Bilkent University

Department of Computer Engineering

Object-Oriented Software Engineering Project

monopoly-game: Digital version of the well-known board game Monopoly

Design Report

Group 1C: Ziya Mukhtarov, Ege Kaan Gürkan, Alper Sarı, Mokhlaroyim Raupova, Javid Baghirov

Instructor: Eray Tüzün
Teaching Assistant(s): Barış Ardıç, Emre Sülün, Elgun Jabrayilzade

Iteration 2

December 13, 2020

Table of Contents

1 Introduction

One of the most popular board games of history is the Monopoly game. First published in 1935, it still holds its popularity to this day. As a group, we decided to digitalize this game since we all are lovers of it. We think that the Monopoly game has the potential to be a really good object-oriented project and fun to work on. We are even more excited to test this game by playing it after we complete the implementation phase.

We aim to create digital game software for the Monopoly Classic version. In the original board game, the players would take turns, buying and selling properties, managing their money, winning chance cards, and having fun. Each player rolls the dice at the beginning of their turn and plays accordingly to the number obtained by the dice roll, which makes the game dependent on luck. The board game has quite a few interesting rules; for example, if a player gets a double dice roll 3 times in a row, he/she is forced to go to the jail block. More on these rules are given later in this document.

Since we are digitalizing the board game, we decided to add many features that would otherwise be impossible to achieve. For instance, the gameplay will be held in an online environment where players get to play with their friends or random people all over the world without the requirement of being in the same room and risking their safety, especially during the quarantine. Moreover, the players can socialize and make new friends by using the video and voice chat feature.

1.1 Purpose of the system

Monopoly is a party game that can be played by 2-6 players, and is a cultural staple of any friendly gathering. This virtual iteration of the game was designed first with socialization and player useability in mind, incorporating distant communication technologies so that you can talk with those you are playing in real-time. The game has a user-friendly UI as well as an in-game guide to let anybody become proficient in playing monopoly online.

1.2 Design goals

The design goals for the game is should also be established, and while deciding on principles to base our design off on we concluded that the end-user should have a good experience while using our program. Therefore, our main design goals are as follows:

1.2.1 Functionality

The game will have all the necessary functionality required to play a game of Monopoly with 2-6 players and will support online play over the internet. On top of this, players will also be able to utilize a web camera and microphone to talk to each other as if they were in the same room. This added functionality will allow for a more fun experience.

1.2.2 User Friendliness

The game should help the user play in the least confusing manner possible and should handle most of its functionality without direct input so that the players will be able to navigate through the interface much more efficiently. Another point in favor of user-friendliness is that the game will have a comprehensive in-game guide that will help with any misunderstanding regarding the game’s functionality or subsystems

1.2.3 Runtime Efficiency

As mentioned in our analysis report, the game has several non-functional requirements regarding performance, it should work efficiently in almost every system. The system will be in two parts with multiple clients and a central server; therefore, we will focus on the efficiency of data flow between server and client during gameplay to make sure the game works smoothly with most user’s internet connection.

2 High-level software architecture

2.1 Subsystem decomposition

Our project has three high-level subsystems: “Client”, “Server”, and “Common” where the common subsystem is used as a middle ground for communication between server and client systems. In this manner, the client mostly has the user interface functionality side of our project, and it communicates with the server to send over commands from the players, where the server said input according to game logic.

We decided to use a client-server architecture when building our system, which we thought would fit well as the game was designed to be played over the internet. It also allowed us to host multiple game lobbies within a single cloud server, as the game logic behind monopoly never changes, we can create several instances of lobbies and games that have unique player groups that are handled by a single server machine.

The low-level subsystem decomposition is as follows:

Client:

- **Network**: This part of the subsystem handles communication between the client instance and the server-model structure. The user input taken from the UI systems is sent over the network using the common network packets, to the server.
- **Media Capture**: After a player joins a game lobby, they can choose to start a video and voice call with the other users within that lobby so that they can communicate in a more interpersonal manner. This subsystem is responsible for capturing their webcam and/or microphone input.
- **User Interface**: The user interface subsystem is divided into two sections, for navigating the program menus until the start of a game, and for handling gameplay UI after a game has started.
  - **Menu UI**: The menu UI handles navigation within the client itself, and only interacts with the server subsystem while the user joins and uses the lobby functionalities such as banning players and starting a game.
  - **Gameplay UI**: The gameplay UI exists to handle user’s input during their turn, after a game has started, and sends that input to the server instance using the network packets within the common subsystem. The server will then process said input, and update all client UIs accordingly.
- **Audio System**: The audio system plays the music and audio related to the game, including the received microphone input.

