Bilkent University

Department of Computer Engineering

CS319-OBJECT ORIENTED SOFTWARE ENGINEERING

Q-Bitz: Q-Bitz

Iteration 2 - Design Report

Sait Aktürk, Zafer Tan Çankırı, Berkin İnan, Halil Şahiner, Abdullah Talayhan

Supervisor: Eray Tüzün

1.Introduction

1.1 Purpose of the system

In *Q-bitz* players use their special cubes to recreate patterns on the cards, gain points and win the game.[2] Although the original game has three rounds which will be played in order to win a game, our game will not be round based but mode based style which will consists of those rounds. We chose Q-bitz because it has the potential to be extended with new game modes and features that we will add.

In our game we have:

- Online Q-bitz
    - Multiplayer Game Modes
- Offline Q-bitz
    - Practice
    - Single Player Game
- Options
    - Adjustable color set
    - Sound options
    - Controls
- How to play

With this structure, the game will use the benefits of being digital, meaning that a user can choose multiplayer mode to enjoy the game with friends while in single player mode users can have a taste of an arcade game without the need of any other user.

1.2 Design goals

Design is one of the important process in the development of the system. It makes more clear the required systems that should be developed. As we also stated in our analysis report, the project has some non-functional requirements, and these requirements should become more clear in the design part. Particularly, these are the focuses of this paper. Following sections are the descriptions of the important design goals.

1.2.1 Criteria

End User Criteria

Usability:

One of the target user groups of the game are the children. Therefore, we paid attention to make the game simple as possible. We chose the controller parts of the game carefully, and we tried to make them modifiable as possible as we can. To reduce complexity of the game, we put the related screens, menus, and actions together. We tried to keep screens as clean as possible. The unnecessary components on the screens are ignored. The game can be both played with keyboard and mouse. We also thought that the game may be played with touch screens. Therefore, we are arguing about adding a gesture support to the game. There are also help menus in the game, and we prepared a “How to Play” module for the game. Therefore, the players can learn about the game and its controls.

Performance:

One of the main concerns of the project is the performance, since it a multiplayer game project. To overcome the performance issues we use a real time network communication protocol; a stable, well-known, and old database system; and a native GUI library for the project.

Maintenance Criteria

Extendibility:

Since the project will be developed in a fully object oriented and modular behaviour, all subparts of the project can work almost independently, and thanks to the object oriented design of the project, the abstractions and the composition of the project let us to extend the program easily by extending the existing classes. At the end, by writing some simple lines of code, the project can be extended easily.

Modifiability:

Since the project will be developed in a fully object oriented modular behaviour, all subparts of the project can work almost independently. Therefore, to modify the game, only thing need to be done is to change the parts related to the modifying process.

Reusability:

There are some subsystems which can be also used for other game projects, and similar kind of projects, since our program will be developed in a fully object oriented modular behaviour.

Portability:

Portability is also another important required for this project. Since it is a multiplayer playing supported game, it should also be easy to operate on a variety of systems. This situation is one of the main reasons why the project will be developed in Java language. Java programs run on Java Virtual Machine which is available for almost every system, and this situation also makes our game platform independent. As an extension to this, there are not any other dependencies which are required for our game to run. Therefore, it is an almost fully portable application.

Performance Criteria

Performance is one of the topics which is argued the most. The game includes 3D Graphics and it is a multiplayer game, thus there can be a reasonable amount of performance issues if bad design decisions are made. Since the game includes 3D Graphics for rotating a cube, these rotations has to be smooth. We have tackled this issue by implementing all 3D Graphics using native JavaFX classes.[1]

For the Client/Server model, we are going to use a callback mechanism in order to notify the users in an efficient manner. The amount of users supported depends on the current bandwidth used by the remote server that we are planning to use.

1.2.2 Design Patterns

The game will be implemented using Model View Controller(MVC) and Observer design patterns. MVC will be used for the user interface and classical game logic by using Manager classes as controllers with the mechanics depicted in the sequence diagram in Figure 1. Observer pattern will be used for Client-Server interactions with callback mechanism, meaning that the clients will be notified for the changes depending on different states of the game.

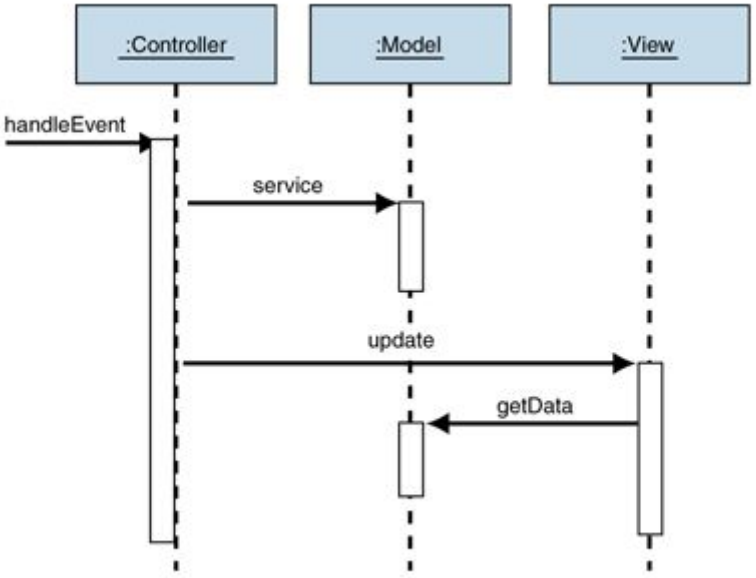


Figure 1: MVC Sequence Diagram

2. High-Level Software Architecture

2.1 Subsystem decomposition

Our system is composed of four main subsystems which are network, user interface, game logic and user.

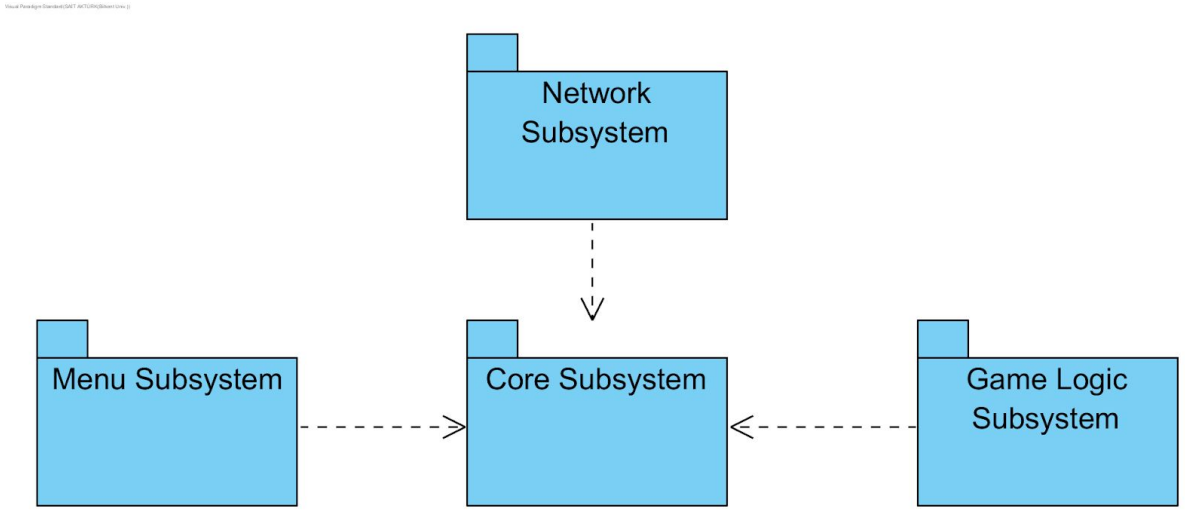


Figure 2: SubSystem Decomposition

Network Subsystem: This subsystem contains client and server classes that implements the interaction between the game and the server for handling multiplayer games. The network subsystem also captures the interactions between the server and database with database connector classes.

User Interface Subsystem: This subsystem contains the menu classes that enables the navigation features. We classified all the non-game screens as menu, and the playable screens as the game instance.

Game Logic Subsystem: This subsystem contains the classes related to an instance of the game which captures all the mechanics and controller classes that enables the game to be played.

Core Subsystem: This subsystem contains classes that application needs to be functional such as configuration, user and scene management classes.

2.2 Hardware/software mapping

Q-Bitz will be implemented with Java and will use JavaFX for the game graphics and the menus.

The game will be controlled using mouse and keyboard. The mouse will be used to interact with the menus and the gameplay screens. The cubes can be selected by clicking on it with the mouse and the keyboard will be used to rotate the cube.

The game can be played both online and offline. To play the game with other players, the device that the game is installed needs to have internet access.

For implementing the Server and database we are using a remote ubuntu instance with MySQL installed. The Server is also implemented in Java and runs on the remote machine that we own.

2.3 Persistent data management

Our system required databases that holds room and user's information. Any changes with user will be updated in databases. Cube patterns and information of single player stages will
hold in hard drive of the client, however, any progress in single player stage will also holds in user's information database to reach another computer in another game instance. Cube faces images and icons related to the game will hold in hard drive of the client as a ".png" format. Game musics and sounds will also hold in hard drive of the client as ".wav" format.

2.4 Access control and security

In order to play the game in multiplayer mode, the user needs to create an account. The account credentials will be stored in the database as previously mentioned in section 2.3. The passwords for each user will be stored in the database as salted SHA-256 hash values. During registration, an e-mail verification is required to prevent players to open redundant accounts for spamming game rooms or the database system itself.

Additionally the game has private rooms in multiplayer mode, these rooms are accessed via a special password dedicated to room. The system will request this password for each player that tries to join to any private room.

The inputs for login credentials will be checked for containing any kind of SQL keywords in order to prevent casual SQL injection attacks. The server can also blacklist a player based on IP address if there are multiple failed login attempts.

2.5 Boundary conditions

2.5.1 Starting of The Game

When the game is launched it checks for the local configuration files in the file system that the game is initialized. It will give an error if this configuration file is missing and proceed

by creating the necessary configuration files for starting the game. If the configuration files are not missing, the game starts normally and initializes the single player settings directly.

2.5.2 Server Availability of Multiplayer System

The multiplayer system will check whether the server is currently responding or not. This can happen due to maintenance or any kind of failure that the server may have. The current situation will be prompted to the player and players will only be able to play in single player mode until the server starts responding again.

2.5.3 Server Failure During a Multiplayer Game

If the server crashes during a Multiplayer Game instance, all the players will be disconnected. The game rooms will be destroyed as well. Any kind of progress such as experience that is already gained after a game ends will be saved only if the game has properly ended. Any kind of server unavailability during a game will not be able to update player data since the game is not ended yet.

2.5.4 Termination of The Game

If the user exits the game properly (without failure). There will be no unchanged saves because the options will be saved in the options menu. If the user exits during a game intentionally, the progress for that game will be lost. It shouldn’t continue because this is a game with time constraints.

3.Subsystem Services

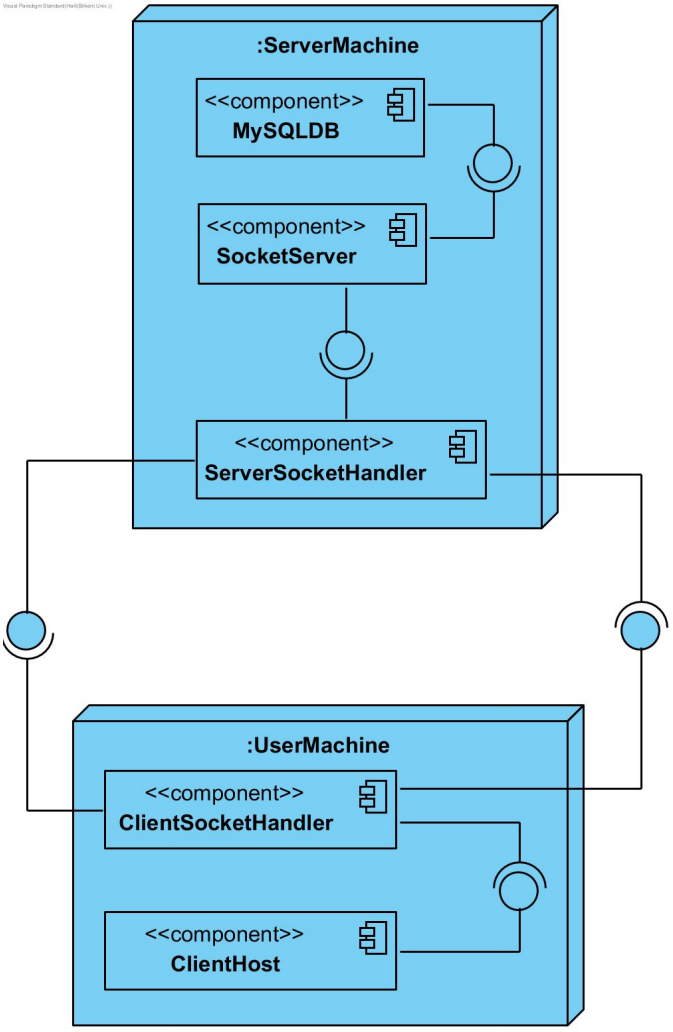


Figure 3: Deployment Diagram

3.1 Network Subsystem

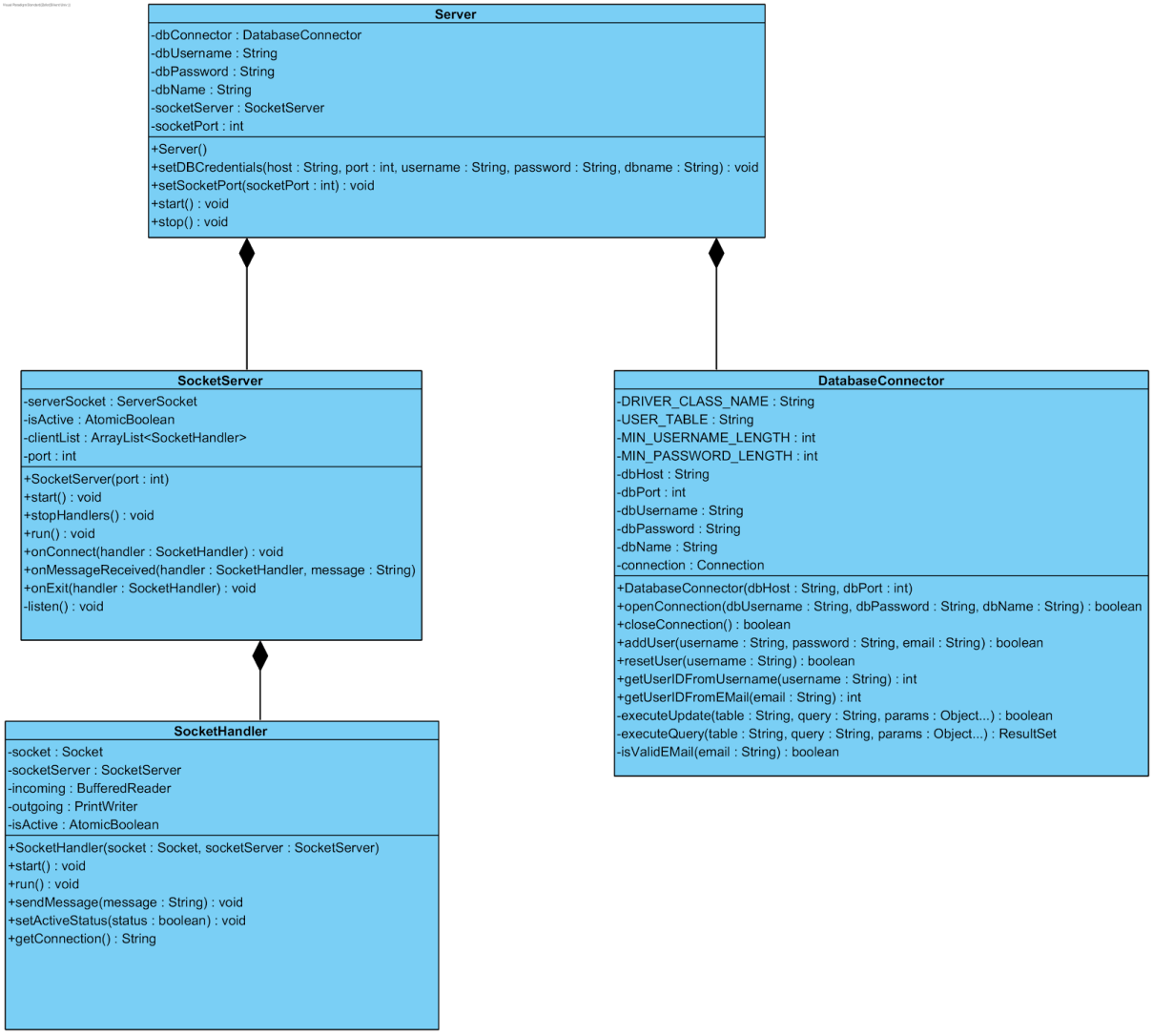


Figure 4: Network Subsystem

3.1.1 Server Class

This class is the main class which holds the Server properties of the game.
This class has instances of DatabaseConnector and SocketServer classes which are the main systems of the server of the game.

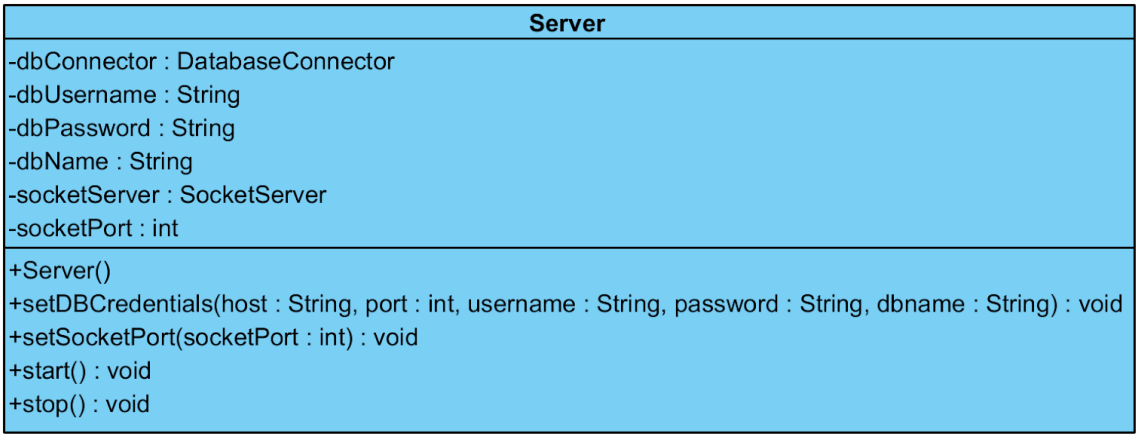


Figure 5: *Server Class*

Server:

Attribute:

private DatabaseConnector db : Holds the instance of DatabaseConnector object.

private String dbUsername : Holds the database connection username.

private String dbPassword : Holds the database connection password.

private String dbName : Holds the database name.

private SocketServer socketServer : Holds the instance of SocketServer object.

private int socketPort : Holds the socket connection port.

