# Object-Oriented Software Engineering

**CS319**

**Term Project**
**Design Report Q-Bitz**

Group No: 3F
Group Name: Wasted Potentials

Mertkan Akkus 21602951 Yasin
Alptekin Ay 21601849 Ahmet
Furkan Biyik 21501084
Ramazan Mert Cinar 21601985
Yaman Yagiz Tasbag 21601639

# 1. Introduction

## 1.1 Purpose of the System

Q-Bitz is originally a boardgame where players push the limits of speed and memory in different rounds of the game. Our purpose is to create a desktop version of this game with more fun by adding more futures. We will implement all the features of the real game to our project. However, our game will have additional modern features such as the game modes. We will offer the players a smooth multiplayer experience. Our game has singleplayer mode which is not available in the board game. We aim our desktop version of Q-Bitz to be adjustable by the users, so they can adapt in less than 10 minutes. Our game will also offer a pleasurable visual and audial environment for the players.

## 1.2 Design Goals

### 1.2.1 Usability

One of the main goals of our user interface design is to make the game easier to play. Since Q-Bitz is a game designed for children, menu navigation should take less than 10 minutes to understand. A player should not spend more than 1 minutes to find his/her friend's lobby. Game controls should be designed in a satisfying way for the users. Also the UI design and game control decisions should not be obstacles for a player.

### 1.2.2 High-Performance

A smooth gameplay experience is one of our goals. Our game performs its functions within a time-frame that is tolerable for the players, and does not require too much memory.

### 1.2.3 Reliability

Our system should operate properly in an event of failure at any component of the system. For example, if the server is down, which is not likely, the player should be able to keep using the non-server-related parts of the game or if a player gets disconnected from the game because his/her network issues, game should keep operating properly for the other players who are in the same lobby.

### 1.2.4 Portability

Our game requires small amounts of memory so users do not need too much space in their computers. Also Kubitz can be played on every platform that has JRE installed.

### 1.2.5 Scalability

Another objective is scalability of the game. Kubitz is adaptable to the increasing number of users and data.

## 1.3 Implementation Choices

### 1.3.1 Why Java?

- Practicality
- Backwards compatibility
- Scalability/Performance/Reliability
- Freshness

### 1.3.2 Why Spring?

- Exposing RESTful services
- Dependency injection approach
- Powerful database transaction management capabilities
- Integration with other Java frameworks

### 1.3.3 Why Maven?

- Easy build process
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

# 2. High-level Software Architecture

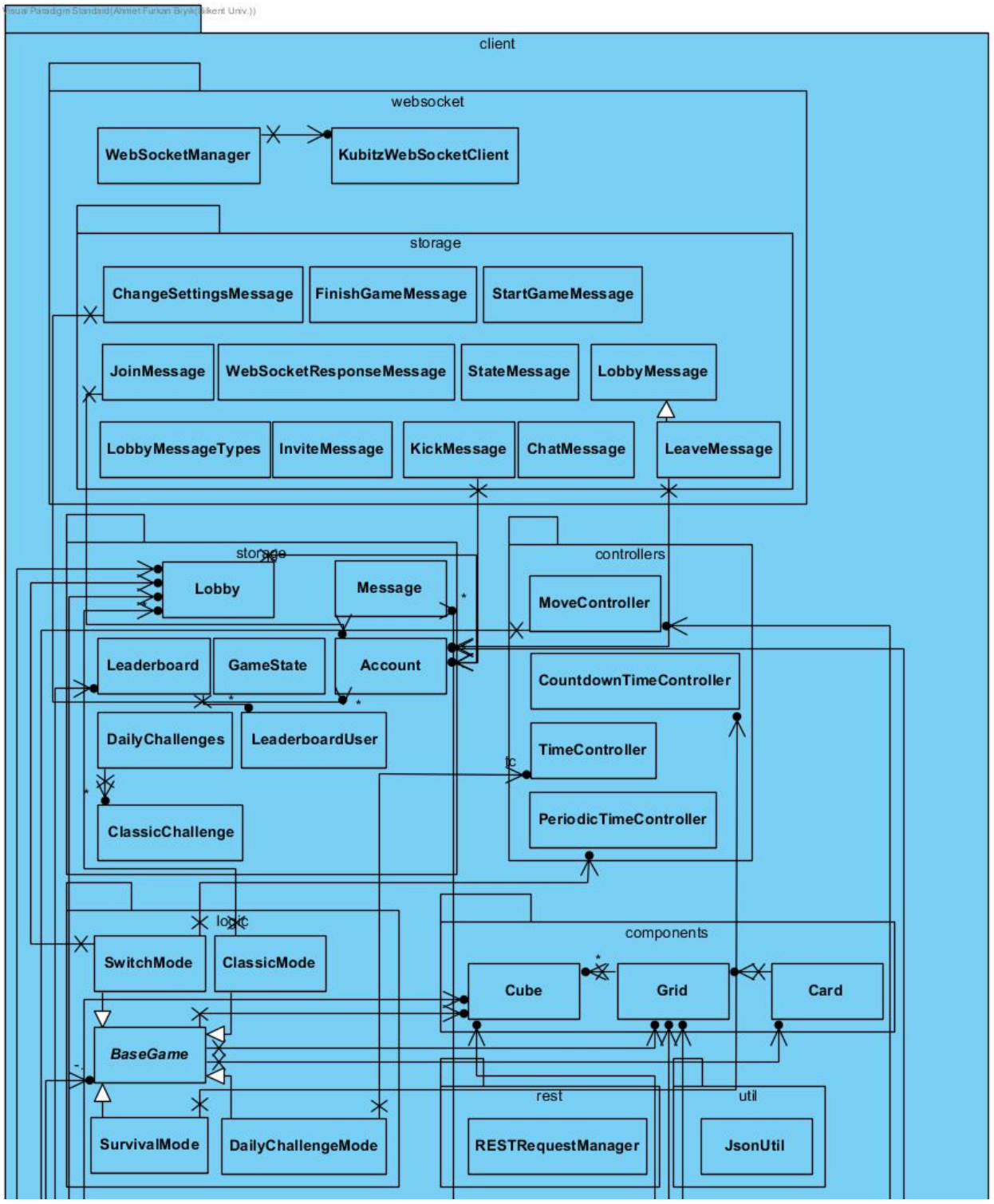
## 2.1 Overview

Main design issues and decisions will be discussed in this report. And our proposed design for the system will be illustrated. The overall architecture of the system is described and in the following sections the subsystems are detailed. Various diagrams and figures are used wherever they enhance the understanding of the system.

## 2.2 Subsystem Decomposition

Our game can be decomposed into two main parts: client and server. Client part consists of six main packages: gui, components, logic, controllers, storage and websocket as it is shown in the figure 1. GUI package includes the classes that are responsible of creating graphical user interfaces for all screens within the game. Controller package has the classes that are responsible of time and move controls of some game modes. Logic package includes the classes that implement game functionalities. Components package includes object classes needed for the game like cube, card and grid (game board). Storage package has the classes that needed for a multiplayer gameplay. Websocket package has manager for websocket communication with server and storage package. Storage package has classes to parse messages.

Server part consist of three packages which are controllers, websocket and database as it is shown in figure 2. Controller package has endpoints for each controller class in client. Websocket package has manager for websocket communication with client and storage package. Storage package has classes to parse messages. Database package is for communicating with database. It has packages that have repository and model for collections in database. It also includes a KubitzServer which implements communication with client.


client
websocket
WebSocketManager
KubitzWebSocketClient
storage
ChangeSettingsMessage
FinishGameMessage
StartGameMessage
JoinMessage
WebSocketResponseMessage
StateMessage
LobbyMessage
LobbyMessageTypes
InviteMessage
KickMessage
ChatMessage
LeaveMessage
storage
Lobby
Message
Leaderboard
GameState
Account
DailyChallenges
LeaderboardUser
ClassicChallenge
controllers
MoveController
CountdownTimeController
TimeController
tc
PeriodicTimeController
logic
SwitchMode
ClassicMode
BaseGame
SurvivalMode
DailyChallengeMode
components
Cube
Grid
Card
rest
RESTRequestManager
util
JsonUtil