Server:

- **Network**: This part of the subsystem is the counterpart to the network system in a client, it exists to receive and process network packets sent from a client and to send updates to the client whenever some change occurs within a lobby or a game.
- **Lobby**: The lobby system is there to manage the multitudes of lobbies where the users will be able to join and start a game amongst themselves. There is a list of lobby instances in the server, and every lobby has a list of user instances as well. These lists are updated as lobbies are created and users join and leave said lobbies.
- **Gameplay**: The game logic section of the server handles player commands sent from the client instances and manipulates the game logic according to the official monopoly ruleset. It uses several systems such as the trade and the auction system to handle the game from start to finish, as well as updating the clients over the network whenever the game state changes.

Common: The common subsystem is there to facilitate common ground between client instances and the server as they send data to each other. It has classes and data types such as LobbyPacketData and PacketTypeData which allow that communication. By that measure, both the server and the client subsystems are coupled to the common subsystem.

2.2 Hardware/software mapping

Since our monopoly game will be implemented using client-server architecture, and therefore users of the program will require an internet connection to be able to play the game. The client will be light enough to run on an average computer of modern standards, and will not require any special hardware.

To be able to play, the users will need a keyboard and mouse, and to be able to utilize the social aspects of the game such as video and voice streaming, they will need a webcam and microphone connected to their machine. However, the use of microphone and cameras are not mandatory, and a user will still be able to play with others even if they have no such hardware installed.

The server itself will be hosted by Amazon, and will not require any downtime maintenance, most of the data and functionality are hosted on this server and it has no UI, as only the clients will have a graphical user interface.

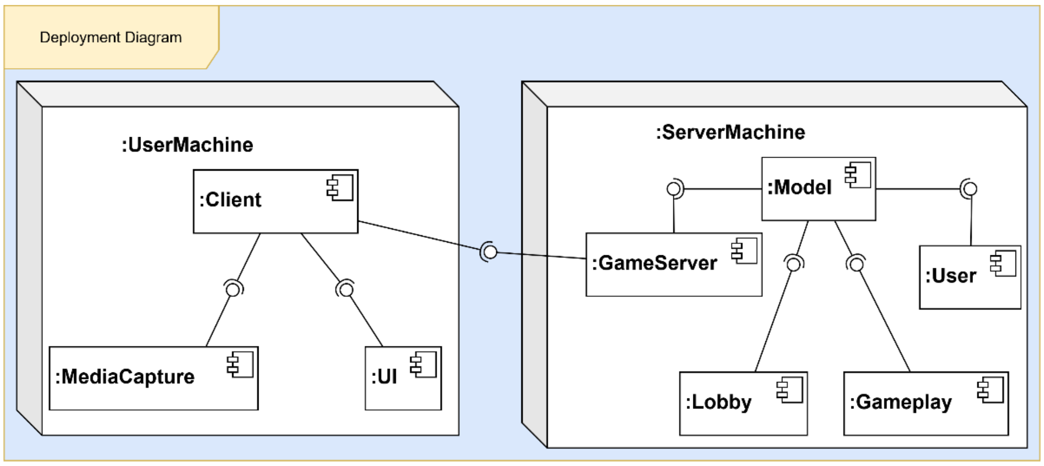

Figure 1. Deployment Diagram

2.3 Persistent data management

Monopoly does not require any sort of persistent data management in its first iteration and therefore it does not require any sort of file or database management. There are several reasons behind that decision, but mainly it is because the users cannot save an ongoing game, or create personal user accounts beyond having unique usernames. The server should also not close down, and if it does then the players will not be able to access the game regardless, therefore no persistent data is required.

However, there is a list of features that, if we decide to implement them, will include being able to save a game state which can be returned to later and registering player accounts with user names and also relevant data such as the total number of games played and won, etc. These later functionalities will require a database system that is managed on a server to store saved game states and player account data that will be fetched as a user goes through the login screen or picks a saved game to continue.

2.4 Access control and security

Since the game works between a server instance and several client instances, there must be limits on the functionalities that the clients can access. The most notable of these restrictions is that a user can only influence the lobby that they have connected to, and only a lobby owner can change the lobby settings and has the capability of banning players from the said lobby.

Also, during the game, only the player in turn order has access to the game's functionalities as the game goes through every user within the list, and allow them to run their methods. This means that while the classes which the user associates with will have access to functions that manipulate game logic, only one instance will have that access at all times.