Constructor:

Server() : Constructor for Server Class.

Methods:

void setDBCredentials(String host, int port, String username, String password, String dbname)
This method initializes the database connection credentials.
host : Hostname of the Database Server

port : Port of the Database Server
username : Username for the Database Server
password : Password for the Database Server
dbname : Database Name on the Database Server

void setSocketPort(int socketPort)
This method initializes the port of socket connections.
socketPort : The connection port for the Socket Server.

void start()
This method starts the main systems of the server. (Database and SocketServer)

void stop()
This method stops the main systems of the server. (Database and SocketServer)

3.1.2 SocketServer Class

This class is used to hold socket operations.
It has a collection of ServerSocketHandlers and it is responsible for managing the socket connections with the clients.
It receives operation requests from clients, processes them, and sends responses.
This class is also an extension of Thread class. Therefore, it runs on a different thread other than the main thread of the server application.

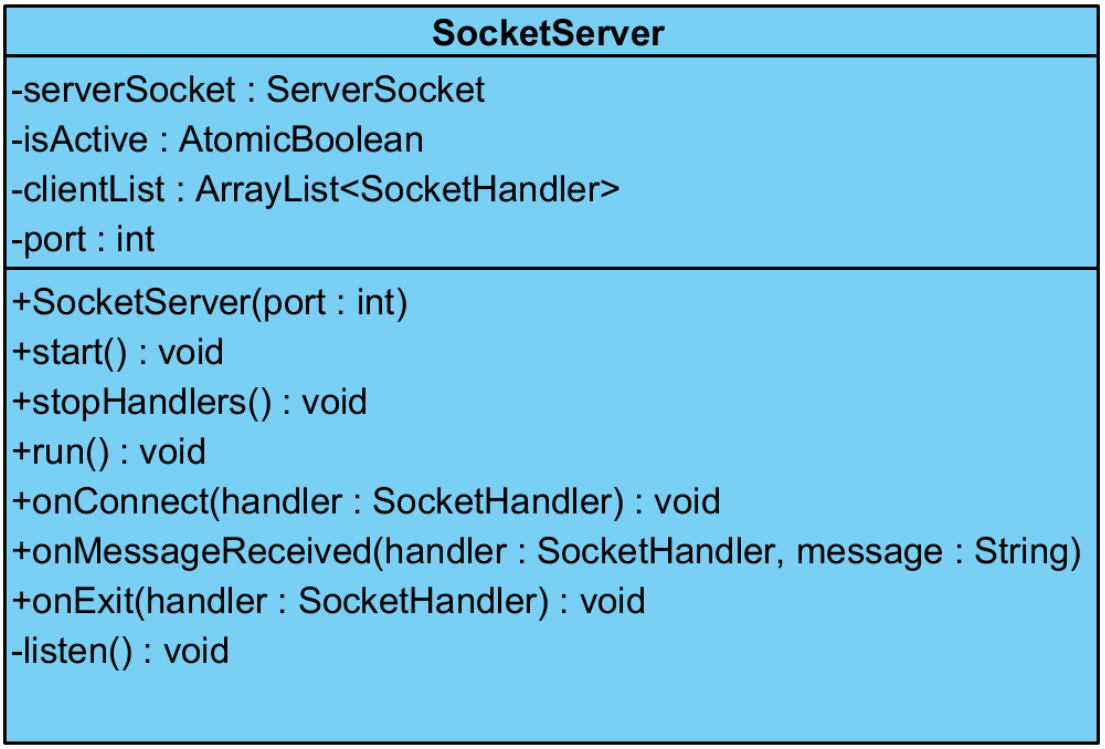


Figure 6: *SocketServer Class*

SocketServer:

Attributes:

private ServerSocket serverSocket : Holds the instance of socket server object.

private AtomicBoolean isActive : Holds the status of the socket handler.

private ArrayList<ServerSocketHandler> clientList : Holds the collection of ServerSocketHandler instances.

private int port : Holds the socket connection port.

Constructors:

SocketServer(int port)
Constructor for SocketServer Class.
port : The connection port of the socket connections.

Methods:

public synchronized void start()
This method start the thread of SocketServer Class, also it initializes the isActive property as true to start

void stopHandlers()
This method stops the ServerSocketHandlers in the collection.

public void run()
This method runs the operations of the Thread of SocketServer class.

void onConnect(ServerSocketHandler handler)
This method is a callback method which is called when a client connected.
handler : The handler for the socket connection with the client.

void onMessageReceived(ServerSocketHandler handler, String message)
This method is a callback method which is called when a client sent a message through the socket connection.
handler : The handler for the socket connection with the client.
message : The message came through the socket connection.

void onExit(ServerSocketHandler handler)
This method is a callback method which is called when a client disconnected.
handler : The handler for the socket connection with the client.

private void listen()
This method is responsible for listening the incoming socket connections. It accepts the incoming socket connection requests, and adds them to the collection.
It also starts different threads for each socket connection.

3.1.3 ServerSocketHandler Class

This class is used to handle socket connections.

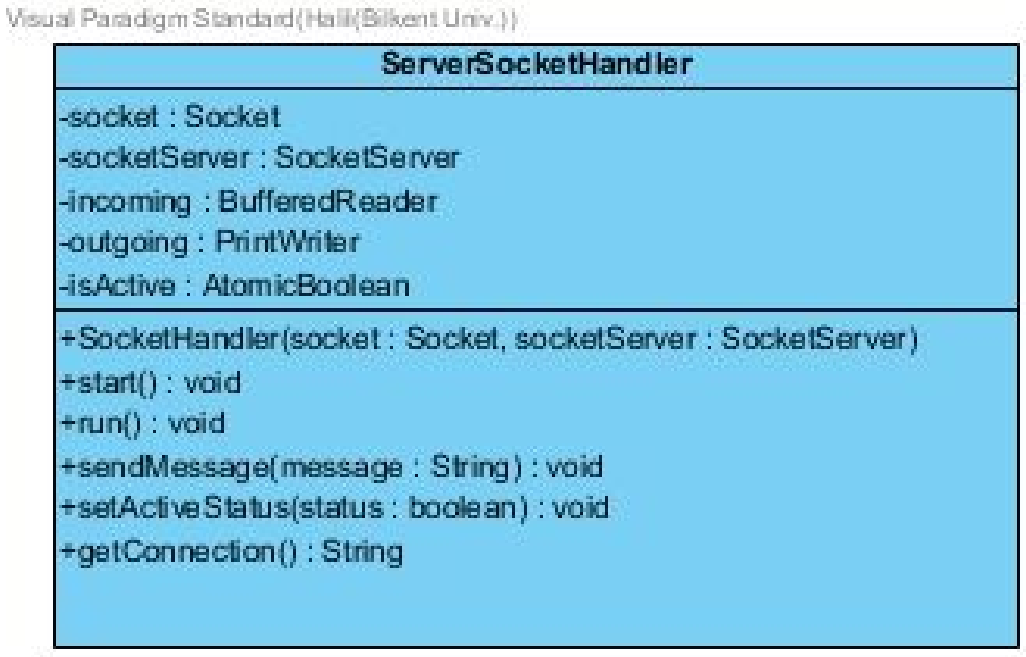


Figure 7: *ServerSocketHandler Class*

ServerSocketHandler:

Attributes:

private Socket socket : Holds the instance of socket object of the client.

private SocketServer socketServer : Holds the instance of socket server object.

private BufferedReader incoming : Holds the incoming stream from the client.

private PrintWriter outgoing : Holds the outcoming stream to the client.

private AtomicBoolean isActive : Holds the status of the socket handler.

Constructors:

ServerSocketHandler(Socket socket, SocketServer socketServer)
Constructor for ServerSocketHandler Class.
socket : The socket object for the connection.
socketServer : The instance of the SocketServer which holds the collection of ServerSocketHandlers.

Methods:

public synchronized void start()
This method starts the thread of ServerSocketHandler.

public void run()
This method handles the incoming messages over socket connection.

void sendMessage(String message)
This method sends message to the client over the socket connection.
message : The string message.
void setActiveStatus(boolean status)
This method sets the active status of the ServerSocketHandler.
status : The active status.

String getConnectionIP()
This method returns the IP of the client.
Returns the IP of the client.

3.1.4 DatabaseConnector Class

This class is used to handle database operations.
It has specific methods for predefined database operations.
It also has dynamic query methods which can be used querying DB server easily.

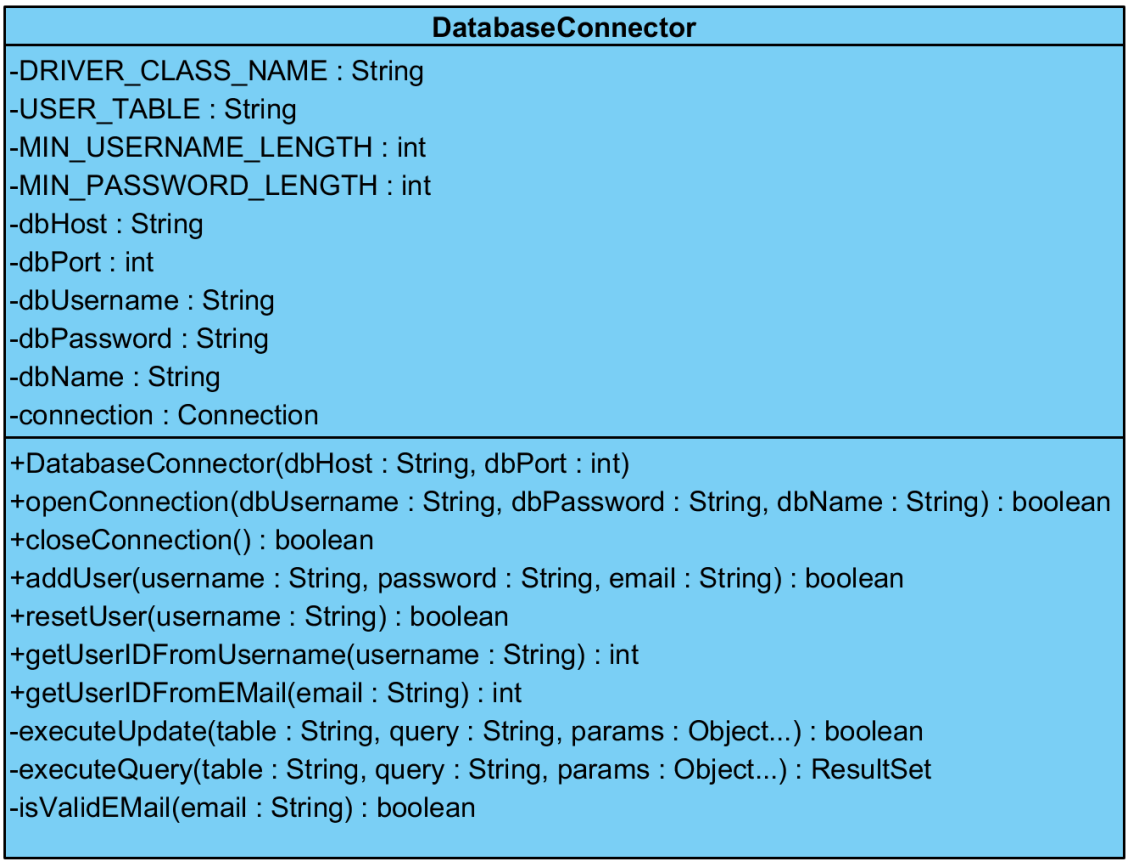


Figure 8: *DatabaseConnector Class*

DatabaseConnector:

Attributes:

private static final String DRIVER_CLASS_NAME = "com.mysql.cj.jdbc.Driver" : Holds the database connector driver class name.

private static final String USER_TABLE = "users" : Holds the name of users table on the database.

private static final int MIN_USERNAME_LENGTH = 4 : Holds the minimum length for the username of a new record.

private static final int MIN_PASSWORD_LENGTH = 8 : Holds the minimum length for the password of a new record.

private String dbHost : Holds the hostname for the database connection.

private int dbPort : Holds the connection port for the database connection.

private String dbUsername : Holds the username for the database connection.

private String dbPassword : Holds the password for the database connection.

private String dbName : Holds the database name for the database connection.

Constructors

DatabaseConnector(String dbHost, int dbPort)
Constructor for DatabaseConnector Class.
dbHost : The hostname for the Database Connection.
dbPort : The port for the Database Connection.

Methods:

boolean openConnection(String dbUsername, String dbPassword, String dbName)
This method opens the database connection with the required credentials.
dbUsername : The username for the database connection.
dbPassword : The password for the database connection.
dbName : The database name for the database connection.
Returns True if the connection is successful, otherwise is returns False.

boolean closeConnection()
This method closes the database connection.
Returns True if the connection is closed successfully, otherwise is returns False.

boolean addUser(String username, String password, String email)
This method adds a new user to the database.
username : The username for the new record.
password : The password for the new record.
email : The e-mail for the new record.
Returns True if the operation is completed successfully, otherwise is returns False.

boolean resetUser(String username)
This method resets the progress of the user.
username : The username of the user.
Returns True if the operation is completed successfully, otherwise is returns False.

int getUserIDFromUsername(String username)
This method finds the SQL ID of the user from its username.
username : The username of the user.
Returns the SQL ID of the user if the operation is completed successfully, otherwise is returns -1.

int getUserIDFromEMail(String email)
This method finds the SQL ID of the user from its e-mail.
email : The username of the user.
Returns the SQL ID of the user if the operation is completed successfully, otherwise is returns -1.

private boolean executeUpdate(String table, String query, Object... params)
This method queries manipulating queries on the desired table.
table : The table name which the operation runs on.
query : The SQL query.
params : The constraint parameters for the SQL query.
Returns True if the operation is completed successfully, otherwise is returns False.

private ResultSet executeQuery(String table, String query, Object... params)
This method queries selecting queries on the desired table.
table : The table name which the operation runs on.
query : The SQL query.
params : The constraint parameters for the SQL query.
Returns a ResultSet object with the desired data in it if the operation completed successfully, otherwise it returns null.

private boolean isValidEMail(String email)
This method checks if an e-mail address is valid.
email : The e-mail address.
Returns True if the e-mail address is valid, otherwise it returns False.

3.1.5 ClientSocketHandler

This class is used to handle socket connections.

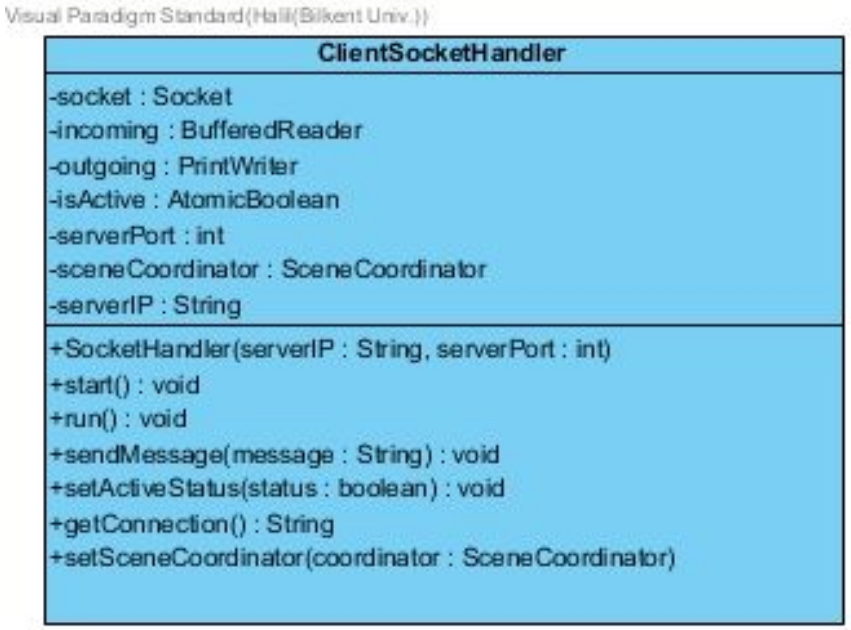


Figure 9: *ClientSocketHandler Class*

ClientSocketHandler:

Attributes:

private String serverIP : Holds the server IP.

private int serverPort : Holds the server port.

private Socket socket : Holds the instance of socket object of the client.

private BufferedReader incoming : Holds the incoming stream from the client.

private PrintWriter outgoing : Holds the outcoming stream to the client.

private AtomicBoolean isActive : Holds the status of the socket handler.

private SceneCoordinator sceneCoordinator : Holds scene coordinator instance.

Constructors:

ClientSocketHandler(String serverIP, int serverPort): Constructor for SocketHandler Class.

Methods:

public synchronized void start(): This method starts the thread of ClientSocketHandler.

public void run(): This method handles the incoming messages over socket connection.

void sendMessage(String message)
This method sends message to the client over the socket connection.
message : The string message.

void setActiveStatus(boolean status)
This method sets the active status of the ServerSocketHandler.
status : The active status.

void setSceneCoordinator(SceneCoordinator coordinator)
This method sets the scene coordinator for socket handler.
coordinator : SceneCoordinator instance.

3.1.6 SceneController

SceneController class allows transitioning between two scenes and transferring data between scenes and the server..
Therefore, it acts as a bridge between server updates and the game screens.

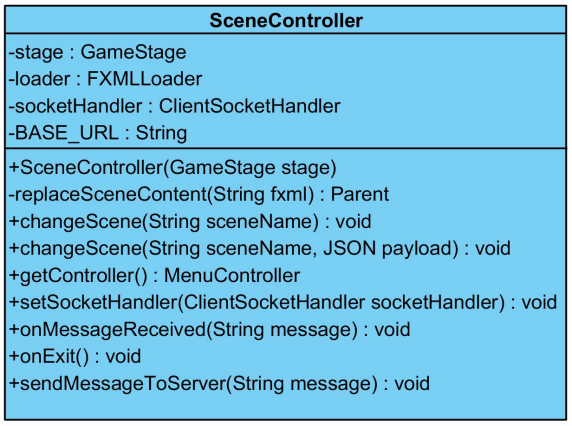


Figure 10: SceneController Class

SceneController:

Attributes:

private Stage stage : This attribute holds the window displayed which is game is played on.

private FXMLLoader loader: This attribute allows loading the FXML files to the current scene.

private ClientSocketHandler socketHandler: This attribute allows the communication between the server and the current scene.

private final String BASE_URL: This attribute holds the base file path for the FXML files.

Constructors:

public SceneController(Stage stage): Constructor for SceneController.

Methods:

private Parent replaceSceneContent(String fxml): Loads the FXML file into the loader.

public void changeScene(String sceneName): This method changes to scene to a new scene.

public void changeScene(String sceneName, JSONObject payload): This method changes to scene to a new scene and transfers data to the new scene.

public MenuController getController(): This method returns the controller of the current scene.