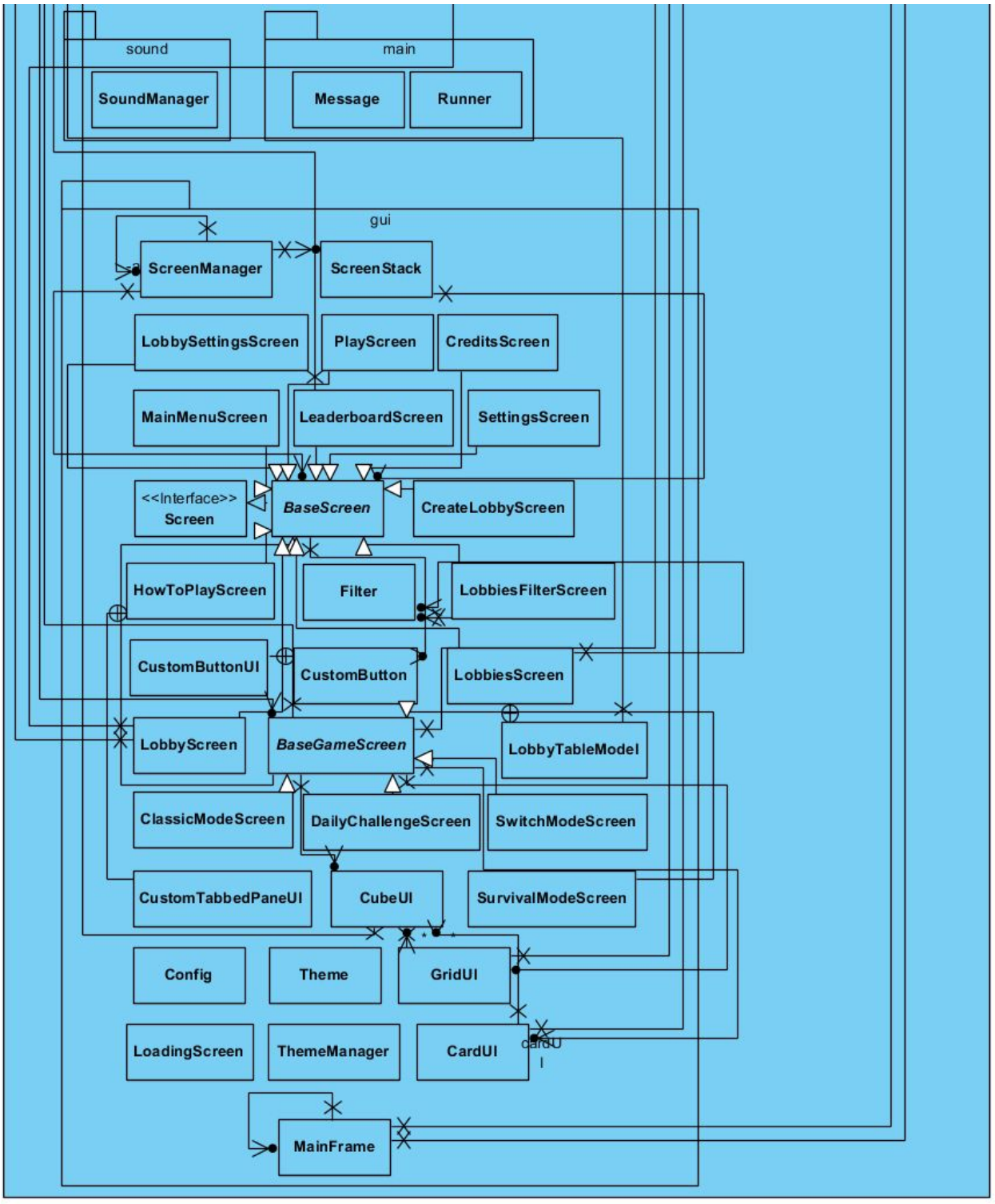
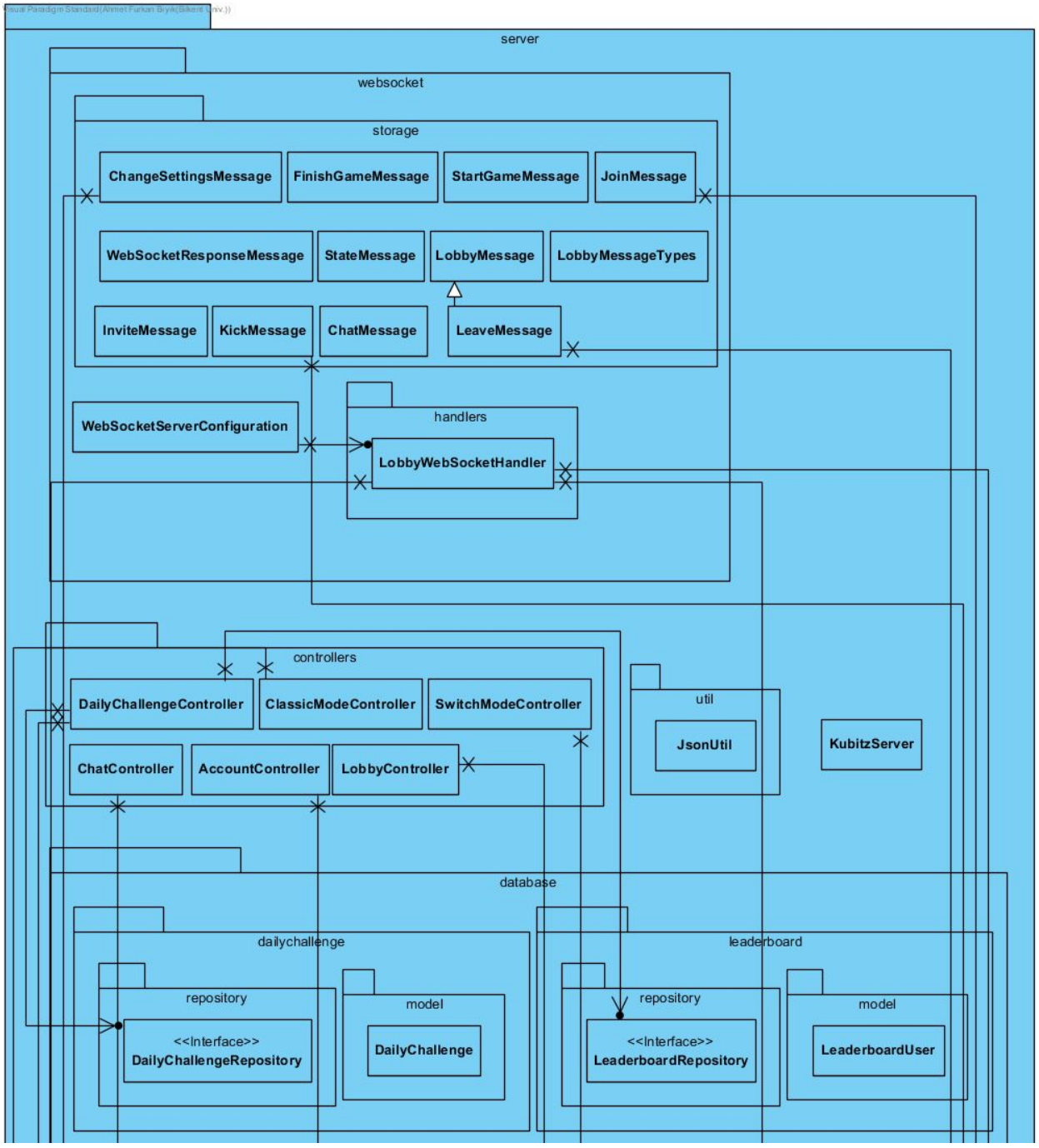

sound
main
SoundManager
Message
Runner
gui
ScreenManager
ScreenStack
LobbySettingsScreen
PlayScreen
CreditsScreen
MainMenuScreen
LeaderboardScreen
SettingsScreen
<<Interface>>
Screen
BaseScreen
CreateLobbyScreen
HowToPlayScreen
Filter
LobbiesFilterScreen
CustomButtonUI
CustomButton
LobbiesScreen
LobbyScreen
BaseGameScreen
LobbyTableModel
ClassicModeScreen
DailyChallengeScreen
SwitchModeScreen
CustomTabbedPaneUI
CubeUI
SurvivalModeScreen
Config
Theme
GridUI
LoadingScreen
ThemeManager
CardUI
cardU
I
MainFrame

Figure 1


server
websocket
storage
ChangeSettingsMessage
FinishGameMessage
StartGameMessage
JoinMessage
WebSocketResponseMessage
StateMessage
LobbyMessage
LobbyMessageTypes
InviteMessage
KickMessage
ChatMessage
LeaveMessage
WebSocketServerConfiguration
handlers
LobbyWebSocketHandler
controllers
DailyChallengeController
ClassicModeController
SwitchModeController
ChatController
AccountController
LobbyController
util
JsonUtil
KubitzServer
database
dailychallenge
repository
<<Interface>>
DailyChallengeRepository
model
DailyChallenge
leaderboard
repository
<<Interface>>
LeaderboardRepository
model
LeaderboardUser