	Model	**Lobby (the actor is in)**	**Game**	**Auction**	**Property**

User	`getLobbies()`
	`createLobby()`
	`viewGuide()`
	`changeUserName()`
	`viewGuide()`
		`setReadinessToPlay()`
		`join()`
		`leave()`
		`kickVote()`			

Lobby Owner		`ban(user)`
		`setLobbyName(name)`
		`setPassword(pass)`
		`makeAdmin(user)`
		`setPlayerLimit()`			

Game Player		`textChat()`
		`videoChat()`
		`voiceChat()`
			`rollDice()`
			`startTrade()`
			`startAuction()`
			`doAction(action)`
				`bid(increaseAmount)`
				`setSatisfied()`
				`getCurrentBid()`
				`getSatisfiedPlayers()`
					`buy()`
					`buildHouse()`
					`buildHotel()`
					`mortgage()`
					`unmortgage()`
					`getRentCost()`

Table 1. General Access Matrix

Table 1 lists the methods which the actors will have access to throughout the game. The user class can only access the functions that are available before the game starts, at which point only the Game Player can access them. The lobby owner also has some permission to manipulate the lobby before the game starts.

	Trade

Trading GamePlayer	`addItem(item)`
	`removeItem(item)`
	`agree()`
	`disagree()`
	`getItemsOnTrade()`

Other GamePlayer	`getItemsOnTrade()`
	`getTradingPlayers()`

Table 2. Trading Access Matrix

The players who are involved in a trade can manipulate the items within that trade as it goes on, but the other players can only observe and do not have access to any of the trade controls.

2.5 Boundary conditions

During initialization, the client executable will be run by the user and they will log into the server. After they log in, their user account will be initialized and they will be able to join or create lobbies on the server.

After a game ends, or a player is kicked from a game or a lobby they can join another lobby as long as the client executable is still running on their machine. They can also abandon an ongoing game from the in-game menu, and doing so will also send them to the main menu. The client can be terminated using the “Quit Game” button in the main menu

Finally, if a client disconnects during an ongoing game, the client will act as if the player was kicked from the game in a normal manner, and the game will continue as long as the minimum viable number of players stay connected.

The server will require initialization once during the launch of the app, where it will be run on an Amazon server, but as the service provider allows us to run the server continuously, it will not require termination and will have no downtime for maintenance, etc.