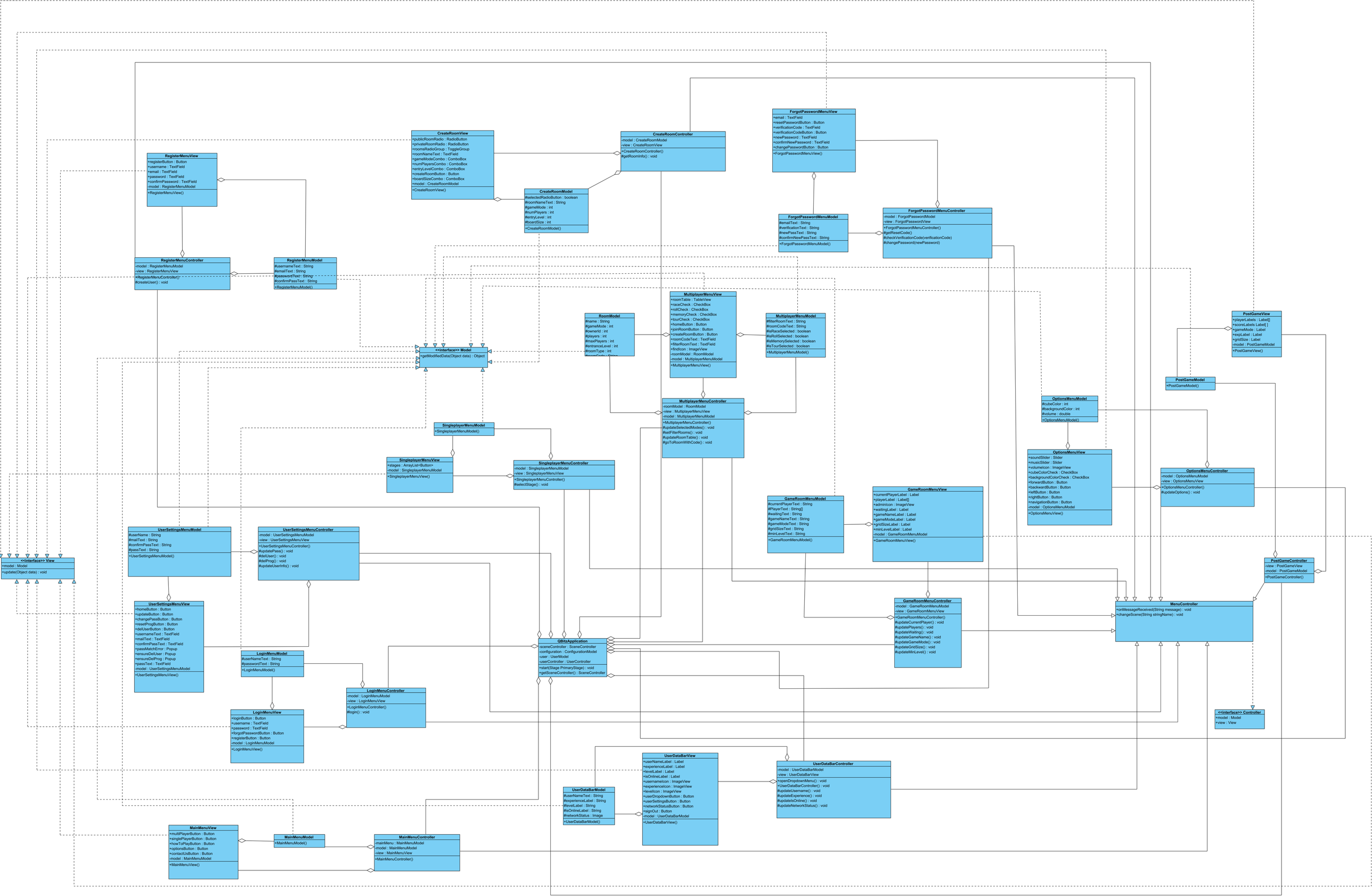
public void onMessageReceived(String message): This method is a callback method which is called when a client sent a message through the socket connection.

public void setSocketHandler(ClientSocketHandler socketHandler): This method sets the socketHandler.

public void onExit(): This method is a callback method which is called when a client disconnected.

public void sendMessage(String message): This method sends message to the client over the socket connection.

3.2 Menu Subsystem


CreateRoomView
+publicRoomRadio : RadioButton
+privateRoomRadio : RadioButton
+roomsRadioGroup : ToggleGroup
+roomNameText : TextField
+gameModeCombo : ComboBox
+numPlayersCombo : ComboBox
+entryLevelCombo : ComboBox
+createRoomButton : Button
+boardSizeCombo : ComboBox
+model : CreateRoomModel
+CreateRoomView()
CreateRoomController
-model : CreateRoomModel
-view : CreateRoomView
+CreateRoomController()
#getRoomInfo() : void
CreateRoomModel
#selectedRadioButton : boolean
#roomNameText : String
#gameMode : int
#numPlayers : int
#entryLevel : int
#boardSize : int
+CreateRoomModel()
ForgotPasswordMenuView
+email : TextField
+resetPasswordButton : Button
+verificationCode : TextField
+verificationCodeButton : Button
+newPassword : TextField
+confirmNewPassword : TextField
+changePasswordButton : Button
+ForgotPasswordMenuView()
ForgotPasswordMenuModel
#emailText : String
#verificationText : String
#newPassText : String
#confirmNewPassText : String
+ForgotPasswordMenuModel()
ForgotPasswordMenuController
-model : ForgotPasswordModel
-view : ForgotPasswordView
+ForgotPasswordMenuController()
#getResetCode()
#checkVerificationCode(verificationCode)
#changePassword(newPassword)
RegisterMenuView
+registerButton : Button
+username : TextField
+email : TextField
+password : TextField
+confirmPassword : TextField
-model : RegisterMenuModel
+RegisterMenuView()
RegisterMenuController
-model : RegisterMenuModel
-view : RegisterMenuView
+RegisterMenuController()
#createUser() : void
RegisterMenuModel
#usernameText : String
#emailText : String
#passwordText : String
#confirmPassText : String
+RegisterMenuModel()
<<interface>> Model
+getModifiedData(Object data) : Object
RoomModel
#name : String
#gameMode : int
#ownerId : int
#players : int
#maxPlayers : int
#entranceLevel : int
#roomType : int
MultiplayerMenuView
+roomTable : TableView
+raceCheck : CheckBox
+rollCheck : CheckBox
+memoryCheck : CheckBox
+tourCheck : CheckBox
+homeButton : Button
+joinRoomButton : Button
+createRoomButton : Button
+roomCodeText : TextField
+filterRoomText : TextField
+findIcon : ImageView
-roomModel : RoomModel
-model : MultiplayerMenuModel
+MultiplayerMenuView()
MultiplayerMenuModel
#filterRoomText : String
#roomCodeText : String
#isRaceSelected : boolean
#isRollSelected : boolean
#isMemorySelected : boolean
#isTourSelected : boolean
+MultiplayerMenuModel()
MultiplayerMenuController
-roomModel : RoomModel
-view : MultiplayerMenuView
-model : MultiplayerMenuModel
+MultiplayerMenuController()
#updateSelectedModes() : void
#setFilterRooms() : void
#updateRoomTable() : void
#goToRoomWithCode() : void
PostGameView
+playerLabels : Label[]
+scoreLabels Label[ ]
+gameMode : Label
+expLabel : Label
+gridSize : Label
-model : PostGameModel
+PostGameView()
PostGameModel
+PostGameModel()
PostGameController
-view : PostGameView
-model : PostGameModel
+PostGameController()
OptionsMenuModel
#cubeColor : int
#backgroundColor : int
#volume : double
+OptionsMenuModel()
OptionsMenuView
+soundSlider : Slider
+musicSlider : Slider
+volumeIcon : ImageView
+cubeColorCheck : CheckBox
+backgroundColorCheck : CheckBox
+forwardButton : Button
+backwardButton : Button
+leftButton : Button
+rightButton : Button
+navigationButton : Button
-model : OptionsMenuModel
+OptionsMenuView()
OptionsMenuController
-model : OptionsMenuModel
-view : OptionsMenuView
+OptionsMenuController()
#updateOptions() : void
SingleplayerMenuModel
+SingleplayerMenuModel()
SingleplayerMenuView
+stages : ArrayList<Button>
-model : SingleplayerMenuModel
+SingleplayerMenuView()
SingleplayerMenuController
-model : SingleplayerMenuModel
-view : SingleplayerMenuView
+SingleplayerMenuController()
#selectStage() : void
GameRoomMenuModel
#currentPlayerText : String
#PlayerText : String[]
#waitingText : String
#gameNameText : String
#gameModeText : String
#gridSizeText : String
#minLevelText : String
+GameRoomMenuModel()
GameRoomMenuView
+currentPlayerLabel : Label
+playerLabel : Label[]
+adminIcon : ImageView
+waitingLabel : Label
+gameNameLabel : Label
+gameModeLabel : Label
+gridSizeLabel : Label
+minLevelLabel : Label
-model : GameRoomMenuModel
+GameRoomMenuView()
GameRoomMenuController
-model : GameRoomMenuModel
-view : GameRoomMenuView
+GameRoomMenuController()
#updateCurrentPlayer() : void
#updatePlayers() : void
#updateWaiting() : void
#updateGameName() : void
#updateGameMode() : void
#updateGridSize() : void
#updateMinLevel() : void
UserSettingsMenuModel
#userName : String
#mailText : String
#confirmPassText : String
#passText : String
+UserSettingsMenuModel()
UserSettingsMenuController
-model : UserSettingsMenuModel
-view : UserSettingsMenuView
+UserSettingsMenuController()
#updatePass() : void
#delUser() : void
#delProg() : void
#updateUserInfo() : void
UserSettingsMenuView
+homeButton : Button
+updateButton : Button
+changePassButton : Button
+resetProgButton : Button
+delUserButton : Button
+usernameText : TextField
+mailText : TextField
+confirmPassText : TextField
+passMatchError : Popup
+ensureDelUser : Popup
+ensureDelProg : Popup
+passText : TextField
-model : UserSettingsMenuModel
+UserSettingsMenuView()
LoginMenuModel
#userNameText : String
#passwordText : String
+LoginMenuModel()
LoginMenuController
-model : LoginMenuModel
-view : LoginMenuView
+LoginMenuController()
#login() : void
LoginMenuView
+loginButton : Button
+username : TextField
+password : TextField
+forgotPasswordButton : Button
+registerButton : Button
-model : LoginMenuModel
+LoginMenuView()
QBlitzApplication
-sceneController : SceneController
-configuration : ConfigurationModel
-user : UserModel
-userController : UserController
+start(Stage PrimaryStage) : void
+getSceneController() : SceneController
MenuController
+onMessageReceived(String message) : void
+changeScene(String stringName) : void
<<interface>> Controller
+model : Model
+view : View
<<interface>> View
+model : Model
+update(Object data) : void
UserDataBarModel
#userNameText : String
#experienceLabel : String
#levelLabel : String
#isOnlineLabel : String
#networkStatus : Image
+UserDataBarModel()
UserDataBarView
+userNameLabel : Label
+experienceLabel : Label
+levelLabel : Label
+isOnlineLabel : Label
+usernameIcon : ImageView
+experienceIcon : ImageView
+levelIcon : ImageView
+userDropdownButton : Button
+userSettingsButton : Button
+networkStatusButton : Button
+signOut : Button
-model : UserDataBarModel
+UserDataBarView()
UserDataBarController
-model : UserDataBarModel
-view : UserDataBarView
+openDropdownMenu() : void
+UserDataBarController() : void
#updateUsername() : void
#updateExperience() : void
#updateIsOnline() : void
#updateNetworkStatus() : void
MainMenuModel
+MainMenuModel()
MainMenuController
-mainMenu : MainMenuModel
-model : MainMenuModel
-view : MainMenuView
+MainMenuController()
MainMenuView
+multiPlayerButton : Button
+singlePlayerButton : Button
+howToPlayButton : Button
+optionsButton : Button
+contactUsButton : Button
-model : MainMenuModel
+MainMenuView()

3.2.1 User Settings Menu

This class is used to handle user settings menu and send to changes about user settings to Menu Class.

UserSettingsMenuView:

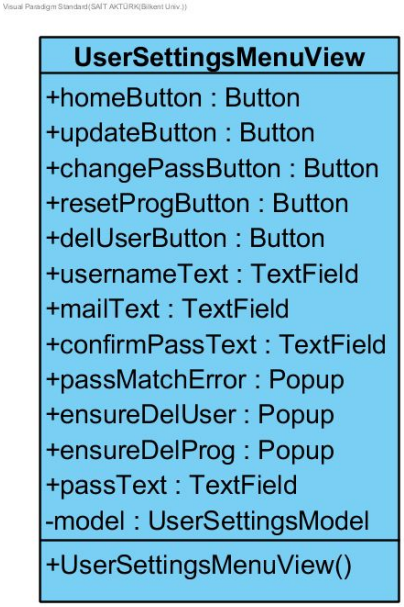


Figure 11: UserSettingsMenuView Class

Attributes:

public Button homeButton: This menu item will enable to endFrame() function, when this item is clicked.

public Button updateButton: This menu item will update information about valid changed username and mail. This item will activate updateUserInfo(String[]) to send updated info.

public Button changePassButton: This menu item will update user's password about valid new password. This item will activate updatePass(String) to send new password.

public Button resetProgButton: This menu item will activate delProg(), when this button is clicked. Also, a pop-up will activate to ensure about deletion.

public Button delUserButton: This menu item will activate delUser(), when this button is clicked. Also, a pop-up will activate to ensure about deletion.

public TextField usernameText: This menu item will get new valid username.

public TextField mailText: This menu item will get new valid mail.

public TextField confirmPassText: This menu item will get new password with second time to ensure user to set new password.

public Popup passMatchError: If user's new passwords does not match then menu initialize this popup screen to give information about unmatched error.

public Popup ensureDelUser: This menu will be appear, when user wants to delete his/her account to alert user.

public Popup ensureDelProg: This menu will be appear, when user wants to delete his/her progress to alert user.

private UserSettingsModel model: This is data model of the UserSettings menu.

Constructor:

public UserSettingsMenuView(): This is the constructor of userSettingsMenu, it instantiates the view components.

UserSettingsMenuModel:

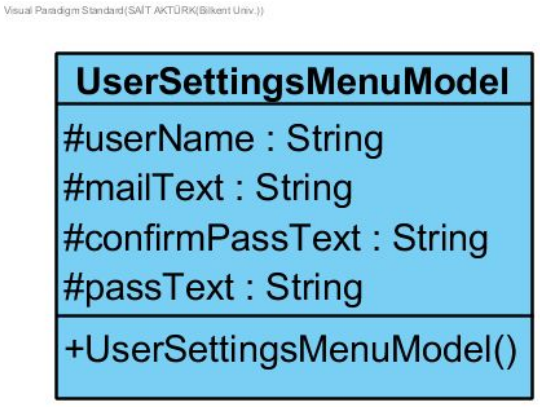


Figure 12: UserSettingsMenuModel Class

Attributes:

protected String userName: Holds the username of the user.

protected String mailText: Holds the email address of the user.

protected String confirmPassText: New password of the user for confirming password change.

protected String passText: New password of the user for password change.

Constructor:

public UserSettingsMenuModel(): This is the constructor of userSettingsMenu, it instantiates the view's data components.

UserSettingsMenuController:

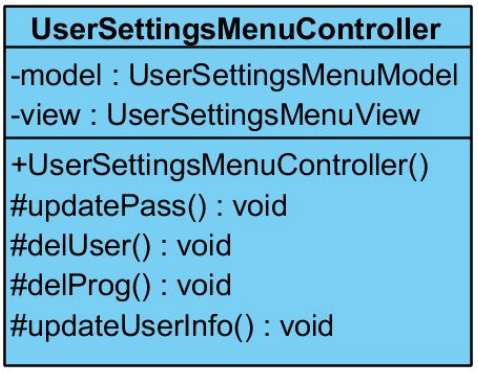


Figure 13: UserSettingsMenuController Class

Attributes:

private UserSettingsMenuModel model: Data model of the user settings menu.

private UserSettingsMenuView view: View of the user settings menu.

Constructor:

public UserSettingsMenuController(): This is the constructor of UserSettingsMenuController, it instantiates the controller class.

Methods:

protected void updatePass(): This function updates user's password.

protected void updateUserInfo(): This function updates username or mail or both.

protected void delUser(): This function deletes user's account.

protected void delProg(): This function deletes user's progress.

3.2.2 Options Menu

This class is used to handle options menu and send to changes about options to Menu Class.

OptionsMenuModel:

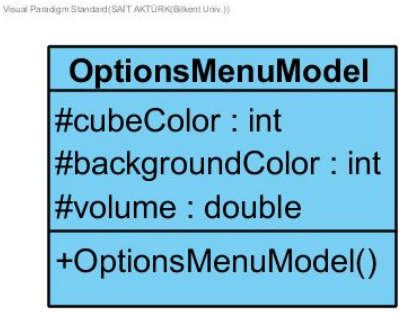


Figure 14: OptionsMenuModel Class

Attributes :

protected int cubeColor: This attribute represents the cube color as an integer based on predefined colors.

protected int backgroundColor: This attribute represents the background color of the menus as an integer based on predefined colors.

protected double volume: This attribute represents the volume level of the game.

Constructors:

public OptionsMenuModel(): This is the constructor of OptionsMenuModel, it instantiates the view's data components.

OptionsMenuView:

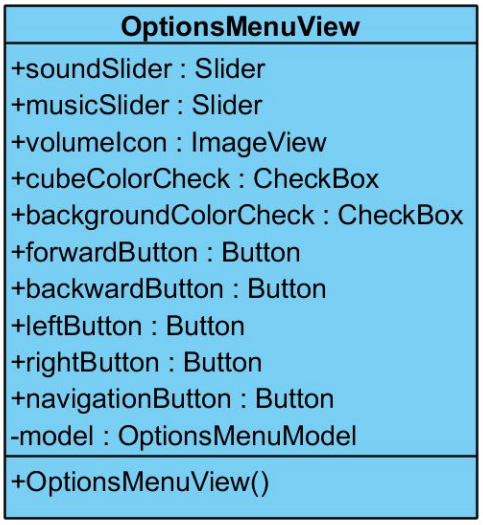


Figure 15: OptionsMenuView Class

Attributes:

private Slider soundSlider : This menu item will control the volume of sounds in the game. User could adjust to sound volume between 0 and 100.

private Slider musicSlider : This menu item will control the volume of musics in the game. User could adjust to music volume between 0 and 100.

private ImageView volumeIcon : This menu item will show volume of sounds and musics. There will be two different volumeIcon that one of the icons will belong sounds of the game and other icon will belong musics of the game. User could pressed these icons to mute sounds or musics. The user could also press these icons to turn on volume of musics and sounds.

private CheckBox cubeColorCheck: This menu item will help user to select color of cubes that user wants to change.

private CheckBox backgroundColorCheck: This menu item will help user to select color of background of the game that user wants to change.

private Button forwardButton: This menu item will help user to determine keyboard key that responsible forward turn of components of cube.

private Button forwardButton: This menu item will help user to determine keyboard key that responsible forward turn of components of cube.

private Button backwardButton: This menu item will help user to determine keyboard key that responsible backward turn of components of cube.

private Button leftButton: This menu item will help user to determine keyboard key that responsible left turn of components of cube.

private Button rightButton: This menu item will help user to determine keyboard key that responsible right turn of components of cube.

private Button navigationButton: This menu item will help user to determine mouse button that responsible selection of grid in board.

private OptionsMenuModel model: This is data model of the options menu.