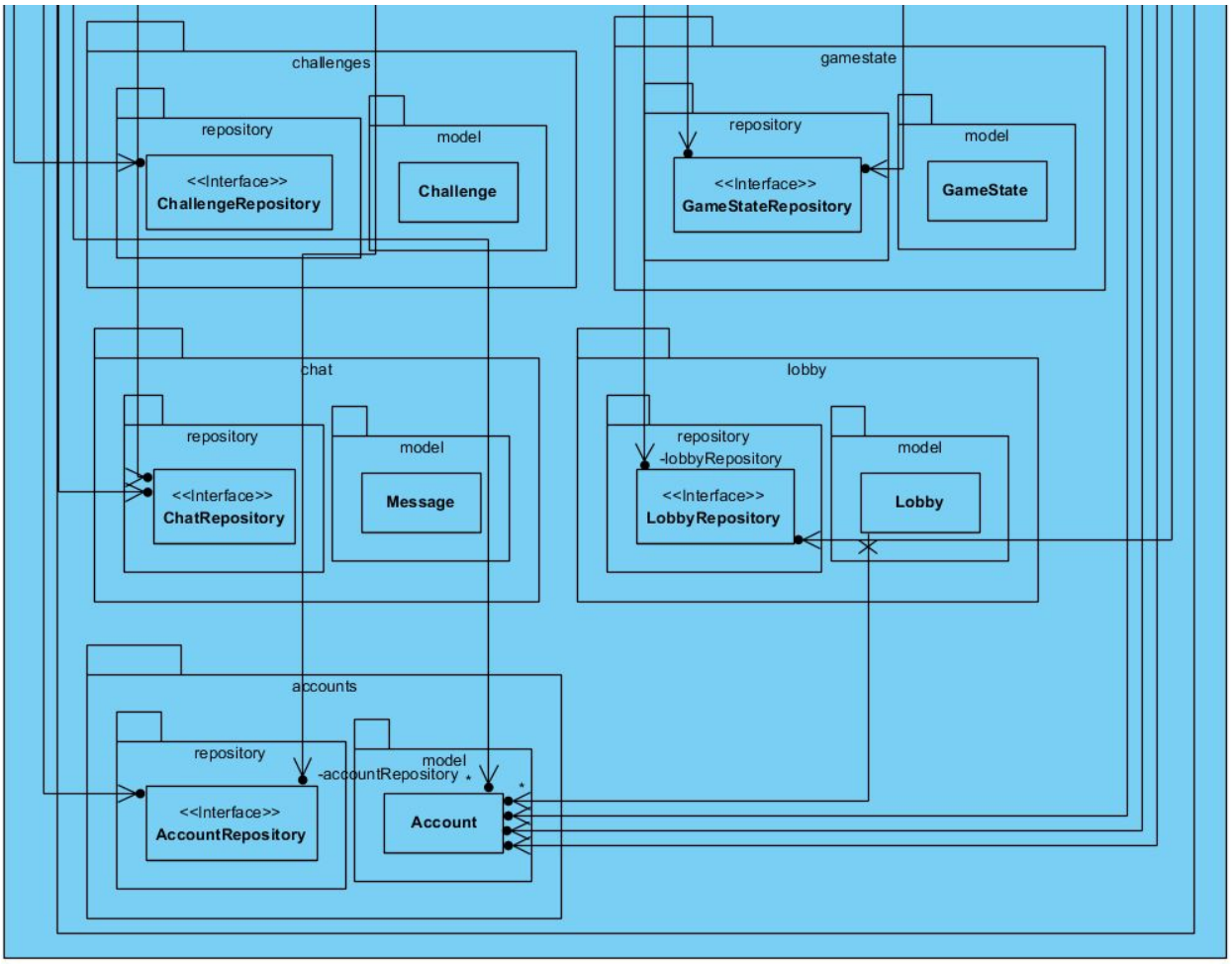



Figure 2

## 2.3 Hardware Software Mapping

This is a game project so there is no complex hardware usage. Following hardware components are mapped to our game using Java API:

- Keyboard: For cube rotation manipulation.
- Mouse: For menu navigation, also for cube rotation and placement.
- Monitor: The GUI is displayed on monitor.
- Speakers: For sound effects and game music.

## 2.4 Persistent Data Management

In our game, we do not have much data management issues however there are some points which we want to have persistent data management to create a seamless multiplayer experience. In order to reach this goal, we use both filesystem and database for efficiency related issues. It is not recommended to download all the data every time the game starts.

- User settings and game music are held in filesystem
- Leaderboard, a list of user data, game data, lobby data and chat data are held in database.

## 2.5 Access Control and Security

Kubitz game has a multiplayer game option. Normally all users are referred as players. Every player can create a lobby in multiplayer modes. However, the lobby creator is called admin and the admin has an access to kick player function, which basically kicks a player from the lobby. Other players do not have access to this kick button. There is another player type, which is called spectator. This player type can join the lobby as spectator and the only permission is to watch the game while other players are playing the game. In the lobby there is also the chat that players communicate each other. Only players in a lobby can see messages in chat. Players who are not in the lobby cannot access chat messages.

## 2.6 Boundary Conditions

### Initialization

The system starts up by the user opening the game. Then a splash screen comes up. After that, the main menu options will be shown. Game also communicates with the server and database beginning from the initialization state.

#### Termination

All the subsystems are terminated with the exit. The single user related things like user settings are already saved to the filesystem and the other multiplayer related things like leaderboard scores are already saved to the database.

#### Failure

When a player disconnects during the game, other players in the same lobby are informed that the player has disconnected. Every data related to multiplayer game modes are stored in the server. Therefore, the user data is not being lost in a system failure situation.

# 3 Low Level Design

## 3.1 Object Design Trade-offs

Some of the design goals that we chose are contradicting with some other software design goals in some circumstances. These contradictions are explained below.

### 3.1.1 Usability vs Flexibility

We are promising to the user that our game will be usable. However, usability conflicts with flexibility. As the flexibility of a system increases, its usability decreases. This trade off occurs because providing flexibility requires satisfying a larger set of requirements, which concludes in complexity and usability adjustments. Therefore, we chose usability over flexibility.

### 3.1.2 High-Performance vs Robustness

*"Something is efficient if it performs optimally under ideal circumstances. Something is robust if it performs pretty well under less than ideal circumstances."*

We are trying to create the game according to ideal circumstances. Since, we will not be able to test the game with a large scale of players we cannot predict what will happen to our system under stress. Therefore, we did our best to make our game operate as its best under ideal circumstances.

### 3.1.3 Portability vs Efficiency

Our game focuses on porting between different environments easily. With Java portability can be achieved easily. However, efficiency decreases since Java communicates machine via virtual machine.

### 3.1.4 Scalability vs High-Performance

A smooth gameplay experience is one of our goals. Our game performs its functions within a time-frame that is tolerable for the players, and does not require too much memory. However, it is not easy to foresee the situation where large amounts of players do the same operation at the same time. Our game can withstand large scale of users even it is not possible to test in our situation. But, when the game tolerates large amount of stress, it may waive the high-performance, for example this may result in higher server response time.

## 3.2 Packages and Final Object Design

Our project consists of two sub-projects which are client and server sides. Both packages will be explained further in their sub-sections.

### 3.2.1 Client Package

This package consists of six main packages which are components package, logic package, controllers package, gui package, storage package and websocket package. These packages will be explained below.

#### 3.2.1.1 Components Package

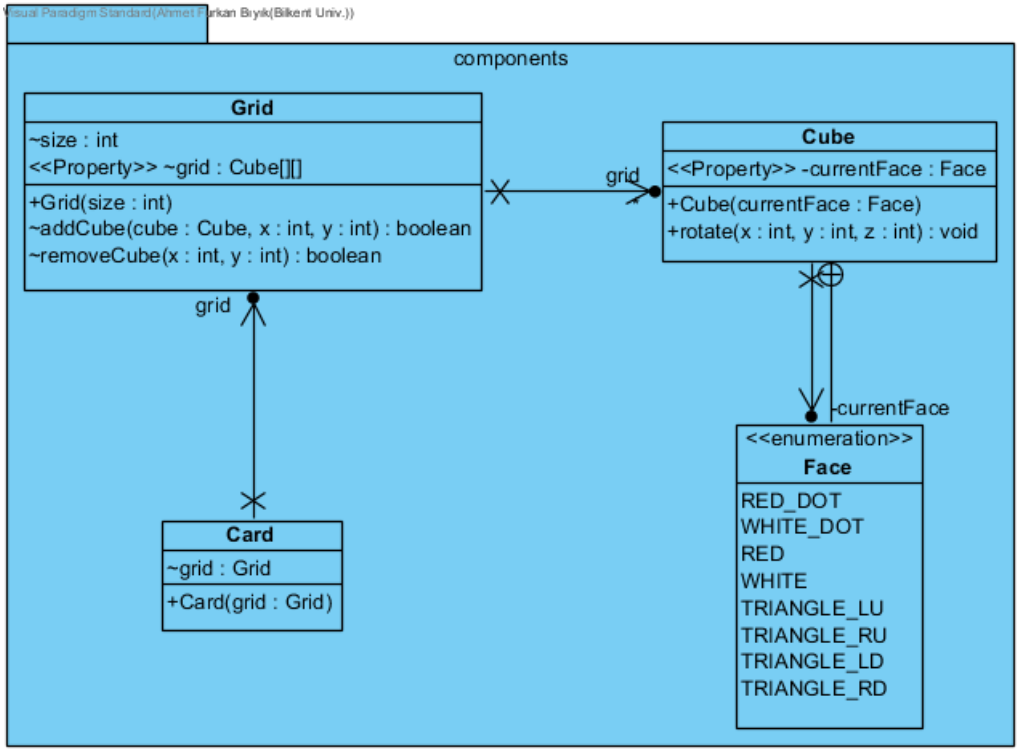



Figure 3

This Package contains the main components of this game, such as the parts in the original board game Q-bitz which are the simply game cards, cubes and a grid.

These main components are basically used like; Players/player check the figure on the game card and try to arrange the cubes on the grid as same as the figure.

#### 3.2.1.1.1 Grid

In Kubitz game, we have a class called Grid which is the base where the cubes will be placed on. It is designed as 4x4, so 16 cubes will be fitted when the grid is full.

#### 3.2.1.1.2 Cube

Cubes are used by the players to obtain the same figure as the game card. A cube has 6 faces and all these faces contain different shapes on a single cube. However, all the cubes are identical in the game. In the gameplay there is one particular cube (which is also same with the other cubes) that will be shown to the players, and be moved by the players to the grid.

#### 3.2.1.1.3 Card

Card class is used in all of the game modes such as the other components. Every card has a different figure on it and the players will try to arrange the cubes according to these cards. The cards will change every round depending on a game mode in a game but it is possible for a player to see a card twice or more by chance.

