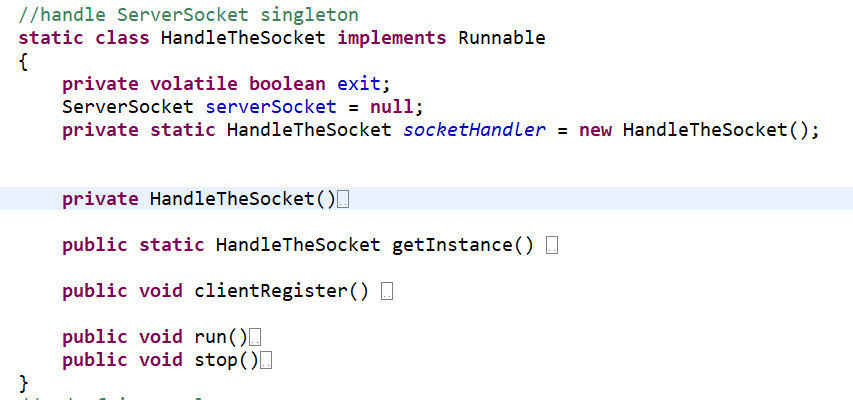
We have used the design patterns to develop the application.

## Mediator Pattern

Mediator pattern is used to reduce communication complexity between multiple objects or classes. For the model-controller, it would be very complicated if every class has a relationship with a client. Therefore, HandleAClient class has become the mediator, transferring messages between the client and the conceptual server (Serer class and Room class). By applying mediator pattern, the cost of communication decreases a lot and the classes are more loosely coupled.

## Singleton Pattern

The intent of the Singleton pattern is to ensure that a class has only one instance and to provide a global point of access to it. For HandleTheSocket class, who is responsible for handling the socket connection, is considered appropriated to be applied with singleton pattern. By doing so, clients are able to establish connections concurrently without causing any error due to there is only one instance of HandleTheSocket class, which is held by Server class.



## MVC Pattern

As for the model-controller composited by Server, Room, HandleAClient, and Client class, is responsible for sending commands to update the models’ state. The view-controller, that is ClientMain class, can access processed and duplicated data by invoking encapsulated methods from model-controller. But the model used for message passing among classes of model-controller, for instance, InitBean class (used to initialize a client), is not visible to the view-controller.

With Model-view-controller pattern applied, our system is more loosely coupled and we were able to develop different parts simultaneously, which has increased the efficiency of development.

****

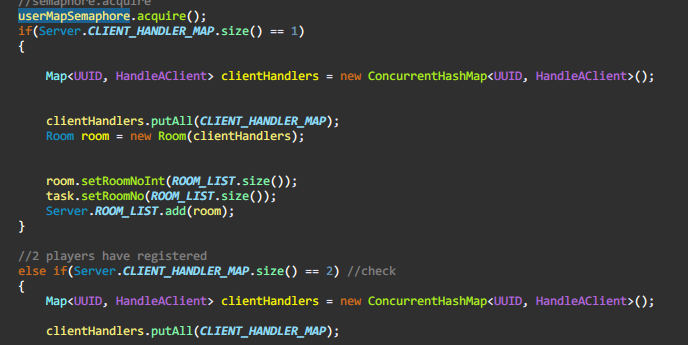
# Additional Features/Work

## Message Passing

We’ve applied both synchronous and asynchronous message passing method. For the message passing between Server and Client, we’ve made it asynchronous, that is, states change according to the events, for example, receiving specific data. For the message passing between view-controller and model-controller, we’ve made it synchronous because they have composition relationship and JavaFX build-in AnimationTimer class makes it convenient to keep them in the same status.

## Multi-threading and Concurrency Control

We’ve applied mutual exclusion and Java build-in ConcurrentHashMap to solve the concurrency problems. Besides, as mentioned previously, singleton pattern also helps us in concurrency control problems.



*Semaphore as Mutual Exclusion*

## Exception Handling