Constructors:

public OptionsMenuView(): This is the constructor of optionsMenuView, it instantiates the view components.

OptionsMenuController:

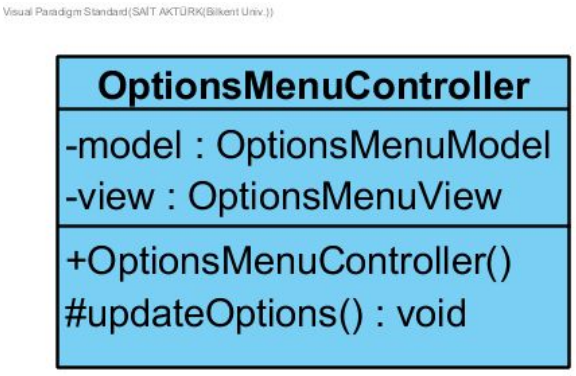


Figure 16 : OptionsMenuController Class

Attributes :

private OptionsMenuModel model: Data model of the options menu.

private OptionsMenuView view: View of the options menu.

Constructors:

public OptionsMenuController(): This is the constructor of OptionsMenuController, it instantiates the controller class.

Methods:

protected void updateOptions(): This function will update any changes related to OptionsMenuView.

3.2.3 **Multiplayer Mode Menu**

This class is used to handle multiplayer menu and send to changes about options to Menu Class.

MultiplayerMenuModel:

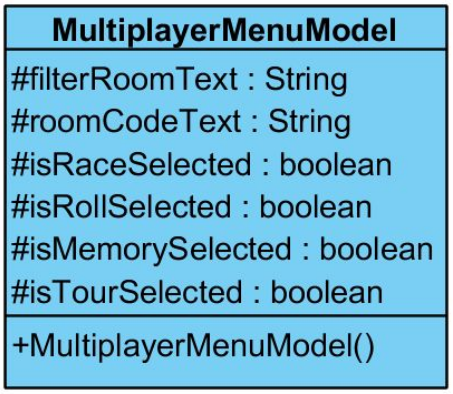


Figure 17 : MultiplayerMenuModel Class

Attributes:

protected String filterRoomText: Search text for the room list. Filters room based on the text.

protected String roomCodeText: Input field data for the room code.

protected boolean isRaceSelected: Data of the checkbox for race mode filter.

protected boolean isRollSelected: Data of the checkbox for roll to win mode filter.

protected boolean isTourSelected: Data of the checkbox for tournament mode filter.

Constructors:

public MulltiplayerMenuModel(): This is the constructor of MulltiplayerMenuModel, it instantiates the model class.

MultiplayerMenuView:

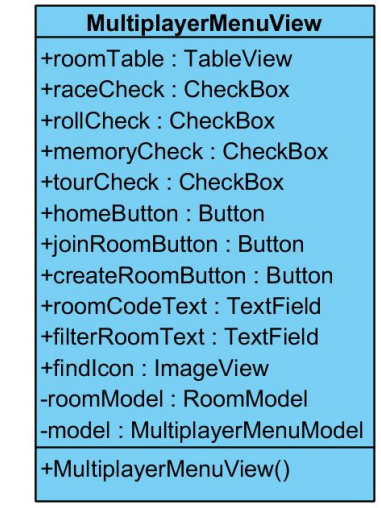


Figure 18 : MultiplayerMenuView Class

Attributes:

private RadioButton publicRoomRadio : This menu item will help user to select room type that whether user wants to create public room or not. When user selects this button, user could not select private room radio button.

private RadioButton privateRoomRadio : This menu item will help user to select room type that whether user wants to create private room or not. When user selects this button, user could not select public room radio button.

private ToggleGroup roomRadioGroup : This menu item will enforce two radio buttons that privateRoomRadio and publicRoomRadio to select only one button.

private TextField roomNameText : This menu item will specify room's name that will be created after all specifications filled. This items could not be blank text.

private ComboBox gameModeCombo : This menu item will specify room's mode that could be "Roll", "Memory", "Race" and "Tournament"..

private ComboBox numPlayersCombo : This menu item will specify number of players for room that will be created. This menu item has value between 2 and 8.

private ComboBox entryLevelCombo : This menu item will specify minimum level of attendants of rooms.This menu item could be at least level of creator of room.

Constructors:

public MulltiplayerMenuView(): This is the constructor of MulltiplayerMenuView, it instantiates the view components.

MultiplayerMenuController:

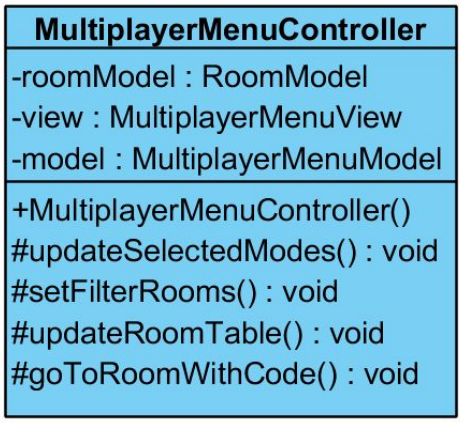


Figure 19 : MlutiplayerMenuController Class

Attributes:

private RoomModel roomModel:

private MultiplayerMenuView view: View of the MultiplayerMenu.

private MultiplayerMenuModel model: Data model for the MultiplayerMenu.

Constructors:

public MultiplayerMenuController() : This is the constructor of MultiplayerMenuController, it instantiates the controller class.

Methods:

protected void updateSelectedModes(): Update room list based on checkbox filters.

protected void setFilterRooms(): Update room list based on filter text.

protected void updateRoomTable(): Update the room list from the data from the server.

protected void goToRoomWithCode(): Join a room with a code.

3.2.4 Login Menu

LoginMenuModel:

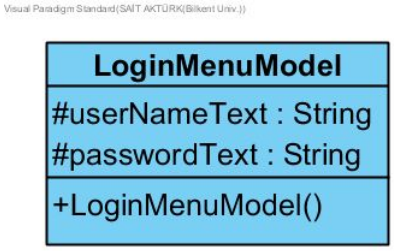


Figure 20 : LoginMenuModel

Attributes:

protected String userNameText: This attribute holds the data for username info of textfield.

protected String passwordText: This attribute holds the data for password info of textfield.

Constructors:

public LoginMenuModel(): This is the constructor of LoginMenuModel, it instantiates the model class.

LoginMenuView:

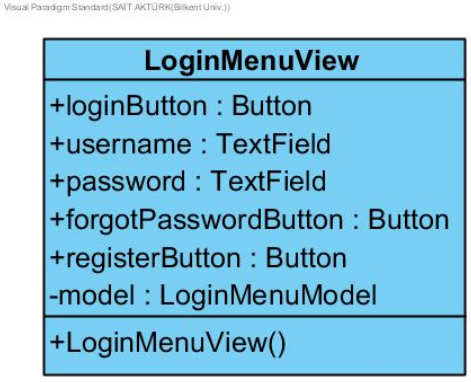


Figure 21 : LoginMenuView Class

Attributes:

private Button loginButton: This button triggers the login() method.

private TextField username: This text field holds username of the user.

private TextField password: This text field holds password of the user.

private Button forgotPasswordButton: This button directs the user to the forgot password menu.

private Button registerButton: This button directs the user to the register menu.

private LoginMenuModel model: This instance saves the data of LoginMenuView.

Constructors:

public LoginMenuView(): This is the constructor of LoginMenuView, it instantiates the view components.

LoginMenuController:

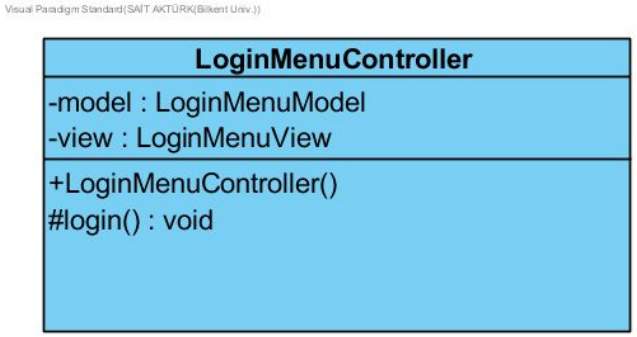


Figure 22 : LoginMenuController Class

Attributes:

private LoginMenuModel model: This instance is data model for LoginMenu.

private LoginMenuView view: This instance is view of LoginMenu.

Constructors:

public LoginMenuController: This is the constructor of LoginMenuControlller, it instantiates the controller components.

Methods:

protected boolean login(username, password): This method send the username and password to the server for logging in.

3.2.5 Register Menu

RegisterMenuModel:

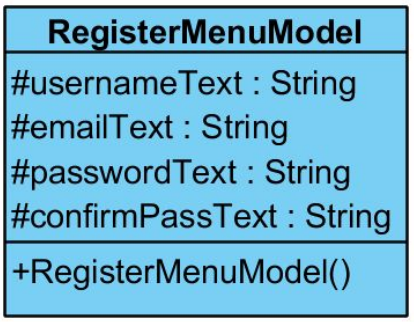


Figure 23 : RegisterMenuModel Class

Attributes:

protected String usernameText: Data of the username input field.

protected String emailText: Data of the email input field.

protected String passwordText: Data of the password input field.

protected String confirmPassText: Data of the confirm password input field.

RegisterMenuView:

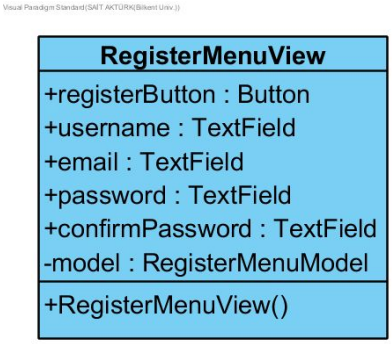


Figure 24 : *RegisterMenuView Class*

Attributes:

private Button registerButton: This button submits the registration form and triggers the createUser() method.

private TextField username: This text field holds username of the new account.

private TextField email: This text field holds email of the new account.

private TextField password: This text field holds password of the new account.

private TextField confirmPassword: This text field is for the user to re-enter their password to reduce the password errors.

Constructors:

public RegisterMenuView(): This is the constructor of RegisterMenuView, it instantiates the view components.

RegisterMenuController:

Figure 25 : *RegisterMenuController Class*

Attributes:

private RegisterMenuModel model: Data model for register menu.

private RegisterMenuView view: View of the register menu.

Methods:

protected boolean createUser(username, email, password): This method sends the account information in the registration form to the server.

3.2.6 Forgot Password Menu

ForgotPasswordMenuModel:

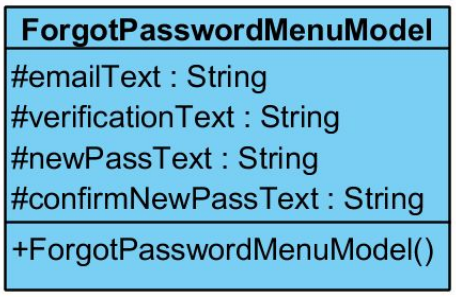


Figure 26 : *ForgotPasswordMenuModel Class*

Attributes:

protected String emailText: Data of the email input field.

protected String verificationText: Data of the verification code input field.

protected String newPassText: Data of the new password input field.

protected String confirmNewPassText: Data of the confirm password input field.

ForgotPasswordMenuView:

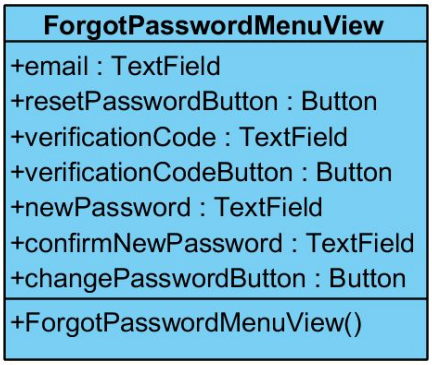


Figure 27 : *ForgotPasswordMenuView Class*

Attributes:

private TextField email: This text field is for user to enter their email for the verification code.

private Button resetPasswordButton: This button sends the email to the server.

private TextField verificationCode: This button is for the user to enter the verification code that they received in the mail.

private Button verifyCodeButton: This button sends the verification code the user entered to the server.

private TextField newPassword: This text field is for user to enter a new password for their account.

private TextField confirmNewPassword: This text field is for user to re-enter the new password for confirmation.

private Button changePasswordButton: This button sends the new password to the server.

Constructors:

public ForgotPasswordMenuView : Instantiates the view components.

ForgotPasswordMenuController:

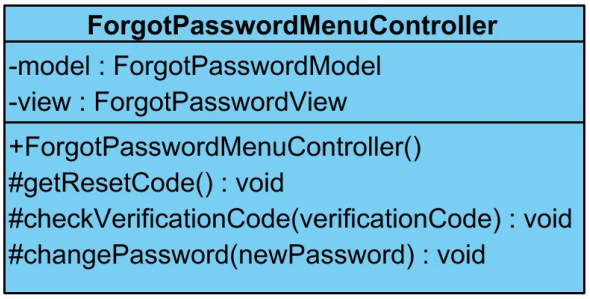


Figure 28 : *ForgotPasswordMenuController Class*

Attributes:

private ForgotPasswordModel model:

private ForgotPasswordView view:

Constructors:

public ForgotPasswordMenuController()

Methods:

protected boolean getResetCode(): This method is called by the resetPasswordButton and requests a verification code from the server.

protected boolean checkVerificationCode(verificationCode): This method is called to check if the entered verification code is correct.

protected boolean changePassword(newPassword): This method called by the changePasswordButton to send the new password to the server.

3.2.7 Main Menu

MainMenuModel:

Figure 29 : *MainMenuModel Class*

Constructors:

public MainMenuModel(): Instantiates data in the main menu model.

MainMenuView:

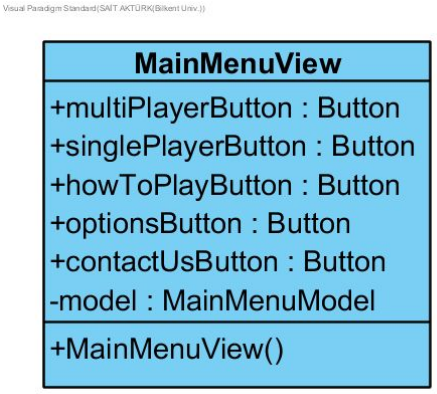


Figure 30 : *MainMenuView Class*

Attributes:

private Button multiplayerButton: This button directs the user to the multiplayer menu.

private Button singlePlayerButton: This button directs the user to the singleplayer menu.

private Button howToPlayButton: This button directs the user to the how to play menu.

private Button optionsButton: This button directs the user to the options menu.

private Button contactUsButton: This button creates a form to send a message to the developers.

MainMenuController:

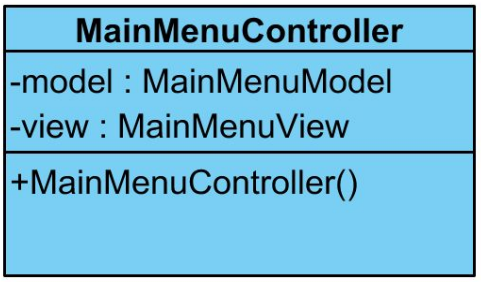


Figure 31 : *MainMenuController Class*

Attributes:

private MainMenuModel mainMenu:

private MainMenuModel model:

3.2.8 Singleplayer Menu

SingleplayerMenuModel:

Figure 32 : *SingleplayerMenuModel Class*

SingleplayerMenuView:

Figure 33 : *SingleplayerMenuView*

Attributes:

private ArrayList<Button> stages: This attribute contain the Stage objects to reach every stage in SingleplayerGame.

SingleplayerMenuController:

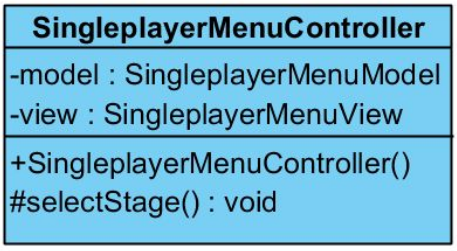


Figure 34 : *SingleplayerMenuController Class*

Attributes:

private SingleplayerMenuModel model : Data model for single player menu.

private SingleplayerMenuView view : View for single player menu.

Methods:

protected void selectStage(): This method loads the stage after selecting it on the menu.

3.2.9 UserDataBar

UserDataBarModel:

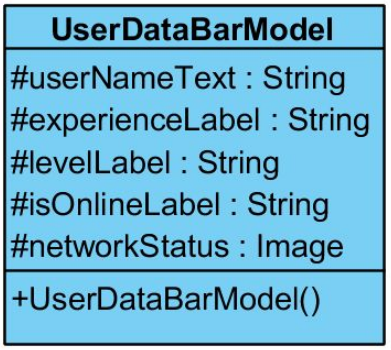


Figure 35 : *UserDataBarModel Class*

Attributes:

protected String userName: Data of the username input field.

String mailText: Data of the email text field.

String confirmPassText: Data of the username input field.

String passText: Data of the username input field.