### 3.2.1.2 Logic Package

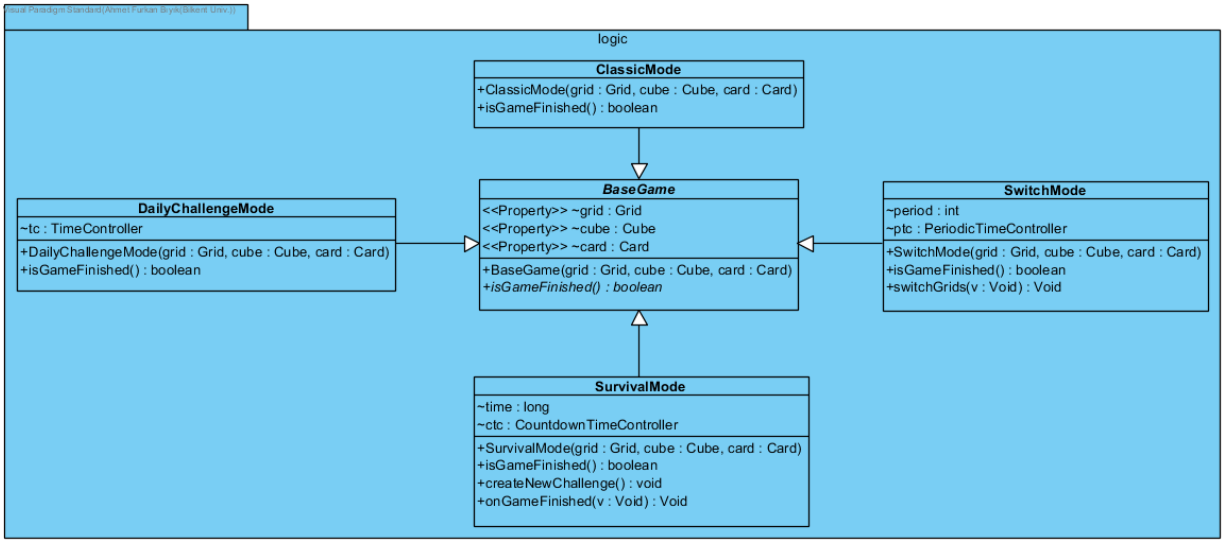



Figure 4

Logic Package (Figure 4) mainly contains the game modes which are classical, survival, switch and daily challenge. There is also a class called BaseGame which will be mentioned below.

#### 3.2.1.2.1 BaseGame

There is an abstract class called BaseGame in this package, which is the parent of the other game modes. This class contains the methods that is common with all the other game modes.

#### 3.2.1.2.2 SurvivalMode

Survival Mode is a game mode where players are competing against time. A player encounters with game cards one after another if they manage to keep solving. Every game card they manage to finish will give them extra time. This mode is a singleplayer game mode.

#### 3.2.1.2.3 DailyChallangeMode

Daily Challenge game mode is a mode that is created by the developers every day for the players to solve. Every player has one shot to try this mode every day. Players will come upon several game cards and they will be listed in the leaderboard according to the total time that they finish all these puzzles. This mode is a singleplayer game mode.

#### 3.2.1.2.4 ClassicMode

Classic Mode is a multiplayer mode. The players can take part in this game by either creating their own lobby or joining an existing lobby from the lobby list. This mode is the basic mode where players try to finish the game first.

#### 3.2.1.2.5 SwitchMode

Switch mode is also a multiplayer mode where the players can take part by either creating their own lobby or joining an existing lobby from the lobby list. Two players will try to finish the game first. In every 15 seconds, they players will switch the boards. If the overall placement of the board is correct, the game is over, and the winner will be displayed.

### 3.2.1.3 Controllers Package

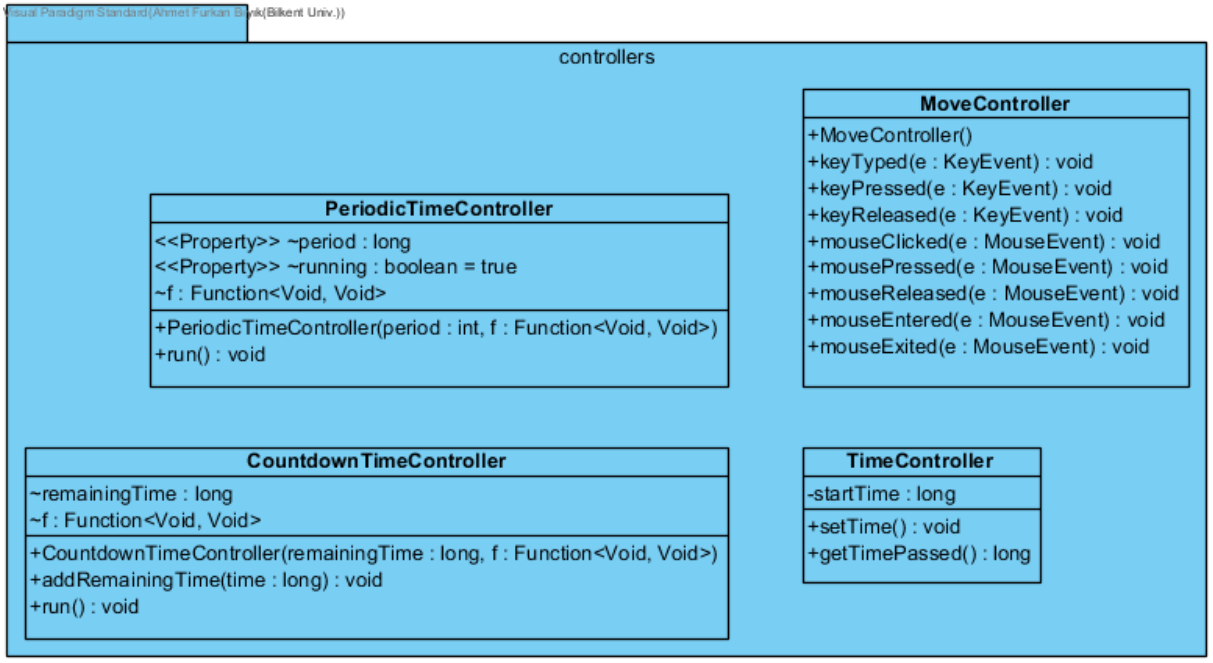



Figure 5

Controllers Package (Figure 5) consists of the main controllers for the game that will be described below in more detail. They are used by different game logics from logic package.

#### 3.2.1.3.1 CountdownTimeController

This time controller class is created to keep track of the time in the survival game mode. It has the time methods which are used in the survival mode. Player starts the game with an initial time which begins to decrease when the game starts and the player gains extra

time when he/she solves a game card.

#### 3.2.1.3.2 MoveController

This class is a controller to track the user actions such as mouse and keyboard activities.