3 Low-level design

3.1 Final object design

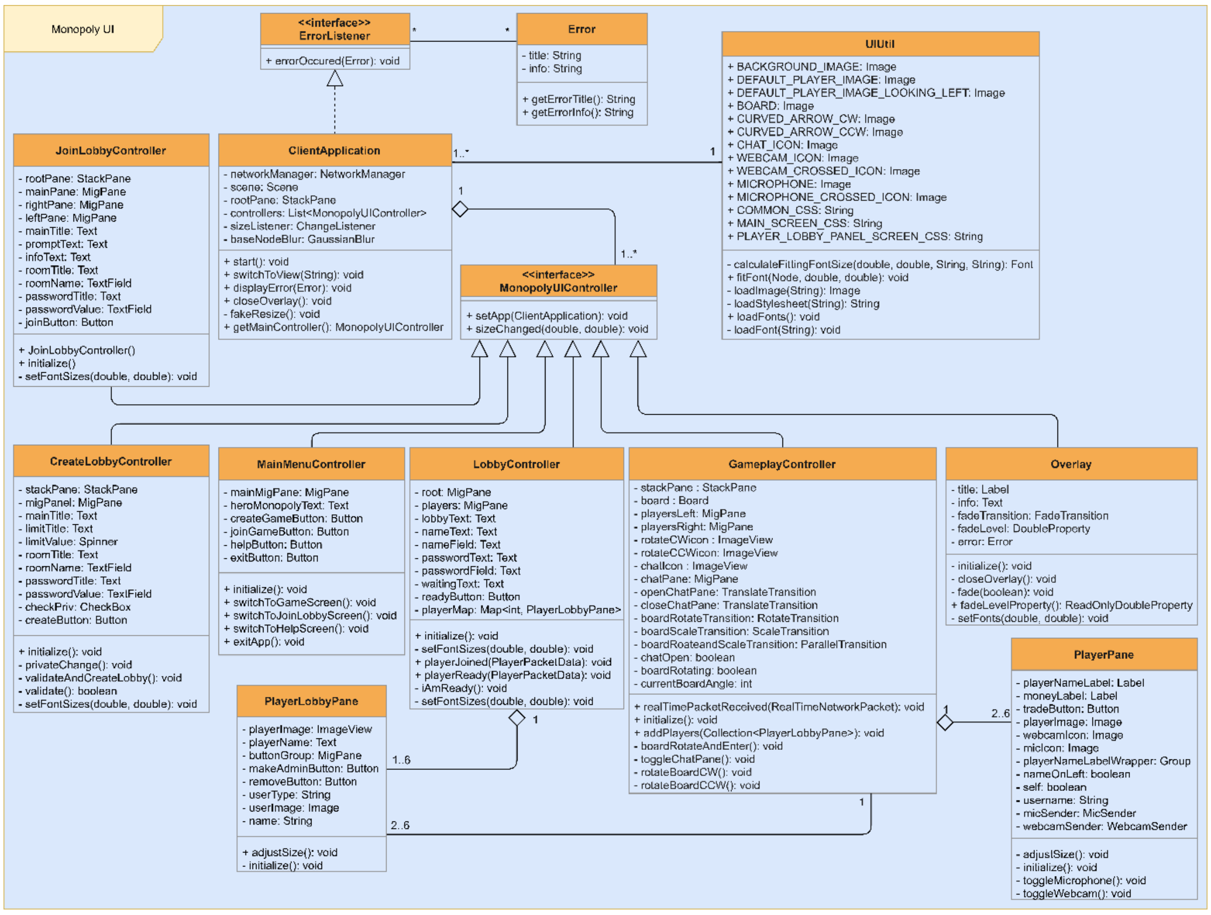

Figure 2. Monopoly Client UI Class Diagram

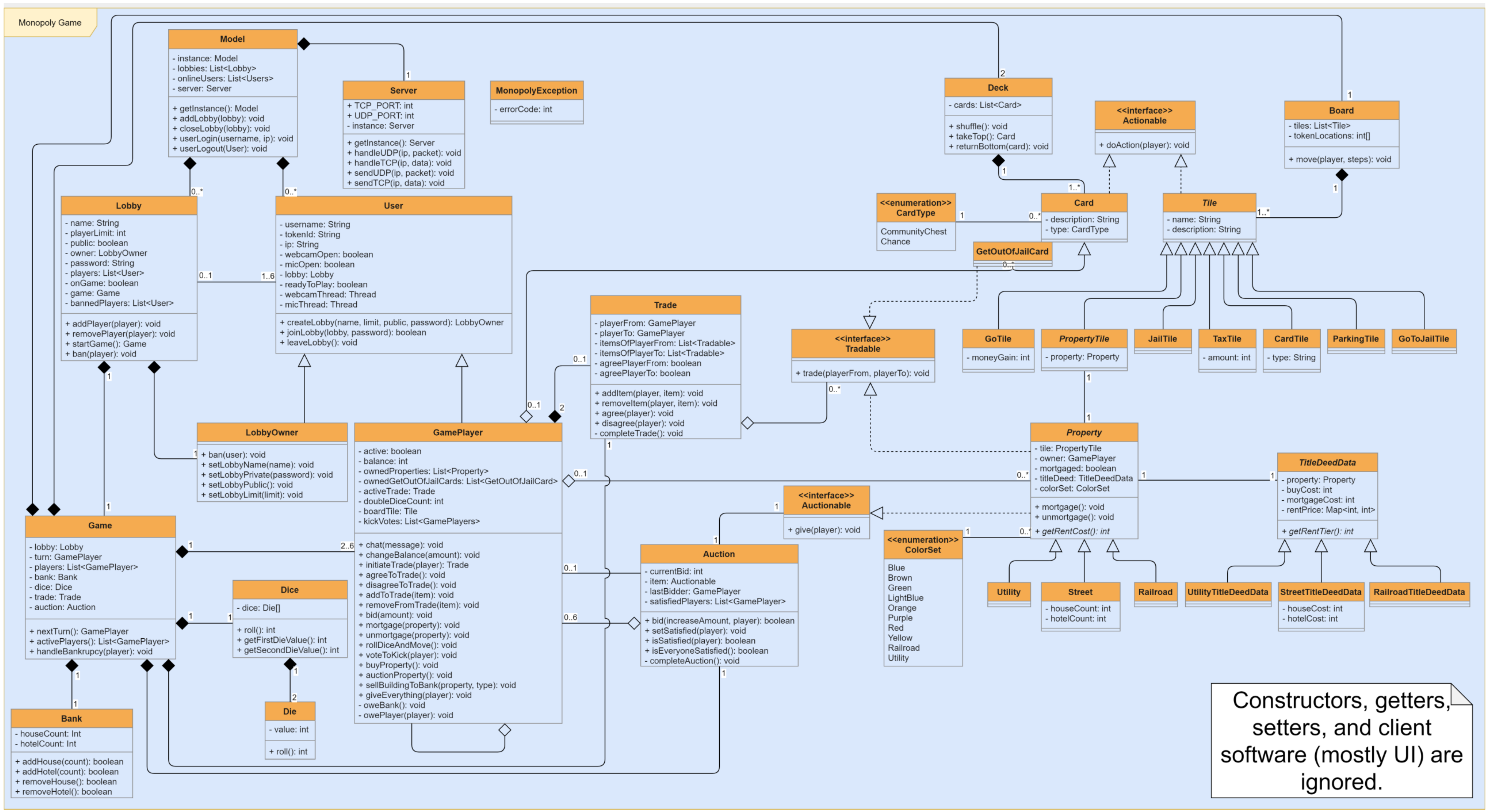

Figure 3. Monopoly Server Class Diagram

3.2 Internal Packages

3.2.1 “server” Submodule

3.2.1.1 monopoly.network

This package holds all the necessary classes for the server infrastructure of the application.

3.2.1.1.1 monopoly.network.lobby

Holds the necessary classes for the Lobby structure.

3.2.1.1.2 monopoly.network.demo

Holds a demo video/voice chat server application.

3.2.2 “common” Submodule

3.2.2.1 monopoly.network

This is the package that holds the necessary classes for the common submodule to work.

3.2.2.1.1 monopoly.network.packet.realtime

This package holds the classes for video/audio packets to be sent to the server.

3.2.2.1.2 monopoly.network.packet.important

This package hosts classes for the important game status changing packets.

3.2.3 “client” Submodule

3.2.3.1 monopoly.ui

This package holds the necessary UI classes, the CSS, FXML files, images, and fonts for the frontend of the app.

3.2.3.2 monopoly.network.demo

This is the package that holds the classes for playing audio and a single demo client.

3.2.3.3 monopoly.network.sender

This package holds the sender classes for the microphone and camera inputs.

3.3 External Packages

3.3.1 com.esotericsoftware.kryonet

This external library provides low-level network functionality.

3.3.2 org.slf4j

We are using this library for logging purposes.

3.3.3 org.projectlombok.lombok

Lombok library takes the need of writing boilerplate codes from us such as getters and setters.

3.3.4 org.openjfx

We are using JavaFX library from this package for the UI.

3.3.5 com.miglayout.miglayout-javafx

MigLayout is a simple but powerful layout manager that we are using for UI design.

3.3.6 com.github.sarxos.webcam-capture

This library provides us with a platform-independent way of accessing user webcams.

3.4 Class Interfaces

See Javadoc: http://3.131.85.191. All 3 submodules’ Javadoc is available.

3.5 Design Patterns

3.5.1 Singleton