Constructors:

public UserSettingsMenuModel()

UserDataBarView:

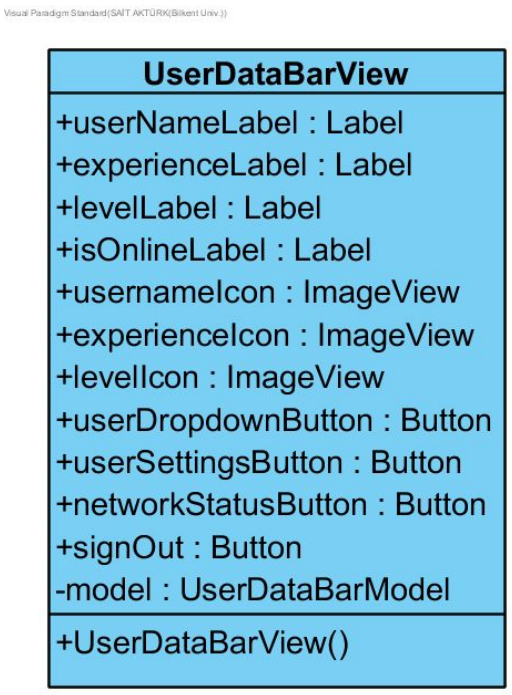


Figure 36 : *UserDataBarView Class*

Attributes:

private Label usernameLabel: This label shows the username of the user.

private Label experienceLabel: This label shows the current experience points of the user.

private Label levelLabel: This label shows the experience level of the user.

private Label isOnlineLabel: This label shows the network status of the user. (Online or offline)

private ImageView usernameIcon: This image is the username icon.

private ImageView experienceIcon: This image is the experience icon.

private ImageView levelIcon: This image is the level icon.

private Button userDropdownButton: This button displays the dropdown menu when clicked on the username icon.

private Button userSettingsButton: This button is in the dropdown menu and directs to the user settings menu.

private Button networkStatusButton: This button is in the dropdown menu and changes the network status of the user.

private Button signOut: This button is in the dropdown menu and logs out of the account of the user.

UserDataBarController:

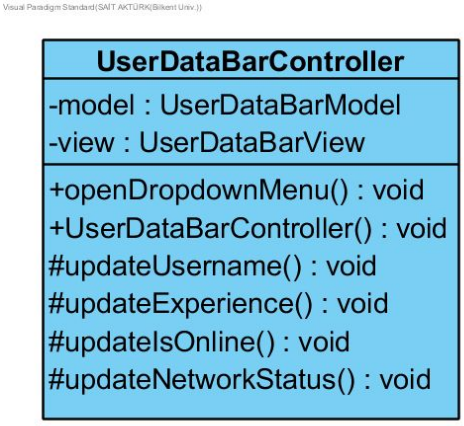


Figure 37 : *UserDataBarController Class*

Attributes:

private UserSettingMenuModel: Model object of UserDataBar.

private UserSettingMenuView: View object of UserDataBar.

Constructors:

public UserSettingsMenuController(): This constructor initializes model and view objects of the UserDataBar.

Methods:

protected void updateUsername(): Update the username in the databar.

protected void updateExperience(): Update the experience points in the databar.

protected void updateIsOnline(): Update the online status in the databar.

protected void updateNetworkStatus(): Update the network status in the databar.

3.2.10 PostGame

This is the scene for the post game screen. Finish time and ranking will be displayed here.

PostGameModel:

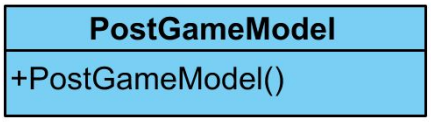


Figure 38 : *PostGameModel Class*

Constructors:

public PostGameModel():

PostGameController:

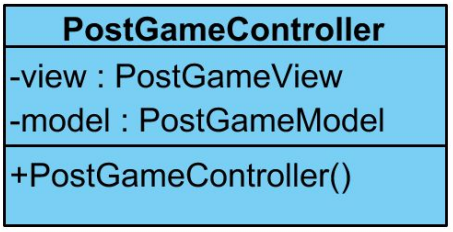


Figure 39 : *PostGameController Class*

PostGameView:

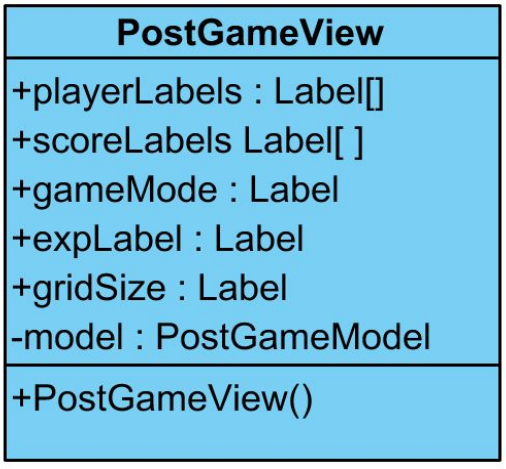


Figure 40 : *PostGameView Class*

Attributes:

public Label[ ] playerLabels: Shows the player names that are in the room

public Label[ ] scoreLabels: Labels of the scores of the players in the room.

public Label gameMode: Label text of the game mode of the room.

public Label expLabel : Label text of the experience gained.

public Label gridSize : Label text of the grid size.

private PostGameModel model: Data model of the post game menu.

Constructors:

public PostGameView():

3.2.11 GameRoomMenu

This is the scene for the game room. After joining a room, user can see the other players in the room, the game mode and the grid size of the board.

GameRoomMenuModel:

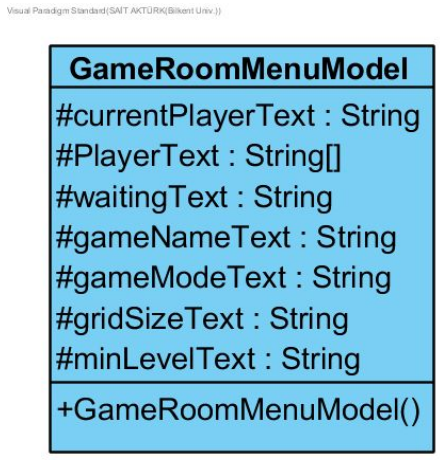


Figure 41 : *GameRoomMenuModel Class*

Attributes:

protected String currentPlayerText: Name of the current player.

protected String[] playerText: Names of the other players in the room.

protected String waitingText: Remaining number of players for game to start.

protected String gameNameText: Name of the game room.

protected String gameModeText: Name of game mode.

protected String gridSizeText: Grid size of the room.

protected String minLevelText: Minimum level to join the room.

GameRoomMenuController:

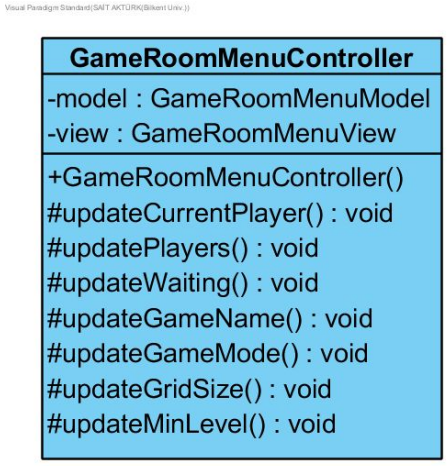


Figure 42 : *GameRoomMenuController Class*

Attributes:

private GameRoomMenuModel model:

private GameRoomMenuView view:

Constructors:

public GameRoomMenuController():

Methods:

These methods update the data model and the data on the server.

protected void updateCurrentPlayer()

protected void updatePlayers()

protected void updateWaiting()

protected void updateGameMode()

protected void updateGridSize()

protected void updateMinLevel()

GameRoomMenuView:

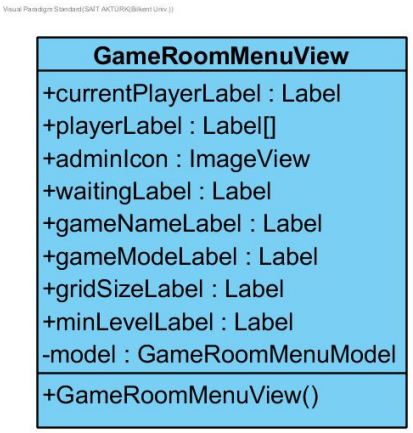


Figure 43 : *GameRoomMenuView Class*

Attributes:

public Label currentPlayerLabel: This attribute shows the player name that created game room

public Label [ ] playerLabel: This attribute shows player labels that currently join the room

public ImageView adminIcon: This attribute shows who is the creator of the game.

public Label waitingLabel: This attribute shows number of currently joined in the room

public Label gameNameLabel: This attribute shows room name.

public Label gameModeLabel: This attribute shows room mode.

public Label gridSizeLabel: This attribute shows game room's grid size.

public Label minLevelLabel: This attribute shows minimum level for game room.

private GameRoomMenuModel model:

3.2.12 CreateRoom

CreateRoomModel:

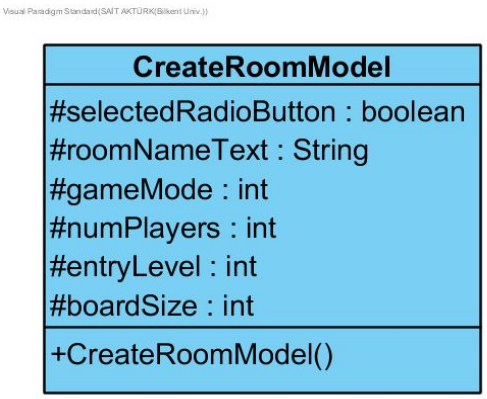


Figure 44 : *CreateRoomModel Class*

Attributes:

#protected boolean selectedRadiobutton: This attribute holds which radio button selected that private or public.

#protected String roomNameText: This attribute holds room name information.

#protected int gameMode : This attribute holds room game mode.

#protected int numPlayers : This attribute holds number of players in the room.

#protected int entryLevel : This attribute holds minimum level in the room.

#protected int boardSize : This attribute holds board size of the room.

Constructors:

public CreateRoomModel()

CreateRoomController:

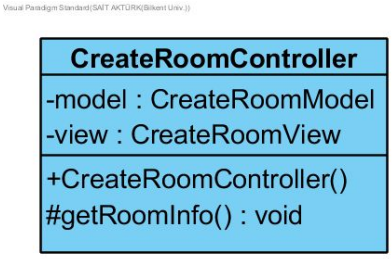


Figure 45 : *CreateRoomController Class*

Attributes:

private CreateRoomModel model:

private CreateRoomView view:

Constructors:

public CreateRoomController():

Methods:

protected void getRoomInfo(): This methods sends room info to server for room creation.

CreateRoomView:

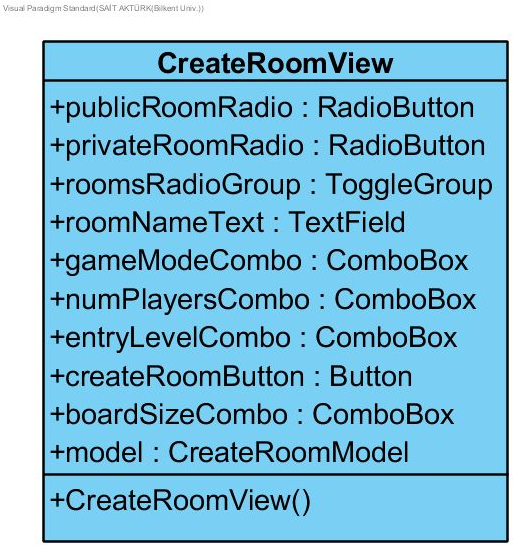


Figure 46 : *CreateRoomView Class*

Attributes:

View components for create room menu.

public RadioButton publicRoomRadio

public RadioButton privateRoomRadio

public ToggleGroup roomsRadioGroup

public TextField roomNameText

public ComboBox gameModeCombo

public ComboBox numPlayersCombo
public ComboBox entryLevelCombo
public Button createRoomButton
public ComboBox boardSizeCombo

public CreateRoomModel model: Data model for create room menu.

Constructors:

public CreateRoomView()

3.3 Game Logic Subsystem

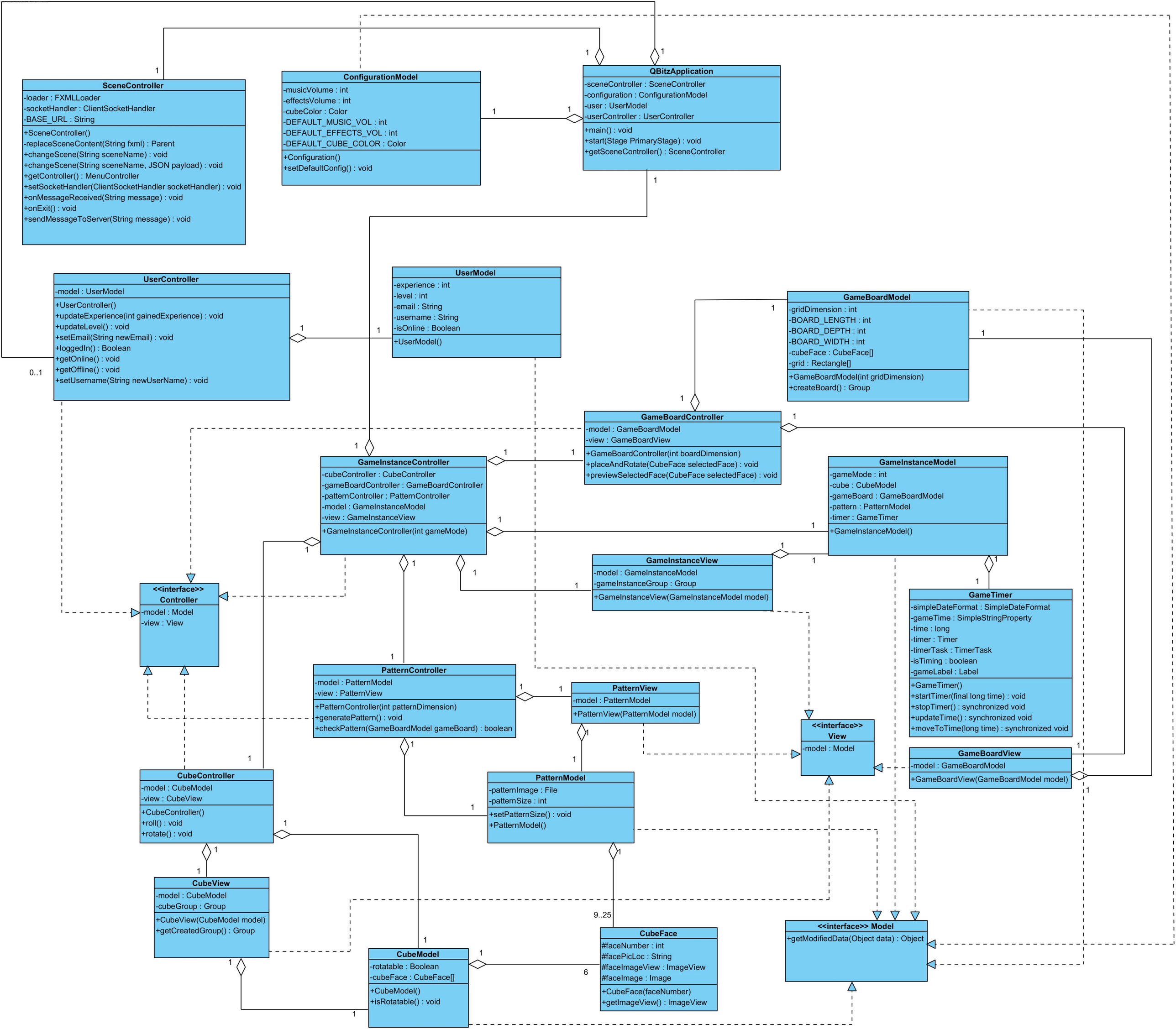

SceneController
-loader : FXMLLoader
-socketHandler : ClientSocketHandler
-BASE_URL : String
+SceneController()
-replaceSceneContent(String fxml) : Parent
+changeScene(String sceneName) : void
+changeScene(String sceneName, JSON payload) : void
+getController() : MenuController
+setSocketHandler(ClientSocketHandler socketHandler) : void
+onMessageReceived(String message) : void
+onExit() : void
+sendMessageToServer(String message) : void
ConfigurationModel
-musicVolume : int
-effectsVolume : int
-cubeColor : Color
-DEFAULT_MUSIC_VOL : int
-DEFAULT_EFFECTS_VOL : int
-DEFAULT_CUBE_COLOR : Color
+Configuration()
+setDefaultConfig() : void
QBitzApplication
-sceneController : SceneController
-configuration : ConfigurationModel
-user : UserModel
-userController : UserController
+main() : void
+start(Stage PrimaryStage) : void
+getSceneController() : SceneController
UserController
-model : UserModel
+UserController()
+updateExperience(int gainedExperience) : void
+updateLevel() : void
+setEmail(String newEmail) : void
+loggedIn() : Boolean
+getOnline() : void
+getOffline() : void
+setUsername(String newUserName) : void
UserModel
-experience : int
-level : int
-email : String
-username : String
-isOnline : Boolean
+UserModel()
GameBoardModel
-gridDimension : int
-BOARD_LENGTH : int
-BOARD_DEPTH : int
-BOARD_WIDTH : int
-cubeFace : CubeFace[]
-grid : Rectangle[]
+GameBoardModel(int gridDimension)
+createBoard() : Group
GameBoardController
-model : GameBoardModel
-view : GameBoardView
+GameBoardController(int boardDimension)
+placeAndRotate(CubeFace selectedFace) : void
+previewSelectedFace(CubeFace selectedFace) : void
GameInstanceController
-cubeController : CubeController
-gameBoardController : GameBoardController
-patternController : PatternController
-model : GameInstanceModel
-view : GameInstanceView
+GameInstanceController(int gameMode)
GameInstanceModel
-gameMode : int
-cube : CubeModel
-gameBoard : GameBoardModel
-pattern : PatternModel
-timer : GameTimer
+GameInstanceModel()
GameInstanceView
-model : GameInstanceModel
-gameInstanceGroup : Group
+GameInstanceView(GameInstanceModel model)
<<interface>>
Controller
-model : Model
-view : View
PatternController
-model : PatternModel
-view : PatternView
+PatternController(int patternDimension)
+generatePattern() : void
+checkPattern(GameBoardModel gameBoard) : boolean
PatternView
-model : PatternModel
+PatternView(PatternModel model)
GameTimer
-simpleDateFormat : SimpleDateFormat
-gameTime : SimpleStringProperty
-time : long
-timer : Timer
-timerTask : TimerTask
-isTiming : boolean
-gameLabel : Label
+GameTimer()
+startTimer(final long time) : void
+stopTimer() : synchronized void
+updateTime() : synchronized void
+moveToTime(long time) : synchronized void
<<interface>>
View
-model : Model
GameBoardView
-model : GameBoardModel
+GameBoardView(GameBoardModel model)
CubeController
-model : CubeModel
-view : CubeView
+CubeController()
+roll() : void
+rotate() : void
PatternModel
-patternImage : File
-patternSize : int
+setPatternSize() : void
+PatternModel()
CubeView
-model : CubeModel
-cubeGroup : Group
+CubeView(CubeModel model)
+getCreatedGroup() : Group
CubeModel
-rotatable : Boolean
-cubeFace : CubeFace[]
+CubeModel()
+isRotatable() : void
CubeFace
#faceNumber : int
#facePicLoc : String
#faceImageView : ImageView
#faceImage : Image
+CubeFace(faceNumber)
+getImageView() : ImageView
<<interface>> Model
+getModifiedData(Object data) : Object
1
0..1
6
9..25

Figure 47: GameLogic Class Diagram

3.3.1 GameInstanceController

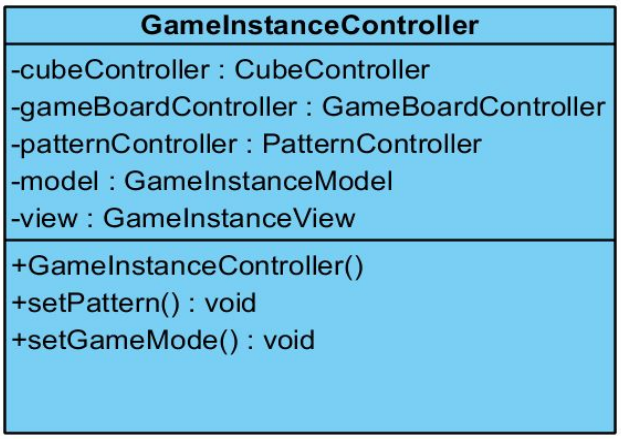


Figure 48 : *GameInstanceController Class*

GameInstanceController:

Attributes:

private CubeController cubeController: This attribute will be used to control the cube on the game instances to update and make it functional according to game instance controller.

private GameBoardController gameBoardController: This attribute will be used to control the game board on the game instances to update and make it functional according to game instance controller.

private PatternController patternController: This attribute will be used to control the pattern on the game instances to update and make it functional according to game instance controller.

private GameInstanceModel model: This attribute will be used to update data for and get data from GameInstanceModel class for controller.

private GameInstanceView view: This attribute will be used to send update signal to update itself according to controller actions.