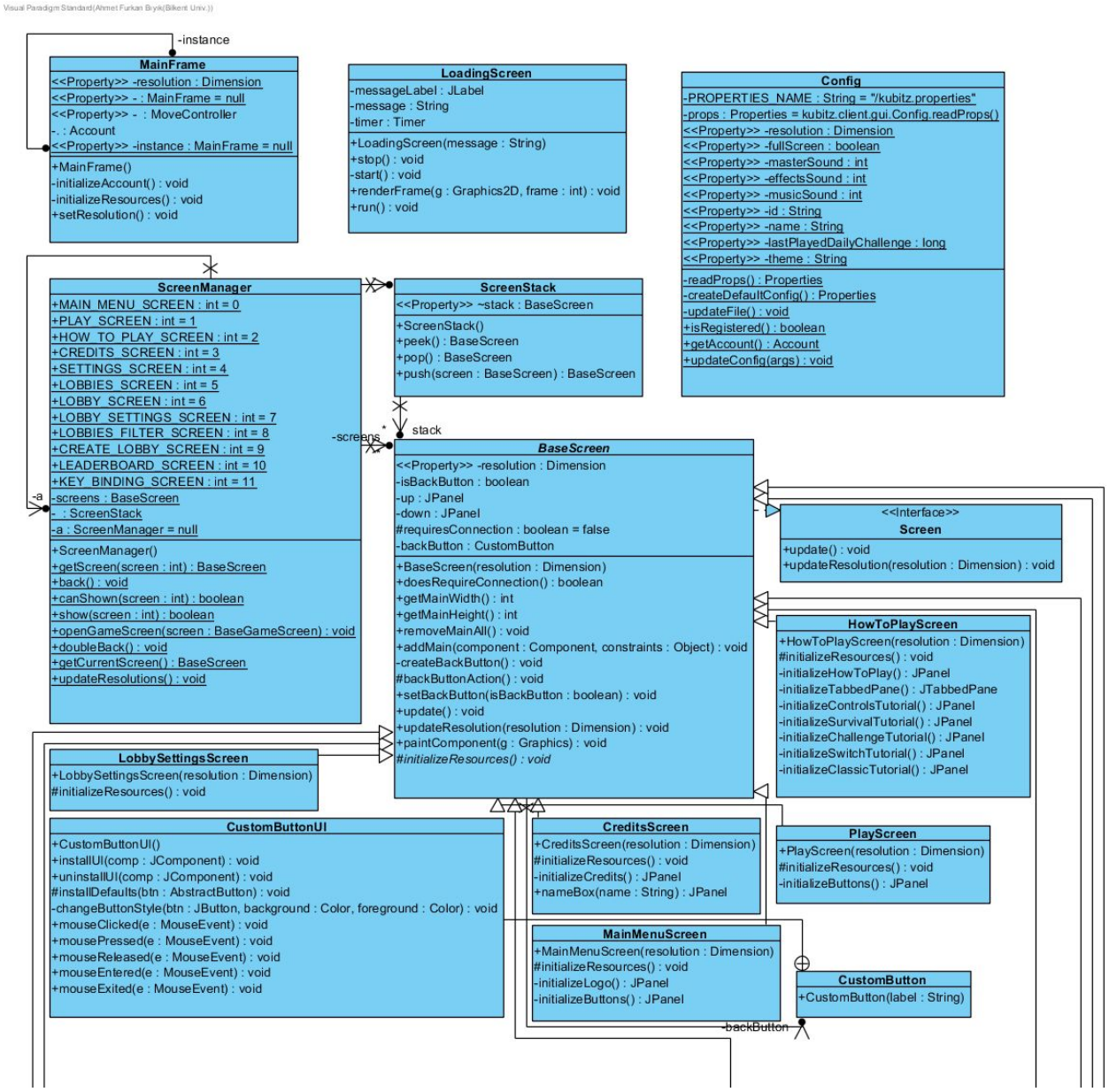
#### 3.2.1.3.3 PeriodicTimeController

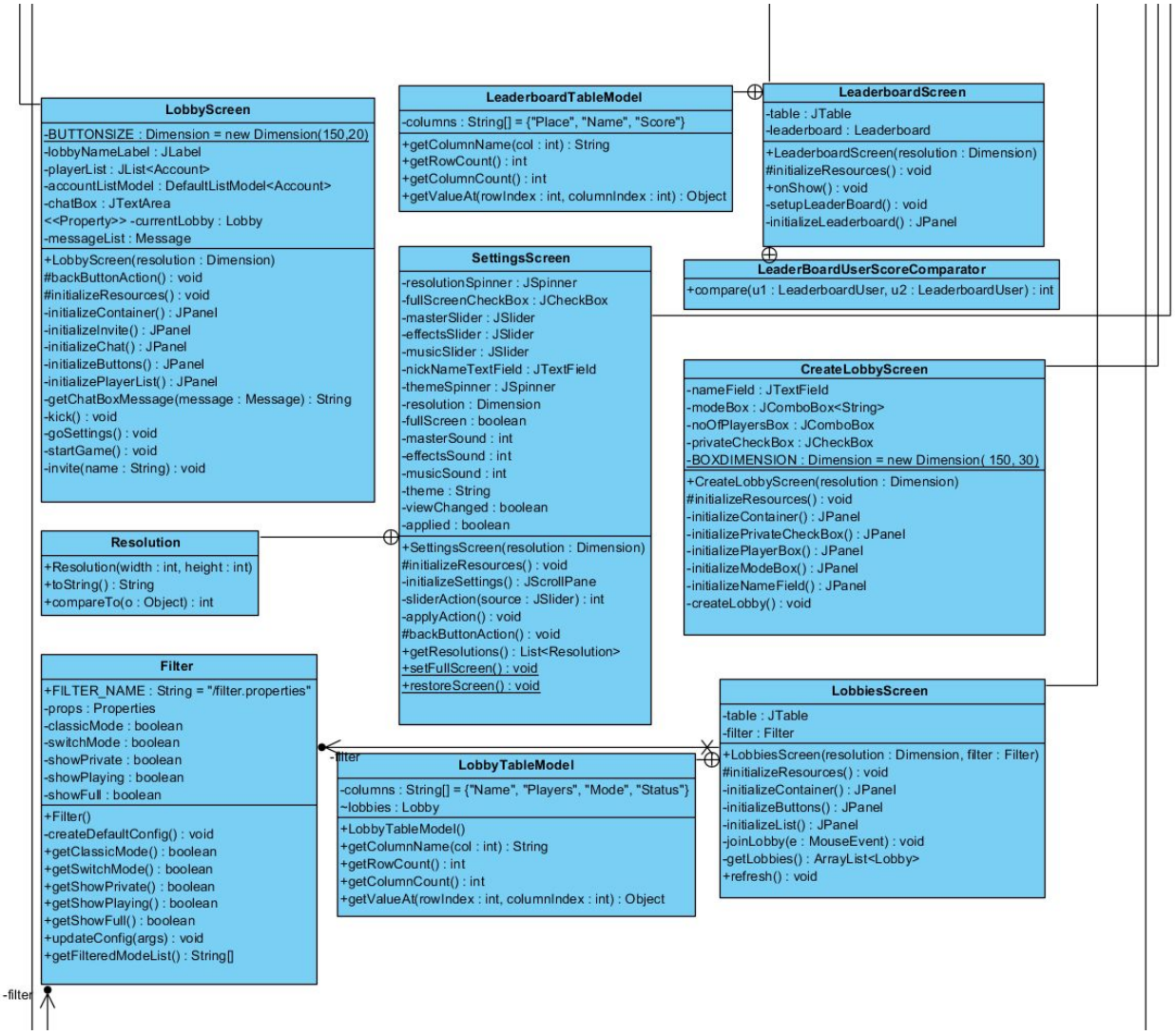
In the Switch game mode, the game must keep track of the time in predefined time periods. This time controller keeps these periods and tells when to switch the boards.

#### 3.2.1.3.4 TimeController

For the Daily Challenge Mode, total time passed to solve all the game cards must be saved by the game. Therefore, time controller saves this time to make the player listed in the leaderboard according to the finish time.

### 3.2.1.4 Gui Package




LobbyScreen
-BUTTONSIZE : Dimension = new Dimension(150,20)
-lobbyNameLabel : JLabel
-playerList : JList<Account>
-accountListModel : DefaultListModel<Account>
-chatBox : JTextArea
<<Property>> -currentLobby : Lobby
-messageList : Message
+LobbyScreen(resolution : Dimension)
#backButtonAction() : void
#initializeResources() : void
-initializeContainer() : JPanel
-initializeInvite() : JPanel
-initializeChat() : JPanel
-initializeButtons() : JPanel
-initializePlayerList() : JPanel
-getChatBoxMessage(message : Message) : String
-kick() : void
-goSettings() : void
-startGame() : void
-invite(name : String) : void
LeaderboardTableModel
-columns : String[] = {"Place", "Name", "Score"}
+getColumnName(col : int) : String
+getRowCount() : int
+getColumnCount() : int
+getValueAt(rowIndex : int, columnIndex : int) : Object
LeaderboardScreen
-table : JTable
-leaderboard : Leaderboard
+LeaderboardScreen(resolution : Dimension)
#initializeResources() : void
+onShow() : void
-setupLeaderBoard() : void
-initializeLeaderboard() : JPanel
LeaderBoardUserScoreComparator
+compare(u1 : LeaderboardUser, u2 : LeaderboardUser) : int
SettingsScreen
-resolutionSpinner : JSpinner
-fullScreenCheckBox : JCheckBox
-masterSlider : JSlider
-effectsSlider : JSlider
-musicSlider : JSlider
-nickNameTextField : JTextField
-themeSpinner : JSpinner
-resolution : Dimension
-fullScreen : boolean
-masterSound : int
-effectsSound : int
-musicSound : int
-theme : String
-viewChanged : boolean
-applied : boolean
+SettingsScreen(resolution : Dimension)
#initializeResources() : void
-initializeSettings() : JScrollPane
-sliderAction(source : JSlider) : int
-applyAction() : void
#backButtonAction() : void
+getResolutions() : List<Resolution>
+setFullScreen() : void
+restoreScreen() : void
CreateLobbyScreen
-nameField : JTextField
-modeBox : JComboBox<String>
-noOfPlayersBox : JComboBox
-privateCheckBox : JCheckBox
-BOXDIMENSION : Dimension = new Dimension( 150, 30)
+CreateLobbyScreen(resolution : Dimension)
#initializeResources() : void
-initializeContainer() : JPanel
-initializePrivateCheckBox() : JPanel
-initializePlayerBox() : JPanel
-initializeModeBox() : JPanel
-initializeNameField() : JPanel
-createLobby() : void
Resolution
+Resolution(width : int, height : int)
+toString() : String
+compareTo(o : Object) : int
Filter
+FILTER_NAME : String = "/filter.properties"
-props : Properties
-classicMode : boolean
-switchMode : boolean
-showPrivate : boolean
-showPlaying : boolean
-showFull : boolean
+Filter()
-createDefaultConfig() : void
+getClassicMode() : boolean
+getSwitchMode() : boolean
+getShowPrivate() : boolean
+getShowPlaying() : boolean
+getShowFull() : boolean
+updateConfig(args) : void
+getFilteredModeList() : String[]
-filter
LobbyTableModel
-columns : String[] = {"Name", "Players", "Mode", "Status"}
~lobbies : Lobby
+LobbyTableModel()
+getColumnName(col : int) : String
+getRowCount() : int
+getColumnCount() : int
+getValueAt(rowIndex : int, columnIndex : int) : Object
-filter
LobbiesScreen
-table : JTable
-filter : Filter
+LobbiesScreen(resolution : Dimension, filter : Filter)
#initializeResources() : void
-initializeContainer() : JPanel
-initializeButtons() : JPanel
-initializeList() : JPanel
-joinLobby(e : MouseEvent) : void
-getLobbies() : ArrayList<Lobby>
+refresh() : void
-filter