The monopoly.GameServer class from the server subsystem of our monopoly game is following one of the most popular creational design patterns which is the singleton pattern. It is because, when there is already an instance of the server, meaning when the server is currently open, it uses a specific port address. When a second instance of the server is to be created, it will also try to use the same port whereas it should not. This is why we want to have only one instance of GameServer and therefore, it is a singleton.

3.5.2 Observer

Considering the UI components such as buttons, it is almost impossible to exclude observer design pattern in our project. It is one of the many behavioral patterns and defines a dependency between objects which helps to notify all the individual components when an object changes its state. In our implementation, error listeners are used for achieving this job and they can be the most obvious examples of the observer pattern.

The listener interface is the monopoly.ErrorListener interface from the client subsystem. monopoly.network.NetworkManager class holds a list of these ErrorListeners, that gets notified when an error occurs. It is then the listener’s job to handle this error, according to the Observer pattern.

3.5.3 Adapter

Another design pattern that we are using is the adapter pattern, a commonly used structural design pattern in Java. We are doing so by having a monopoly.EntitiesWithId class from the server subsystem to maintain the job of an “adapter”. When the client – monopoly.Model wants to call a method for an adaptee – java.util.Map, the adapter handles it in a way that client does not know how it is done and in fact, does not even know that there is an adapter in between itself and the adaptee.

4 Improvement Summary

Below are the points that we improved upon in this iteration of our report:

- Access Matrices: Two access matrices have been added to the Access Control and Security section to clarify the points made in the explanation text and to specify how different classes can access what functionality
- Design Patterns: A “Design Patterns” section has been added which details the patterns we used during our implementation
- Javadoc: The URL in Section 3.4 is fixed and now all submodules’ Javadoc is valid and available
- Deployment diagram: The diagram is edited and now it combines webcam capture and microphone capture modules in one module called “MediaCapture” to be consistent with the subsystem design.