Constructors:

public GameInstanceController(int gameMode) : This constructor will initialize the attributes to create a game instance with controls to change in it according to given game mode indicator through parameter.

3.3.2 GameInstanceModel

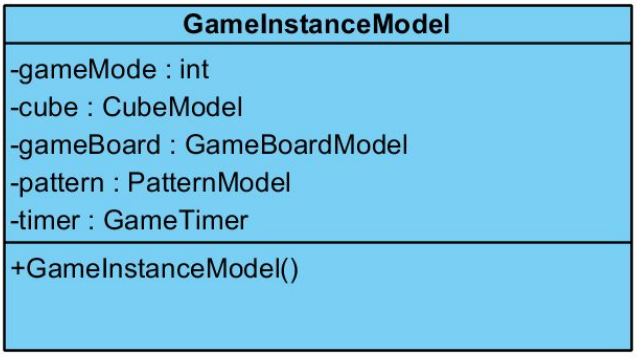


Figure 49 : *GameInstanceModel Class*

GameInstanceModel:

Attributes:

private int gameMode: Stores the game mode as an integer.

private CubeModel cube: 3D cube object for rotating and selecting faces.

private GameBoardModel gameBoard: Board object for the game board.

private PatternModel pattern: Pattern model to be completed in the game.

Constructors:

public GameInstanceModel(): Constructor for the Game Instance.

3.3.3 GameInstanceView

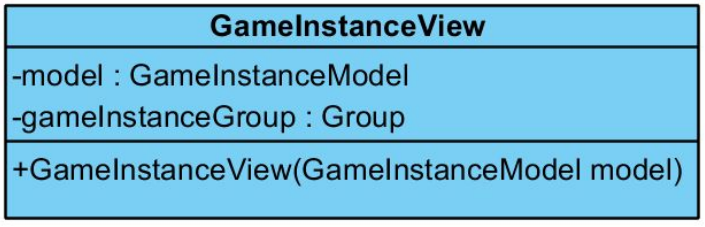


Figure 50 : *GameInstanceView Class*

GameInstanceView:

Attributes:

private GameInstanceModel model: stores the Game Instance model.

private Group gameInstanceGroup: JavaFX group object that stores the scene objects.

Constructors:

public GameInstanceView(GameInstanceModel model): Constructor for the view.

3.3.4 CubeController

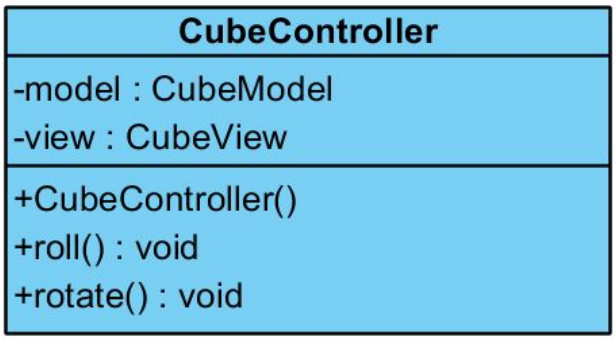


Figure 51 : *CubeController Class*

CubeController:

Attributes:

private CubeModel model:

private CubeView view:

Constructors:

public CubeController():

Methods:

public void roll(): This method gets random top face for roll-to-win mode.

public void rotate(): Rotates the cube.

3.3.5 CubeModel

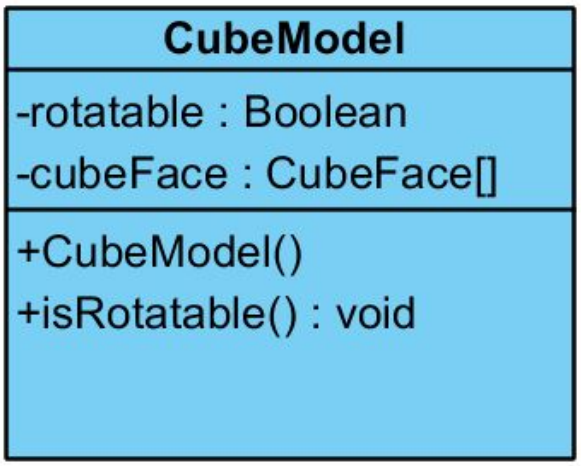


Figure 52 : *CubeModel Class*

CubeModel:

Attributes:

private boolean rotatable
private CubeFace[ ] cubeFace

Constructors:

public CubeModel()

Methods:

public void isRotatable()

3.3.6 CubeView

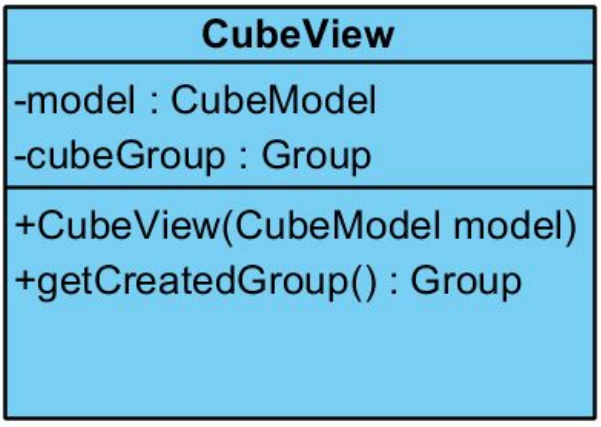


Figure 53 : *CubeView Class*

CubeView:

Attributes:

private CubeModel model
private Group cubeGroup

Constructors:

public CubeView(CubeModel model)

Methods:

private Group getCreatedGroup()

3.3.7 CubeFace

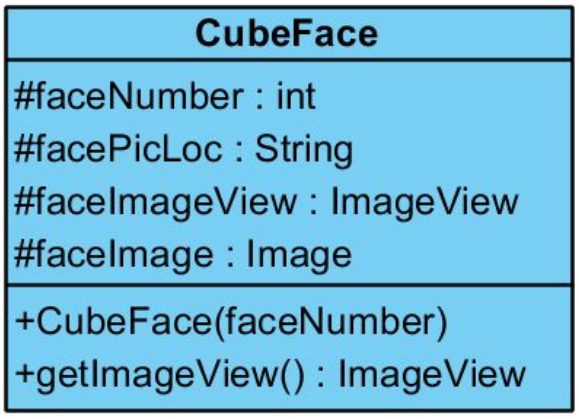


Figure 54 : *CubeFace Class*

CubeFace:

Attributes:

int faceNumber: Number for the picture that belongs to this face.

String facePicLoc: File location of the face picture.

ImageView faceImageView: Picture of the cube face as an ImageView object.

Image faceImage: Picture of the cube face as an Image object.

Constructors:

public CubeFace(): constructor for the CubeFace class.

Methods:

public ImageView getImageView() : Getter method for the cube face object to get a ImageView object as cube face picture

3.3.8 GameTimer

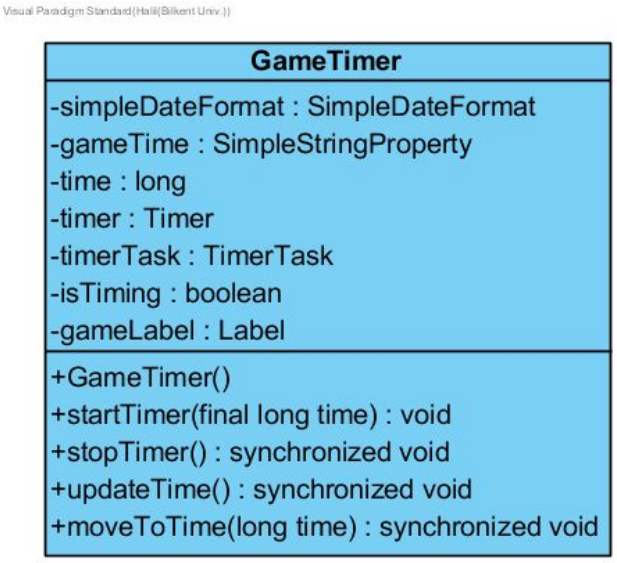


Figure 55 : *GameTimer Class*

GameTimer:

Attributes:

private SimpleDateFormat simpleDateFormat:
private SimpleStringProperty gameTime:
private long time:
private Timer timer:
private boolean isTiming:
private Label gameLabel: The label timer will be attached to.

Constructors:

public GameTimer():

Methods:

public void startTimer(final long time):
public void stopTimer():
public void updateTimer():
public void moveToTime(long time):

3.3.9 GameBoardModel

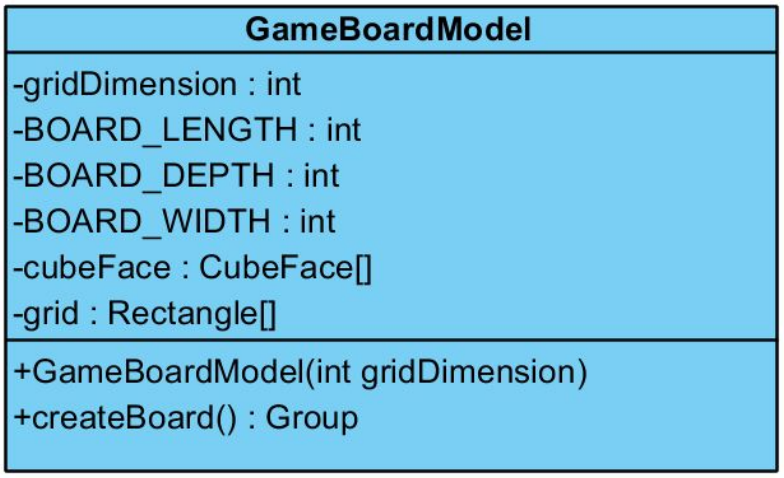


Figure 56: GameBoardModel Class

GameBoardModel:

Attributes:

private int gridDimension: Grid size of the board.

private final int BOARD_LENGTH: Length of the board in size.

private final int BOARD_DEPTH: Depth of the board in size.

private final int BOARD_WIDTH: Width of the board in size.
private CubeFace[] cubeFace
private Rectangle[] grid

Constructors:

public GameBoardModel(int gridDimension):

Methods:

public Group createBoard():

3.3.10 GameBoardController

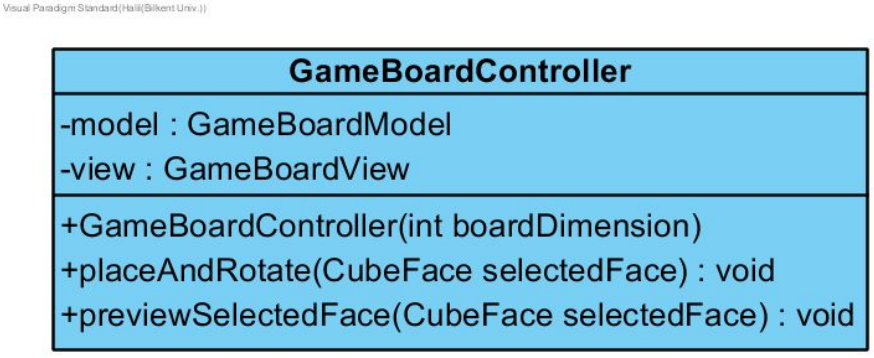


Figure 57 : *GameBoardController Class*

GameBoardController:

Attributes:

private GameBoardModel model
private GameBoardView view

Constructors:

public GameBoardController(int boardDimension)

Methods:

public void placeAndRotate(CubeFace selectedFace)
public void previewSelectedFace(CubeFace selectedFace)

3.3.11 GameBoardView

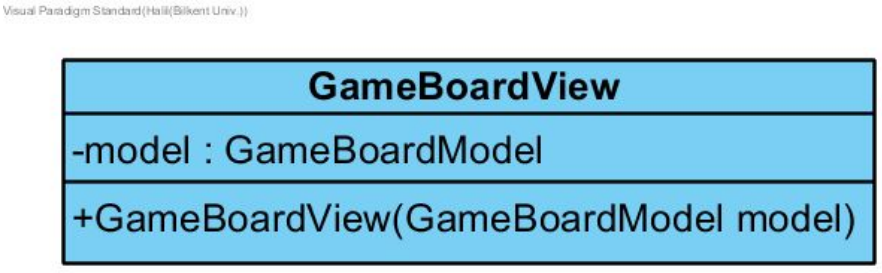


Figure 57 : *GameBoardView Class*

GameBoardView:

Attributes:

private GameBoardModel model

Constructors:

public GameBoardView(GameBoardModel model)

3.3.12 PatternController

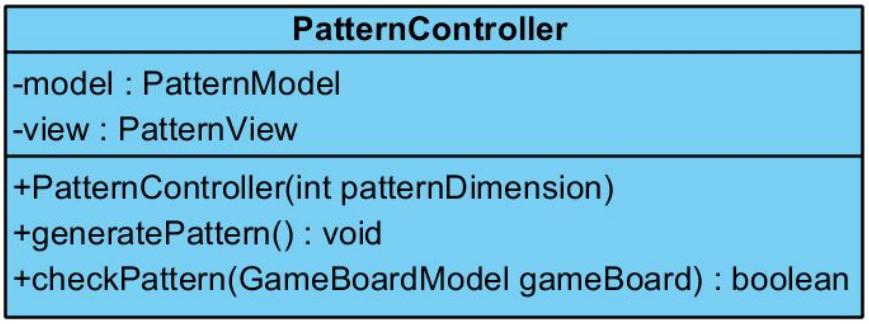


Figure 58: PatternController Class

PatternController:

Attributes:

private PatternModel model
private PatternView view

Constructors:

public PatternController(int patternDimension)

Methods:

public void generatePattern()
public boolean checkPattern(GameBoardModel gameBoard)

3.3.12 PatternModel

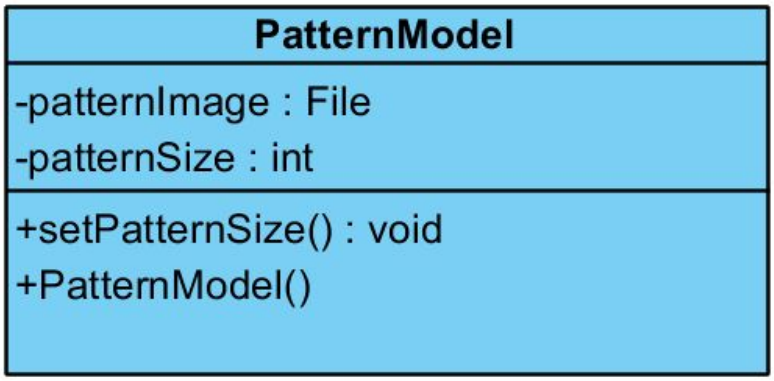


Figure 59: *PatternModel Class*

PatternModel:

Attributes:

private File patternImage: stores the pattern as an image file.

private int patternSize: stores the dimension of the grid.

Constructors:

public PatternModel(): constructor for the PatternModel class.

3.3.13 PatternView

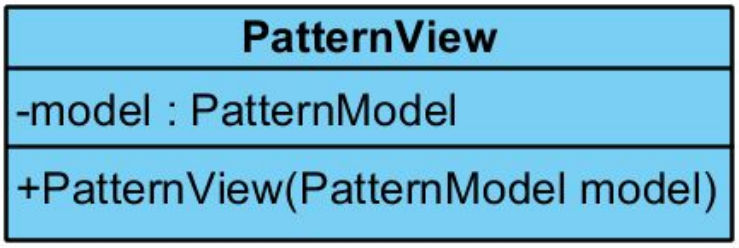


Figure 60: *PatternView Class*

PatternView:

Attributes:

private PatternModel model: stores the pattern model

Constructors:

public PatternView(PatternModel model): constructor for pattern view.