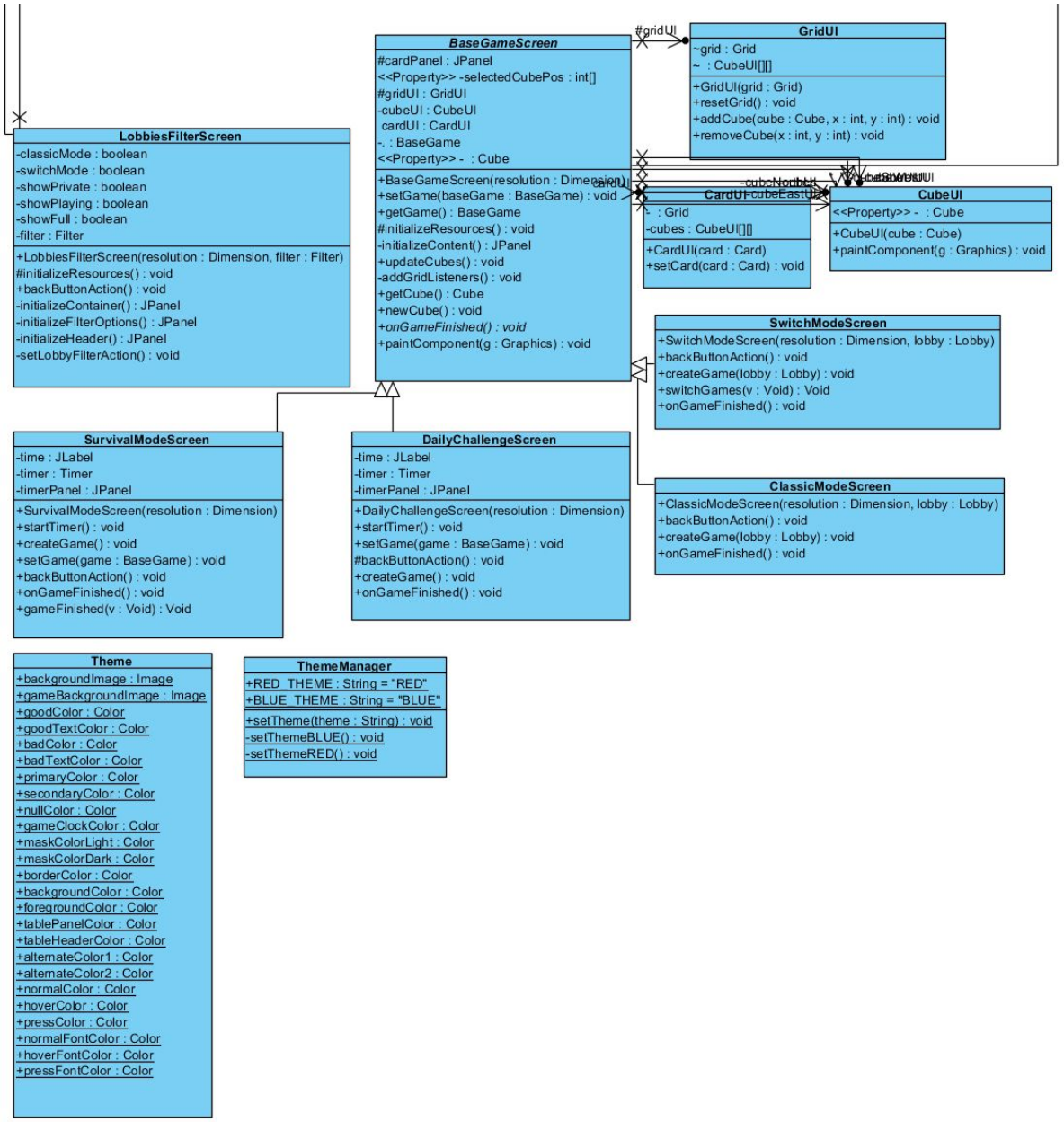



Figure 6

The Gui package (figure 6) contains user interfaces, screens, managers and configuration classes. They will be explained below.

3.2.1.4.1 Screen

This class is an interface which is a basis for other screens. It declares the common functions and properties.

3.2.1.4.2 BaseScreen

This class implements screen interface. It has common functionality for screens such as back button and its functionality.

3.2.1.4.3 LoadingScreen

This class is shown as popup while background operations need some to such as getting leaderboard from server. After background operation is finished this screen disposes.

3.2.1.4.4 Config

This class is I/O manager for settings. It reads properties of game such as resolution, sound volume, player name etc. When a setting is changed, this class writes data to storage.

3.2.1.4.5 ScreenManager

This class manage screens. It loads screens when game is launched in order to shorten response time between screen changes. It can change between screens. It has screen stack to keep track of screens and show previous screen when back button is pressed.

3.2.1.4.6 ThemeManager

This class sets color properties of game. Depends on color properties screens are constructed. Different themes can be set by this class.

3.2.1.4.7 CreditsScreen

This class is not a mostly interactive screen. It shows the credits of the game.

3.2.1.4.8 LobbyScreen

This class connects server and communicates server all the time. It shows players in

lobby and there is a friend invitation interface. There is also a chat box in which the players in the lobby can communicate with each other. In addition, it has some interactions for lobby leader such as start game, kick player, settings.

#### 3.2.1.4.9 LobbySettingsScreen

This class shows the settings of the lobby which only lobby leader can access. In this class lobby name, game mode, lobby size and status of the game lobby private or not can be set. This class also communicates with server.

#### 3.2.1.4.10 CreateLobbyScreen

This class shows the options of the lobby which will be created. In this class lobby name, game mode, lobby size and status of the game lobby private or not can be set. Then it creates a lobby and communicates with server.

#### 3.2.1.4.11 LobbiesScreen

This class shows the list of the lobbies which are created that are gotten from the server. It gets the list of joinable lobbies from the server by the means of sending the current filter preferences to the server and refreshes the list. When a lobby is double clicked on the list, it shows the lobby screen. Once the create lobby button pressed, it shows create lobby screen.

#### 3.2.1.4.12 LobbiesFilterScreen

This class has options for the games showed in the lobbies screen. It updates the current filter preferences.

#### 3.2.1.4.13 Filter

It has current filter preferences. Lobbies list is shown with its preferences.

3.2.1.4.14 MainMenuScreen

The current screen consists of buttons which have the responsibilities to show the respective screens.

3.2.1.4.15 PlayScreen

This class has buttons that are shows respective game choices such as multiplayer, daily challenge or survival. When multiplayer is clicked, it shows lobbies screen.

3.2.1.4.16 LeaderboardScreen

This class shows list of players with their scores in the current daily challenge. It fetches the data from the server.

3.2.1.4.17 HowToPlayScreen

This class shows the tutorials of the game. There are few pages that player can go between information.

3.2.1.4.18 SettingsScreen

This class nick name can be changed by sending request to server. There are options for graphics and sound. Settings can be save, apply or cancel.

3.2.1.4.19 BaseGameScreen

This class has common interfaces for other game modes. It has base game logic as a property. It renders the grid, the card and the cube that will be placed.

3.2.1.4.20 SwitchModeScreen

This class extends base game screen. It has a switch mode logic property. It shows time left from switch mode logic.

#### 3.2.1.4.21 DailyChallangeModeScreen

This class extends base game screen. It has daily challenge mode logic property. It shows time passed from the daily challenge mode logic.

#### 3.2.1.4.22 ClassicModeScreen

This class extends base game screen. It has a classic mode logic property.

#### 3.2.1.4.23 SurvicalModeScreen

This class extends base game screen. It has a survival mode logic property. It shows time left from the survival mode logic.

#### 3.2.1.4.24 CubeIU

This class extends JPanel and will be organized to produce our custom cube panel. Its constructor takes a Cube instance, which is the logic part, as a parameter.

#### 3.2.1.4.25 CardUI

This class extends JPanel and will be organized to produce our custom card panel. Its constructor takes a Card instance, which is the logic part, as a parameter.

#### 3.2.1.4.26 GridUI

This class extends JPanel and will be organized to produce our custom grid panel. Its constructor takes a Grid instance, which is the logic part, as a parameter.