3.4 Core Subsystem

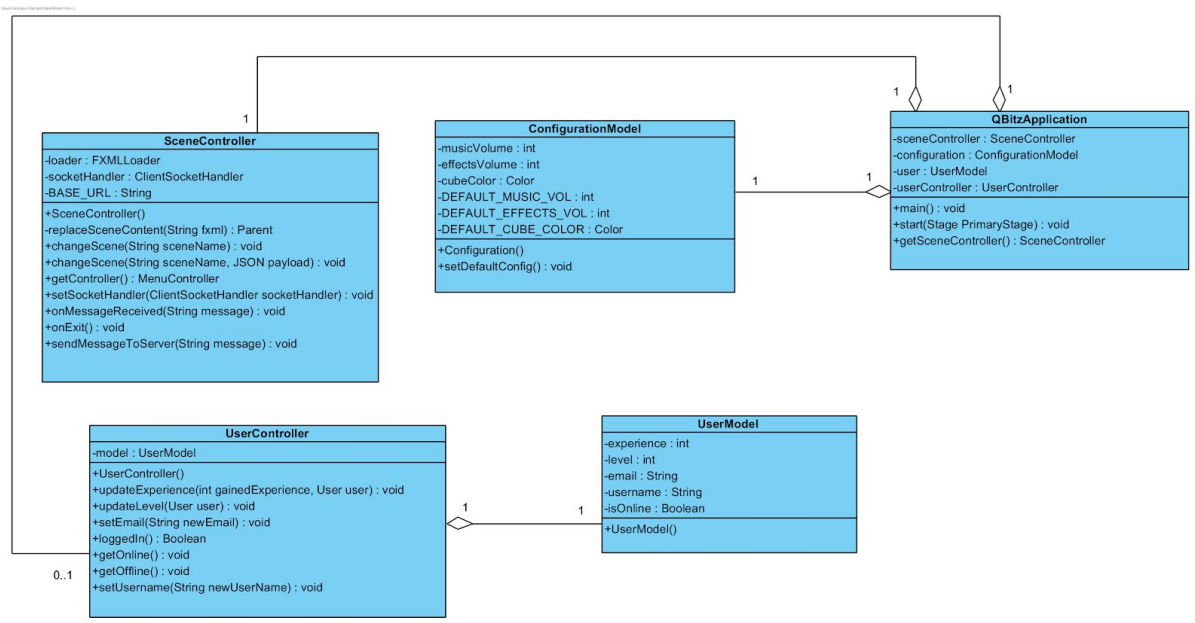


Figure 61: User Class Diagram

3.4.1 UserModel

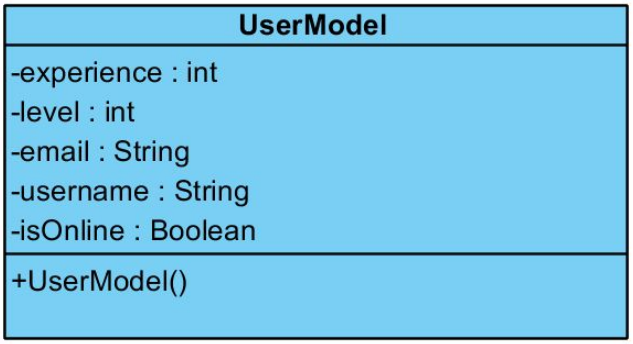


Figure 62: UserModel Class

User:

Attributes:

private int experience : This attribute keeps the experience of the user.

private int level : This attribute keeps the level of the user.

private String email : This attribute keeps the email of the user.

private String username : This attribute keeps the username of the user.

private boolean isOnline : This attribute keeps the state of the user whether it is online or not.

Constructors:

public UserModel() : Constructor of the user of the game to create an object that can keep the user information in it.

3.4.2 UserController

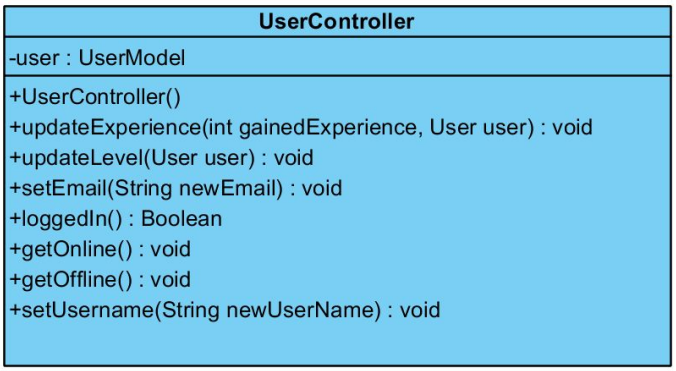


Figure 63: *UserController Class*

UserController

Attributes:

private UserModel model : This is an attribute to change and get the data in the model class through this controller class.

Constructors:

public UserController() : This constructor will initialize the model for the user and control it.

Methods:

public void updateExperience(int gainedExperience) : This function updates the experience of the user for the user with the gained experience.

public void updateLevel() : This function updates the level of the user according to its experience.

public boolean loggedIn() : This function checks the user is logged in or not.

public void getOnline() : This function connects the user to the server to make it online.

public void getOffline() : This function cuts the connection of the user to the server to make it offline in system.

public void setUserName(String newUserName) : A setter function to set the username of the user through user control function.

public void setEmail(String newEmail) : A setter function to set the email of the user through user control function.

3.4.3 ConfigurationModel

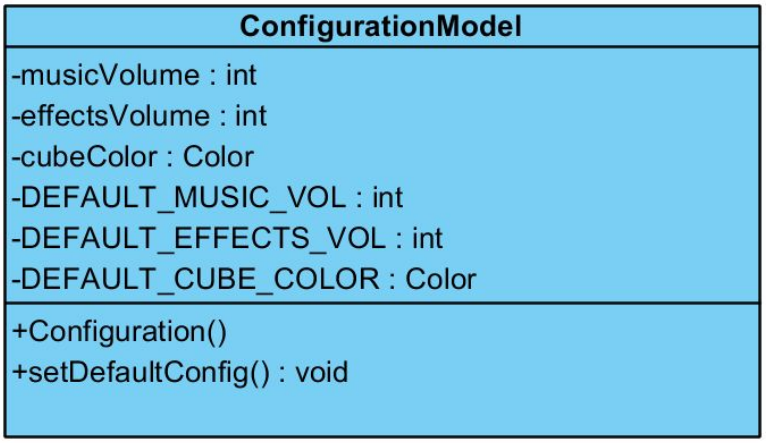


Figure 64*: Configuration Class*

Configuration:

Attributes:

private int musicVolume: This attribute will be used to store the level of music volume of the game.

private int effectsVolume: This attribute will be used to store the level of effects volume of the game.

private Color cubeColor: This attribute will be used to store the color of cube in game screens.

private final int DEFAULT_MUSIC_VOL: This attribute will be used to store the default level of music volume of the game.

private final int DEFAULT_EFFECTS_VOL: This attribute will be used to store the default level of effects volume of the game.

private final Color DEFAULT_CUBE_COLOR: This attribute will be used to store the default cube color of the game.

Constructors:

public Configuration(): Constructor of the configurations of the game to create an object that can modify and set default of configurations of the game.

Methods:

public static void setDefaultConfig(): A function to set the configuration to their default settings.

3.4.4 QBitzApplication

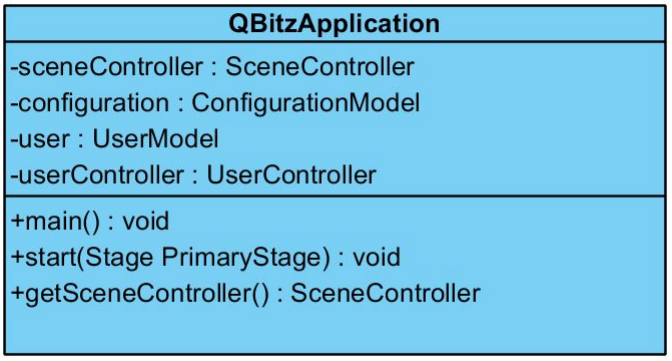


Figure 65*: QBitzApplication Class*

QBitzApplication:

Attributes:

private SceneController sceneController:
private ConfigurationModel configuration:
private UserModel user:
private UserController userController:

Methods:

public void main():
public void start(Stage PrimaryStage):
public SceneController getSceneController():

4.Low-level Design

4.1 Object design trade-offs

4.1.1 Usability vs. Functionality:

As we discussed before, our system should be easy to play and understand the game by user for every age. Therefore, we have made most of the mechanics as simple as possible. This comes with a tradeoff of functionality since each functionality adds another complexity after some threshold. For example, for stage selection we could have many stages for each category. But the patterns will be randomly generated we can use a single stage icon for each game mode and board size. This way, the stage system will be more usable but we won't be able to store records for separate stages for the same board size and category. We will keep the high score instead.

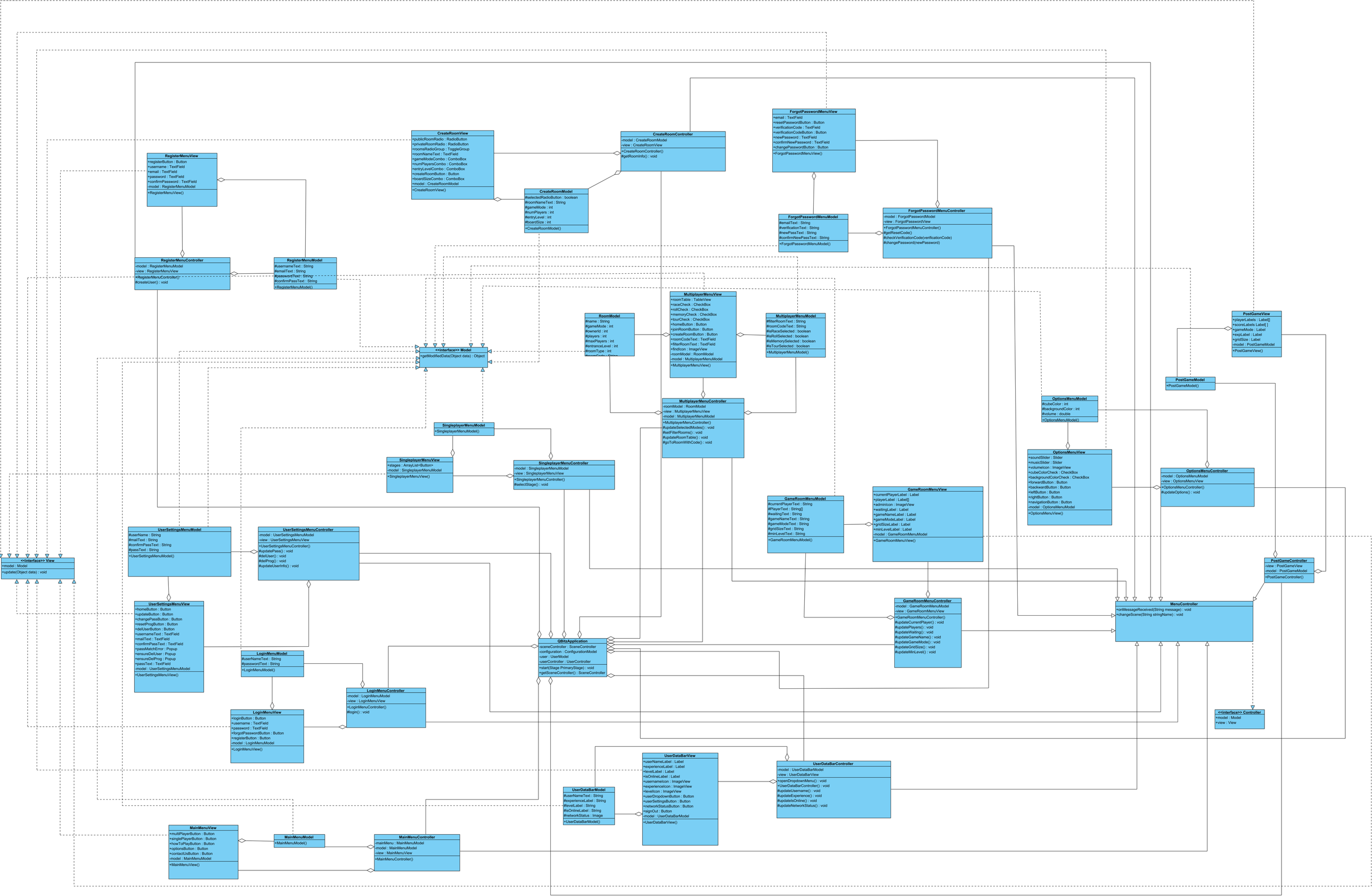
4.1.2 Rapid Development vs Functionality:

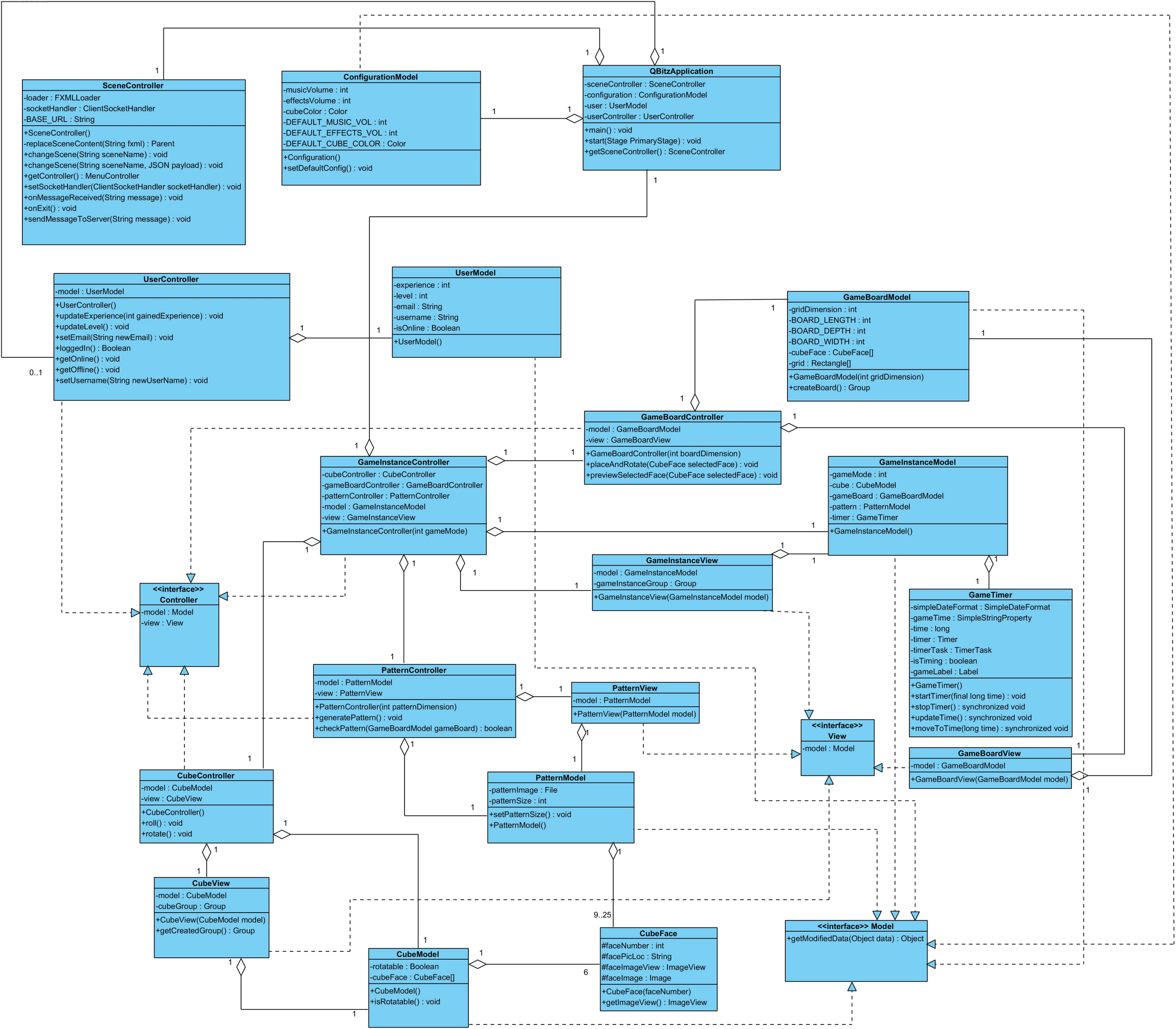
Since we have a time constraint for the project and we are implementing it iteratively, we have limited amount of work dedicated to the project which is allocated as 14 weeks. Thus in order to develop the game within this period we wanted to keep some functionalities as simple as possible. For example we will keep the multiplayer game mode really simple such that it provides enough functionality to play the game with more than one players but additional features such as best player of the month etc. will not be implemented. This is a tradeoff between the rapid development of the project and its functionalities.

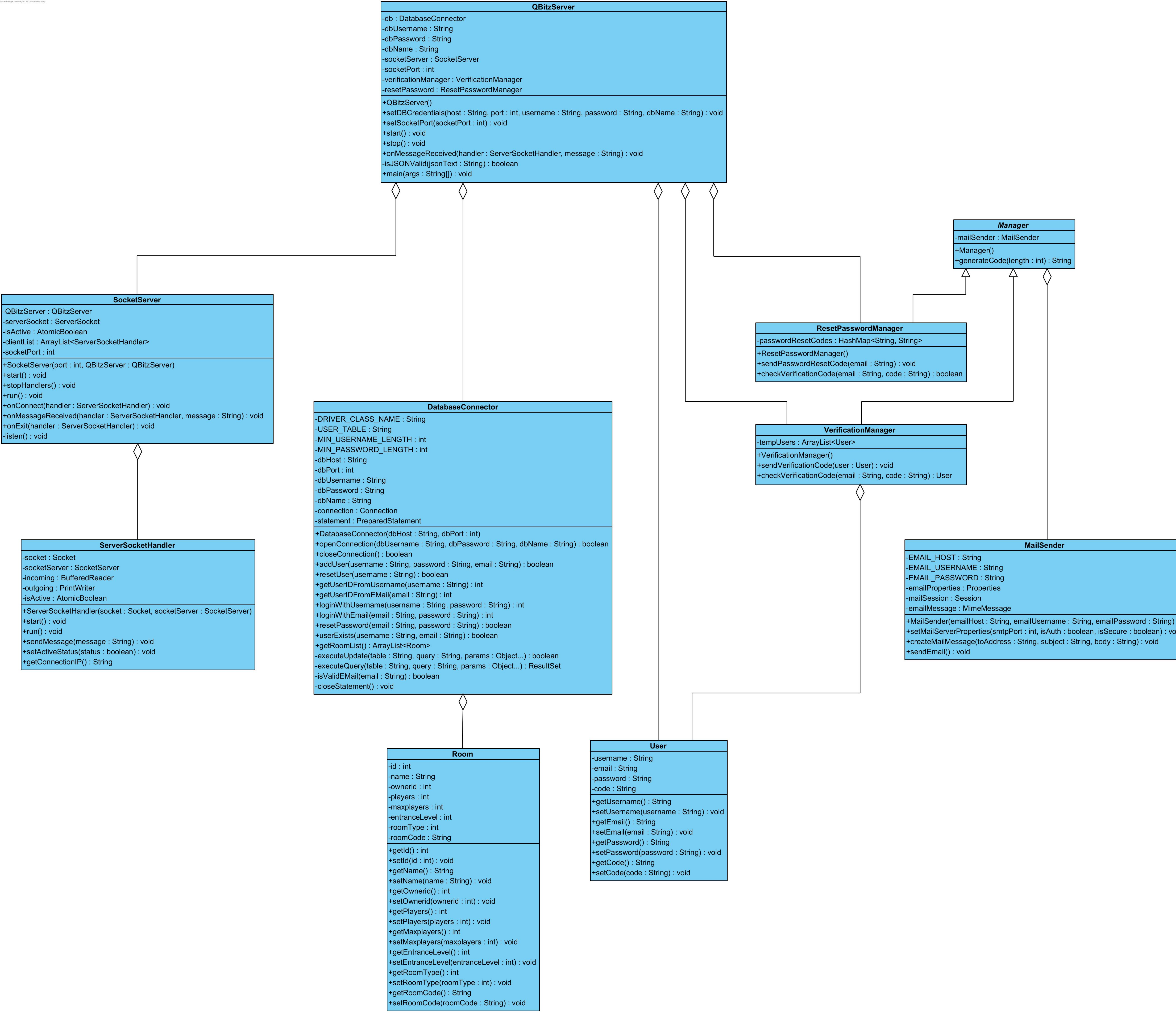
4.1.3 Efficiency vs. Portability:

The game contains 3D graphics which is an extensive task for a computer to render compared to other computations. This could have been achieved in a more efficient manner if we were using C or C++ with complementary 3D Graphics engines. Instead we are building the game using Java in order to be played independently from the platform that it is executed. We also know that 3D Graphics rendering would be slower but we are making a trade of between efficiency and portability to have slightly slower performance vs. being cross-platform.