### 3.2.1.5 Storage Package

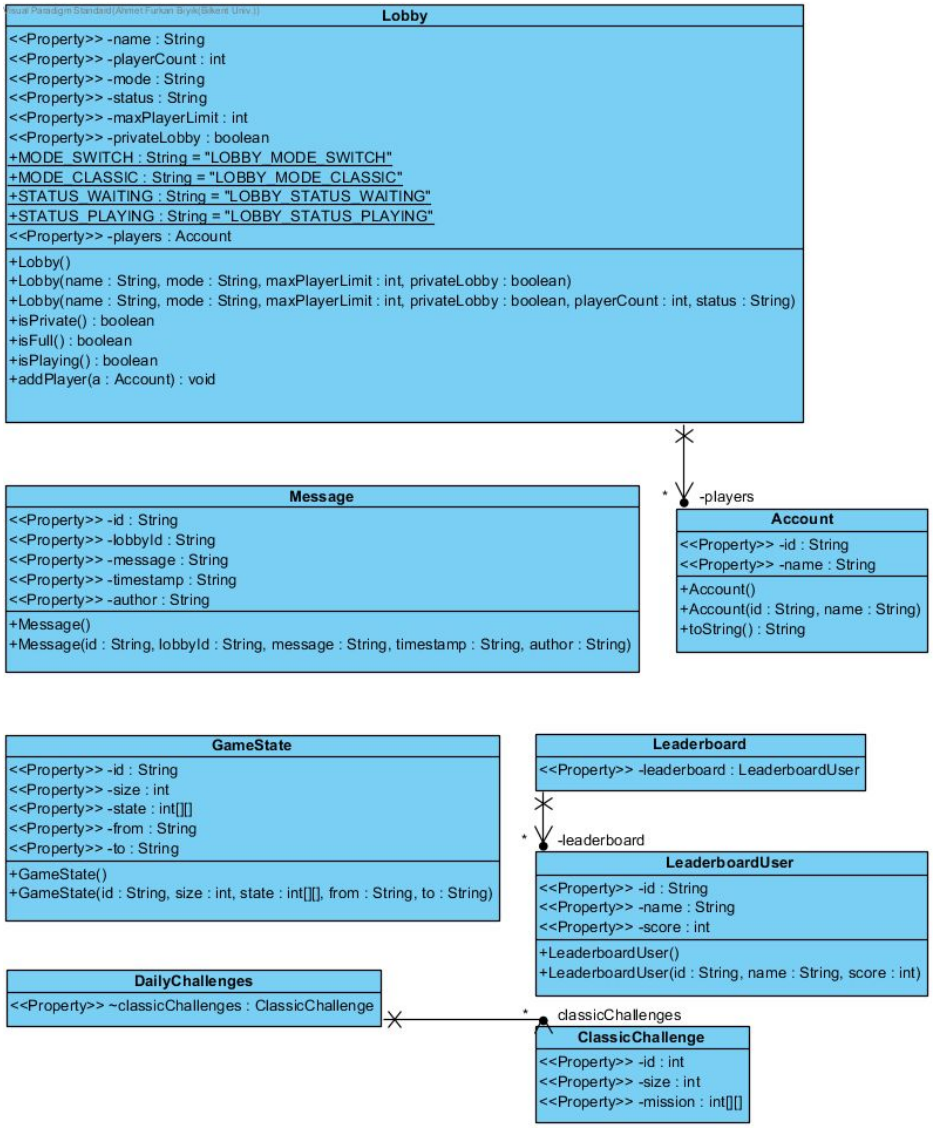



Figure 7

Storage package has the classes that needed for a multiplayer gameplay such as account or lobby. Its classes' properties are got from server and set when needed. For example, when player wanted to join a lobby, lobby information is got from server by sending ID of lobby. After the properties ol lobby set player can see the lobby.

### 3.2.1.6 Websocket Package

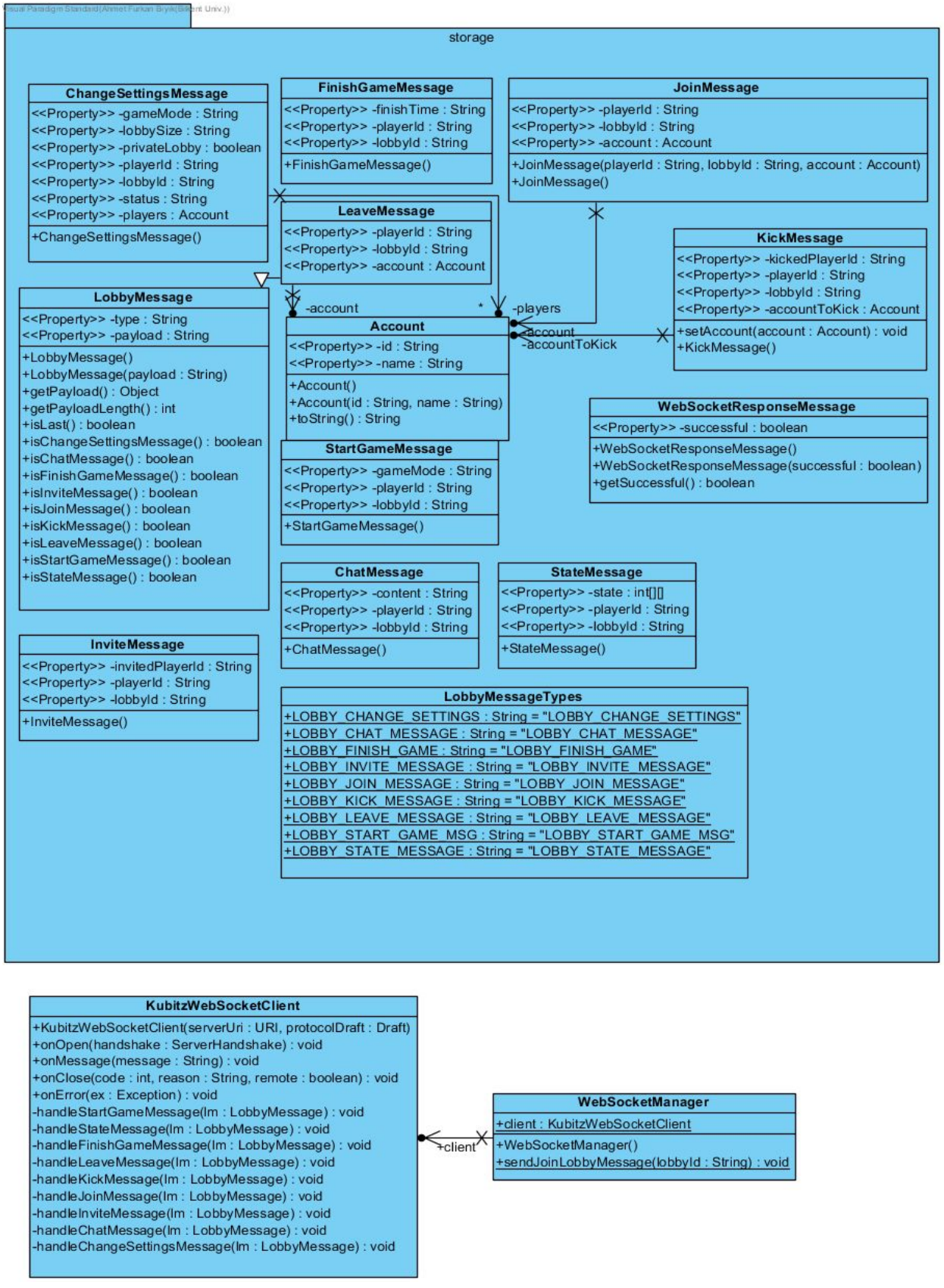



Figure 8

Websocket package is for websocket communication with server. When a change happens in other users or in server, server sends a notify message to client without client sends request to server. For instance, in the lobby players can chat each other. When a message in chat is sent by a player, server notifies the client and the client gets message. After get message manager updates chat and shows new message. It also has storage package. Storage package has classes to parse messages since messages are strings and they needed to parse to onjects.

### 3.2.2 Server Package

We have used Spring framework of Java to implement the server. The server configurations are specified in /src/main/resources/application.properties.

#### 3.2.2.1 KubitzServer

This is the main class which is obliged to start the server. It fetches the configurations from application.properties, which we have mentioned earlier.

### 3.2.2.1 Controllers Package

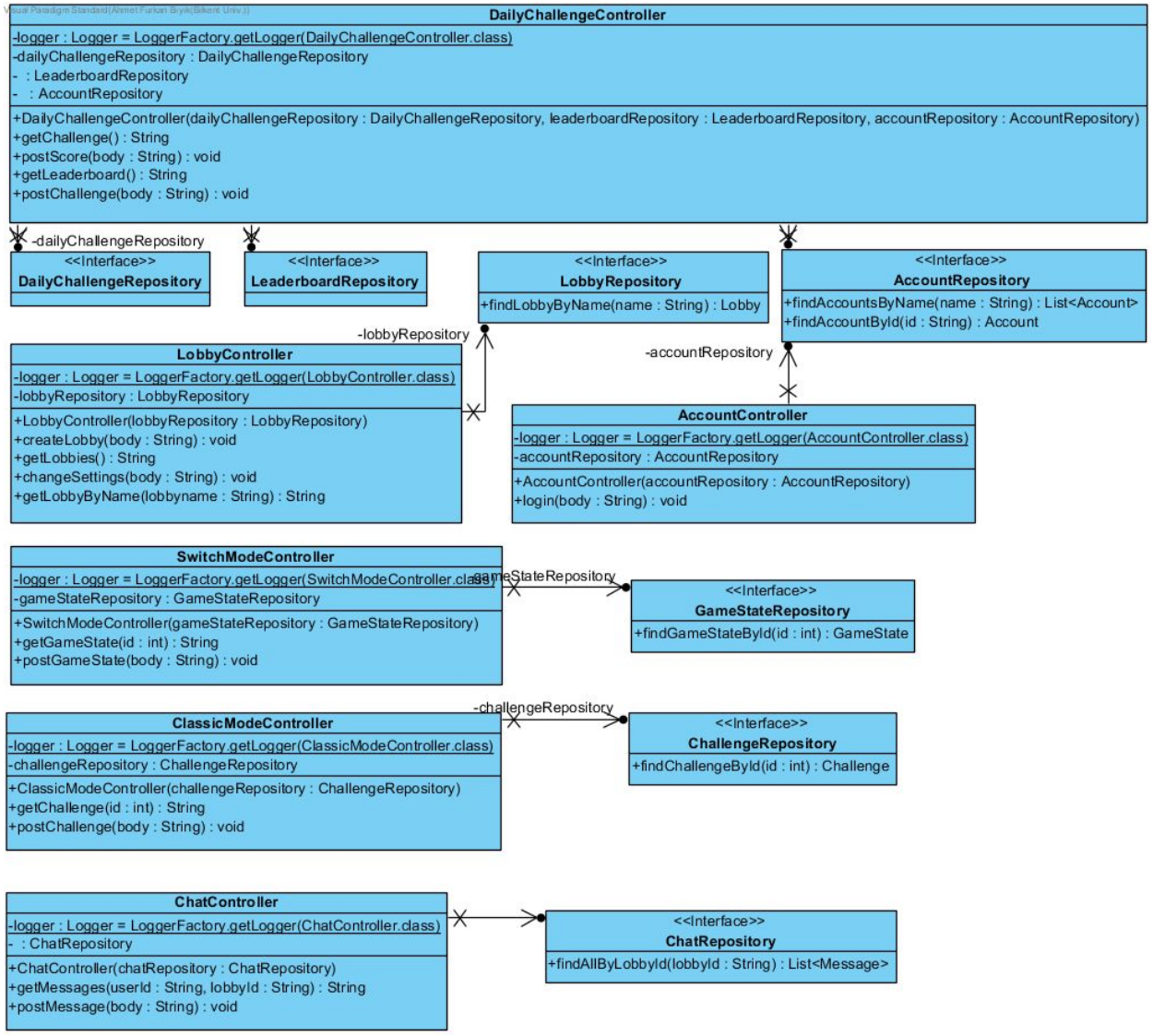



Figure 9

In this package, there are REST controller classes which are meant to be handling the communication between the players (clients).