4.2 Final Object Design


CreateRoomView
+publicRoomRadio : RadioButton
+privateRoomRadio : RadioButton
+roomsRadioGroup : ToggleGroup
+roomNameText : TextField
+gameModeCombo : ComboBox
+numPlayersCombo : ComboBox
+entryLevelCombo : ComboBox
+createRoomButton : Button
+boardSizeCombo : ComboBox
+model : CreateRoomModel
+CreateRoomView()
CreateRoomController
-model : CreateRoomModel
-view : CreateRoomView
+CreateRoomController()
#getRoomInfo() : void
CreateRoomModel
#selectedRadioButton : boolean
#roomNameText : String
#gameMode : int
#numPlayers : int
#entryLevel : int
#boardSize : int
+CreateRoomModel()
ForgotPasswordMenuView
+email : TextField
+resetPasswordButton : Button
+verificationCode : TextField
+verificationCodeButton : Button
+newPassword : TextField
+confirmNewPassword : TextField
+changePasswordButton : Button
+ForgotPasswordMenuView()
ForgotPasswordMenuModel
#emailText : String
#verificationText : String
#newPassText : String
#confirmNewPassText : String
+ForgotPasswordMenuModel()
ForgotPasswordMenuController
-model : ForgotPasswordModel
-view : ForgotPasswordView
+ForgotPasswordMenuController()
#getResetCode()
#checkVerificationCode(verificationCode)
#changePassword(newPassword)
RegisterMenuView
+registerButton : Button
+username : TextField
+email : TextField
+password : TextField
+confirmPassword : TextField
-model : RegisterMenuModel
+RegisterMenuView()
RegisterMenuController
-model : RegisterMenuModel
-view : RegisterMenuView
+RegisterMenuController()
#createUser() : void
RegisterMenuModel
#usernameText : String
#emailText : String
#passwordText : String
#confirmPassText : String
+RegisterMenuModel()
<<interface>> Model
+getModifiedData(Object data) : Object
RoomModel
#name : String
#gameMode : int
#ownerId : int
#players : int
#maxPlayers : int
#entranceLevel : int
#roomType : int
MultiplayerMenuView
+roomTable : TableView
+raceCheck : CheckBox
+rollCheck : CheckBox
+memoryCheck : CheckBox
+tourCheck : CheckBox
+homeButton : Button
+joinRoomButton : Button
+createRoomButton : Button
+roomCodeText : TextField
+filterRoomText : TextField
+findIcon : ImageView
-roomModel : RoomModel
-model : MultiplayerMenuModel
+MultiplayerMenuView()
MultiplayerMenuModel
#filterRoomText : String
#roomCodeText : String
#isRaceSelected : boolean
#isRollSelected : boolean
#isMemorySelected : boolean
#isTourSelected : boolean
+MultiplayerMenuModel()
MultiplayerMenuController
-roomModel : RoomModel
-view : MultiplayerMenuView
-model : MultiplayerMenuModel
+MultiplayerMenuController()
#updateSelectedModes() : void
#setFilterRooms() : void
#updateRoomTable() : void
#goToRoomWithCode() : void
PostGameView
+playerLabels : Label[]
+scoreLabels Label[ ]
+gameMode : Label
+expLabel : Label
+gridSize : Label
-model : PostGameModel
+PostGameView()
PostGameModel
+PostGameModel()
PostGameController
-view : PostGameView
-model : PostGameModel
+PostGameController()
OptionsMenuModel
#cubeColor : int
#backgroundColor : int
#volume : double
+OptionsMenuModel()
OptionsMenuView
+soundSlider : Slider
+musicSlider : Slider
+volumeIcon : ImageView
+cubeColorCheck : CheckBox
+backgroundColorCheck : CheckBox
+forwardButton : Button
+backwardButton : Button
+leftButton : Button
+rightButton : Button
+navigationButton : Button
-model : OptionsMenuModel
+OptionsMenuView()
OptionsMenuController
-model : OptionsMenuModel
-view : OptionsMenuView
+OptionsMenuController()
#updateOptions() : void
SingleplayerMenuModel
+SingleplayerMenuModel()
SingleplayerMenuView
+stages : ArrayList<Button>
-model : SingleplayerMenuModel
+SingleplayerMenuView()
SingleplayerMenuController
-model : SingleplayerMenuModel
-view : SingleplayerMenuView
+SingleplayerMenuController()
#selectStage() : void
GameRoomMenuModel
#currentPlayerText : String
#PlayerText : String[]
#waitingText : String
#gameNameText : String
#gameModeText : String
#gridSizeText : String
#minLevelText : String
+GameRoomMenuModel()
GameRoomMenuView
+currentPlayerLabel : Label
+playerLabel : Label[]
+adminIcon : ImageView
+waitingLabel : Label
+gameNameLabel : Label
+gameModeLabel : Label
+gridSizeLabel : Label
+minLevelLabel : Label
-model : GameRoomMenuModel
+GameRoomMenuView()
GameRoomMenuController
-model : GameRoomMenuModel
-view : GameRoomMenuView
+GameRoomMenuController()
#updateCurrentPlayer() : void
#updatePlayers() : void
#updateWaiting() : void
#updateGameName() : void
#updateGameMode() : void
#updateGridSize() : void
#updateMinLevel() : void
UserSettingsMenuModel
#userName : String
#mailText : String
#confirmPassText : String
#passText : String
+UserSettingsMenuModel()
UserSettingsMenuController
-model : UserSettingsMenuModel
-view : UserSettingsMenuView
+UserSettingsMenuController()
#updatePass() : void
#delUser() : void
#delProg() : void
#updateUserInfo() : void
UserSettingsMenuView
+homeButton : Button
+updateButton : Button
+changePassButton : Button
+resetProgButton : Button
+delUserButton : Button
+usernameText : TextField
+mailText : TextField
+confirmPassText : TextField
+passMatchError : Popup
+ensureDelUser : Popup
+ensureDelProg : Popup
+passText : TextField
-model : UserSettingsMenuModel
+UserSettingsMenuView()
LoginMenuModel
#userNameText : String
#passwordText : String
+LoginMenuModel()
LoginMenuController
-model : LoginMenuModel
-view : LoginMenuView
+LoginMenuController()
#login() : void
LoginMenuView
+loginButton : Button
+username : TextField
+password : TextField
+forgotPasswordButton : Button
+registerButton : Button
-model : LoginMenuModel
+LoginMenuView()
QBlitzApplication
-sceneController : SceneController
-configuration : ConfigurationModel
-user : UserModel
-userController : UserController
+start(Stage PrimaryStage) : void
+getSceneController() : SceneController
MenuController
+onMessageReceived(String message) : void
+changeScene(String stringName) : void
<<interface>> Controller
+model : Model
+view : View
<<interface>> View
+model : Model
+update(Object data) : void
UserDataBarModel
#userNameText : String
#experienceLabel : String
#levelLabel : String
#isOnlineLabel : String
#networkStatus : Image
+UserDataBarModel()
UserDataBarView
+userNameLabel : Label
+experienceLabel : Label
+levelLabel : Label
+isOnlineLabel : Label
+usernameIcon : ImageView
+experienceIcon : ImageView
+levelIcon : ImageView
+userDropdownButton : Button
+userSettingsButton : Button
+networkStatusButton : Button
+signOut : Button
-model : UserDataBarModel
+UserDataBarView()
UserDataBarController
-model : UserDataBarModel
-view : UserDataBarView
+openDropdownMenu() : void
+UserDataBarController() : void
#updateUsername() : void
#updateExperience() : void
#updateIsOnline() : void
#updateNetworkStatus() : void
MainMenuModel
+MainMenuModel()
MainMenuController
-mainMenu : MainMenuModel
-model : MainMenuModel
-view : MainMenuView
+MainMenuController()
MainMenuView
+multiPlayerButton : Button
+singlePlayerButton : Button
+howToPlayButton : Button
+optionsButton : Button
+contactUsButton : Button
-model : MainMenuModel
+MainMenuView()


SceneController
-loader : FXMLLoader
-socketHandler : ClientSocketHandler
-BASE_URL : String
+SceneController()
-replaceSceneContent(String fxml) : Parent
+changeScene(String sceneName) : void
+changeScene(String sceneName, JSON payload) : void
+getController() : MenuController
+setSocketHandler(ClientSocketHandler socketHandler) : void
+onMessageReceived(String message) : void
+onExit() : void
+sendMessageToServer(String message) : void
ConfigurationModel
-musicVolume : int
-effectsVolume : int
-cubeColor : Color
-DEFAULT_MUSIC_VOL : int
-DEFAULT_EFFECTS_VOL : int
-DEFAULT_CUBE_COLOR : Color
+Configuration()
+setDefaultConfig() : void
QBitzApplication
-sceneController : SceneController
-configuration : ConfigurationModel
-user : UserModel
-userController : UserController
+main() : void
+start(Stage PrimaryStage) : void
+getSceneController() : SceneController
UserController
-model : UserModel
+UserController()
+updateExperience(int gainedExperience) : void
+updateLevel() : void
+setEmail(String newEmail) : void
+loggedIn() : Boolean
+getOnline() : void
+getOffline() : void
+setUsername(String newUserName) : void
UserModel
-experience : int
-level : int
-email : String
-username : String
-isOnline : Boolean
+UserModel()
GameBoardModel
-gridDimension : int
-BOARD_LENGTH : int
-BOARD_DEPTH : int
-BOARD_WIDTH : int
-cubeFace : CubeFace[]
-grid : Rectangle[]
+GameBoardModel(int gridDimension)
+createBoard() : Group
GameBoardController
-model : GameBoardModel
-view : GameBoardView
+GameBoardController(int boardDimension)
+placeAndRotate(CubeFace selectedFace) : void
+previewSelectedFace(CubeFace selectedFace) : void
GameInstanceController
-cubeController : CubeController
-gameBoardController : GameBoardController
-patternController : PatternController
-model : GameInstanceModel
-view : GameInstanceView
+GameInstanceController(int gameMode)
GameInstanceModel
-gameMode : int
-cube : CubeModel
-gameBoard : GameBoardModel
-pattern : PatternModel
-timer : GameTimer
+GameInstanceModel()
GameInstanceView
-model : GameInstanceModel
-gameInstanceGroup : Group
+GameInstanceView(GameInstanceModel model)
<<interface>>
Controller
-model : Model
-view : View
PatternController
-model : PatternModel
-view : PatternView
+PatternController(int patternDimension)
+generatePattern() : void
+checkPattern(GameBoardModel gameBoard) : boolean
PatternView
-model : PatternModel
+PatternView(PatternModel model)
GameTimer
-simpleDateFormat : SimpleDateFormat
-gameTime : SimpleStringProperty
-time : long
-timer : Timer
-timerTask : TimerTask
-isTiming : boolean
-gameLabel : Label
+GameTimer()
+startTimer(final long time) : void
+stopTimer() : synchronized void
+updateTime() : synchronized void
+moveToTime(long time) : synchronized void
<<interface>>
View
-model : Model
GameBoardView
-model : GameBoardModel
+GameBoardView(GameBoardModel model)
CubeController
-model : CubeModel
-view : CubeView
+CubeController()
+roll() : void
+rotate() : void
PatternModel
-patternImage : File
-patternSize : int
+setPatternSize() : void
+PatternModel()
CubeView
-model : CubeModel
-cubeGroup : Group
+CubeView(CubeModel model)
+getCreatedGroup() : Group
CubeModel
-rotatable : Boolean
-cubeFace : CubeFace[]
+CubeModel()
+isRotatable() : void
CubeFace
#faceNumber : int
#facePicLoc : String
#faceImageView : ImageView
#faceImage : Image
+CubeFace(faceNumber)
+getImageView() : ImageView
<<interface>> Model
+getModifiedData(Object data) : Object
1
0..1
6
9..25


QBitzServer
-db : DatabaseConnector
-dbUsername : String
-dbPassword : String
-dbName : String
-socketServer : SocketServer
-socketPort : int
-verificationManager : VerificationManager
-resetPassword : ResetPasswordManager
+QBitzServer()
+setDBCredentials(host : String, port : int, username : String, password : String, dbName : String) : void
+setSocketPort(socketPort : int) : void
+start() : void
+stop() : void
+onMessageReceived(handler : ServerSocketHandler, message : String) : void
-isJSONValid(jsonText : String) : boolean
+main(args : String[]) : void
Manager
-mailSender : MailSender
+Manager()
+generateCode(length : int) : String
SocketServer
-QBitzServer : QBitzServer
-serverSocket : ServerSocket
-isActive : AtomicBoolean
-clientList : ArrayList<ServerSocketHandler>
-socketPort : int
+SocketServer(port : int, QBitzServer : QBitzServer)
+start() : void
+stopHandlers() : void
+run() : void
+onConnect(handler : ServerSocketHandler) : void
+onMessageReceived(handler : ServerSocketHandler, message : String) : void
+onExit(handler : ServerSocketHandler) : void
-listen() : void
ResetPasswordManager
-passwordResetCodes : HashMap<String, String>
+ResetPasswordManager()
+sendPasswordResetCode(email : String) : void
+checkVerificationCode(email : String, code : String) : boolean
DatabaseConnector
-DRIVER_CLASS_NAME : String
-USER_TABLE : String
-MIN_USERNAME_LENGTH : int
-MIN_PASSWORD_LENGTH : int
-dbHost : String
-dbPort : int
-dbUsername : String
-dbPassword : String
-dbName : String
-connection : Connection
-statement : PreparedStatement
+DatabaseConnector(dbHost : String, dbPort : int)
+openConnection(dbUsername : String, dbPassword : String, dbName : String) : boolean
+closeConnection() : boolean
+addUser(username : String, password : String, email : String) : boolean
+resetUser(username : String) : boolean
+getUserIDFromUsername(username : String) : int
+getUserIDFromEMail(email : String) : int
+loginWithUsername(username : String, password : String) : int
+loginWithEmail(email : String, password : String) : int
+resetPassword(email : String, password : String) : boolean
+userExists(username : String, email : String) : boolean
+getRoomList() : ArrayList<Room>
-executeUpdate(table : String, query : String, params : Object...) : boolean
-executeQuery(table : String, query : String, params : Object...) : ResultSet
-isValidEMail(email : String) : boolean
-closeStatement() : void
VerificationManager
-tempUsers : ArrayList<User>
+VerificationManager()
+sendVerificationCode(user : User) : void
+checkVerificationCode(email : String, code : String) : User
ServerSocketHandler
-socket : Socket
-socketServer : SocketServer
-incoming : BufferedReader
-outgoing : PrintWriter
-isActive : AtomicBoolean
+ServerSocketHandler(socket : Socket, socketServer : SocketServer)
+start() : void
+run() : void
+sendMessage(message : String) : void
+setActiveStatus(status : boolean) : void
+getConnectionIP() : String
MailSender
-EMAIL_HOST : String
-EMAIL_USERNAME : String
-EMAIL_PASSWORD : String
-emailProperties : Properties
-mailSession : Session
-emailMessage : MimeMessage
+MailSender(emailHost : String, emailUsername : String, emailPassword : String)
+setMailServerProperties(smtpPort : int, isAuth : boolean, isSecure : boolean) : void
+createMailMessage(toAddress : String, subject : String, body : String) : void
+sendEmail() : void
Room
-id : int
-name : String
-ownerid : int
-players : int
-maxplayers : int
-entranceLevel : int
-roomType : int
-roomCode : String
+getId() : int
+setId(id : int) : void
+getName() : String
+setName(name : String) : void
+getOwnerid() : int
+setOwnerid(ownerid : int) : void
+getPlayers() : int
+setPlayers(players : int) : void
+getMaxplayers() : int
+setMaxplayers(maxplayers : int) : void
+getEntranceLevel() : int
+setEntranceLevel(entranceLevel : int) : void
+getRoomType() : int
+setRoomType(roomType : int) : void
+getRoomCode() : String
+setRoomCode(roomCode : String) : void
User
-username : String
-email : String
-password : String
-code : String
+getUsername() : String
+setUsername(username : String) : void
+getEmail() : String
+setEmail(email : String) : void
+getPassword() : String
+setPassword(password : String) : void
+getCode() : String
+setCode(code : String) : void

4.3 Packages

javafx.* : For all the user interfaces and 3D graphics, we used JAVAFX package.

java.util.* : Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes

java.sql : Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™ programming language.

java.io : Provides for system input and output through data streams, serialization and the file system.

java.net : Provides the classes for implementing networking applications.

org.json : We use JSON for the messages sent between the server and client for communication.

4.4 Class Interfaces

Model Interface:

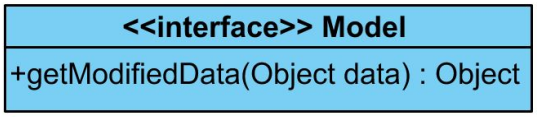


Figure 66: *Model Interface*

View Interface:

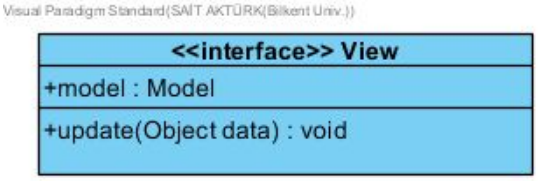


Figure 67: *View Interface*

Controller Interface:

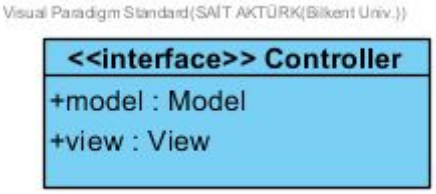


Figure 68: *Controller Interface*

5. Improvement Summary

- We have updated our class diagrams in order to better represent the MVC design pattern that we are enforcing throughout the game.
- We added deployment diagram which shows high level organization of the systems and the communication between the server-side and client-side of the project.
- We added some interfaces which used to label and categorize some model classes and also used to enforce the implementation of specific methods for controllers.
- We updated the tradeoffs we deal with in the project.
- We updated the diagrams with the ones which have higher qualities.

6. Glossary & references

[1] "Getting Started with JavaFX 3D Graphics." *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, http://docs.oracle.com/javase/8/javafx/graphics-tutorial/javafx-3d-graphics.htm

[2] Capital, Board Game. "Q-Bitz Game Rules." *Board Game Capital*, www.boardgamecapital.com/q-bitz-rules.htm

[3] Model-View-Controller
https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649643