#### 3.2.2.1.1 ChatController

There are two APIs in the ChatController. It controls the chatting feature in a lobby. One of the APIs is for posting a message and the other one is for fetching messages.

#### 3.2.2.1.2 SwitchModeController

SwitchModeController is for the management of the game mode Switch. There are two APIs: One is for fetching the game state of the opponent and the other is for posting the game state of the player.

#### 3.2.2.1.3 ClassicModeController

There is only one API in ClassicalModeController which provides the same challenge to all players in the lobby.

#### 3.2.2.1.4 DailyChallangeController

DailyChallengeController is for the management of the game mode Daily Challenge. There are three APIs: One is for getting leaderboard, the other is for posting the score of a player and the last one is for getting the challenge of the day.

#### 3.2.2.1.5 LobbyController

In the LobbyController, there are three APIs. One is for fetching lobbies, the other is for kicking player from the lobby and the last one is for changing the settings of the lobby.

#### 3.2.2.1.6 AccountController

It has the API that provides login and sign up feature for a player.

### 3.2.2.2 Database Package

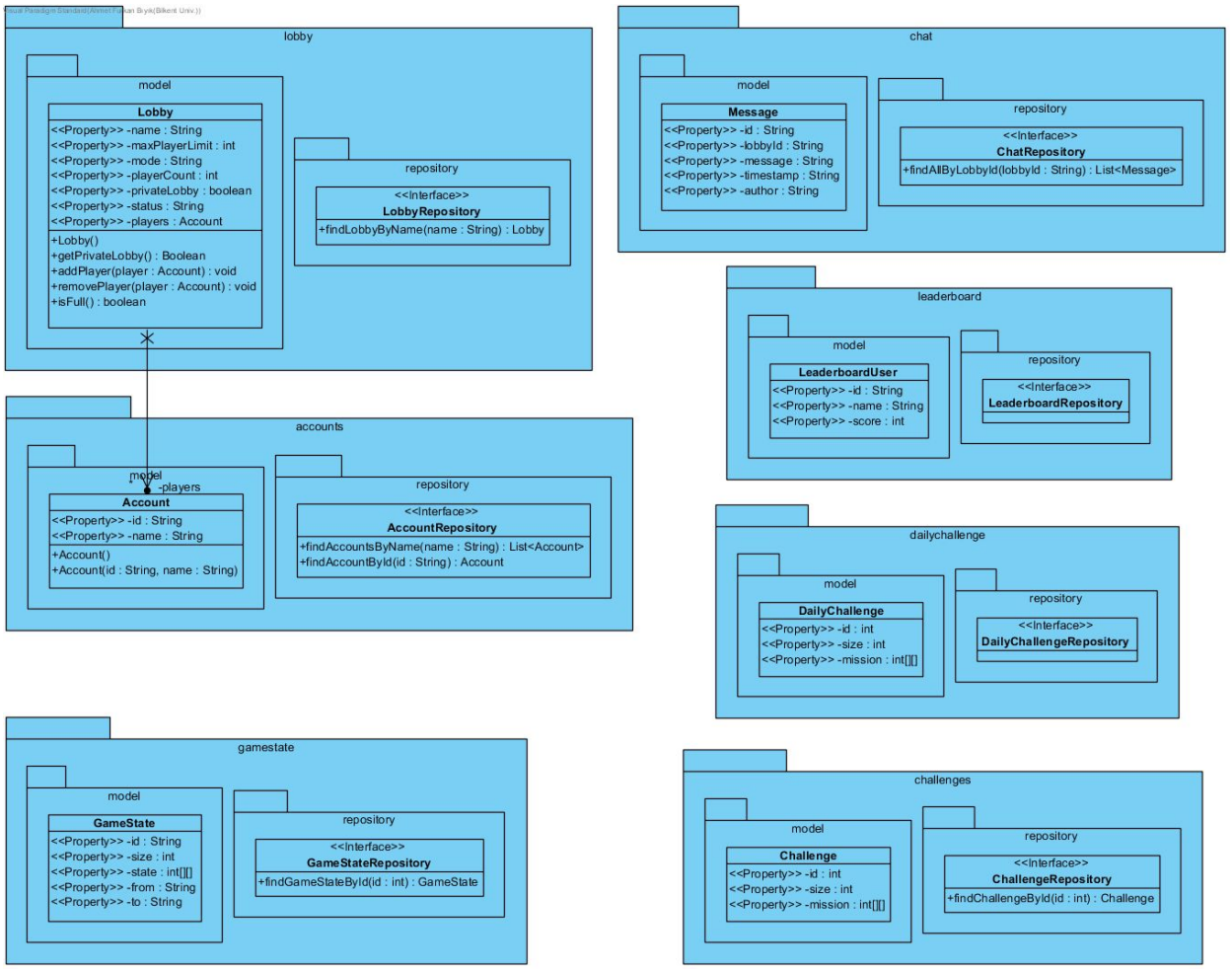
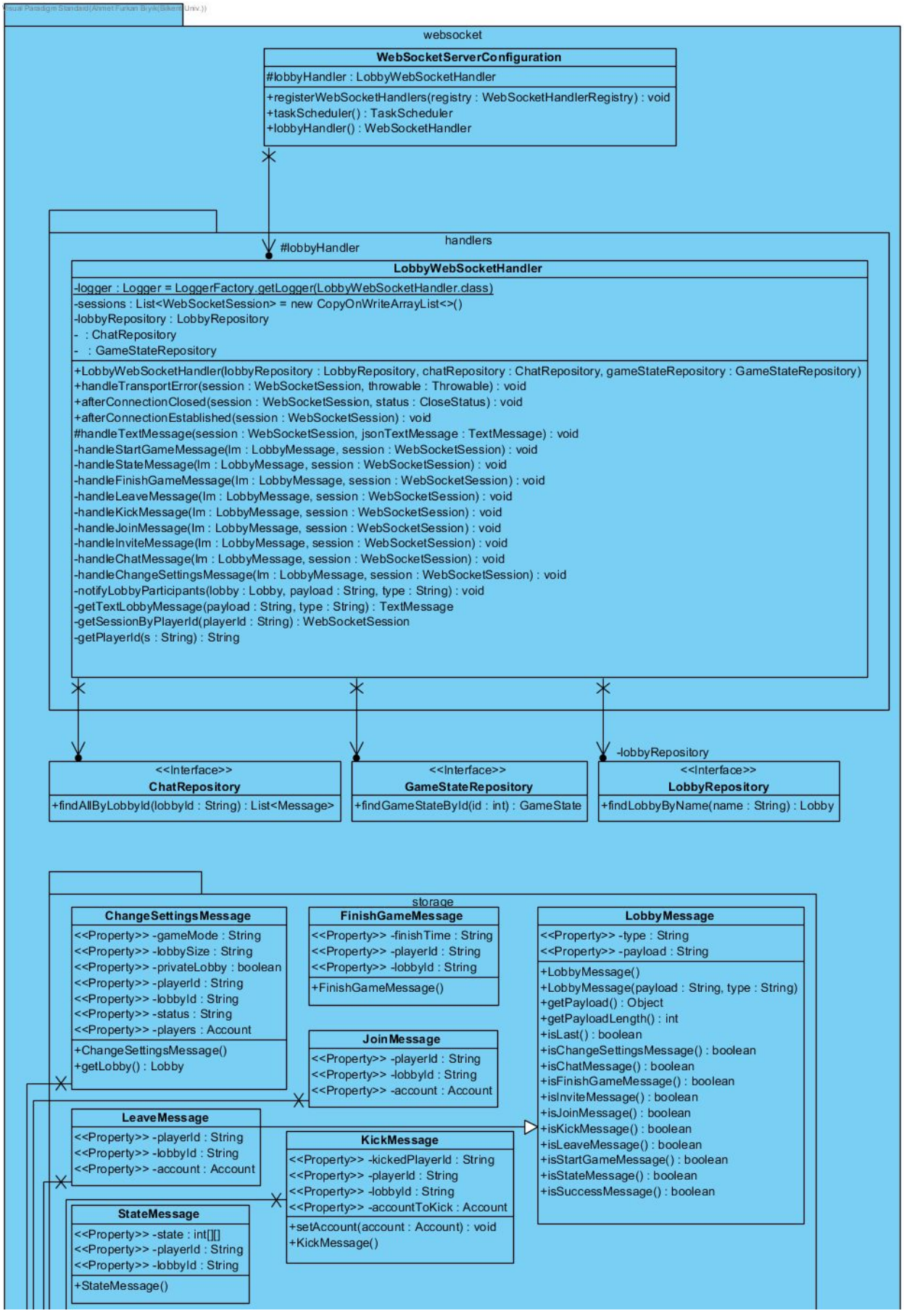



Figure 10

Database package basically communicates with the database. It mostly includes models and repositories. Models are the object types that will be written to the database. Models are also indicating the collections and repository saves to database according to that collection.

### 3.2.2.3 Websocket Package



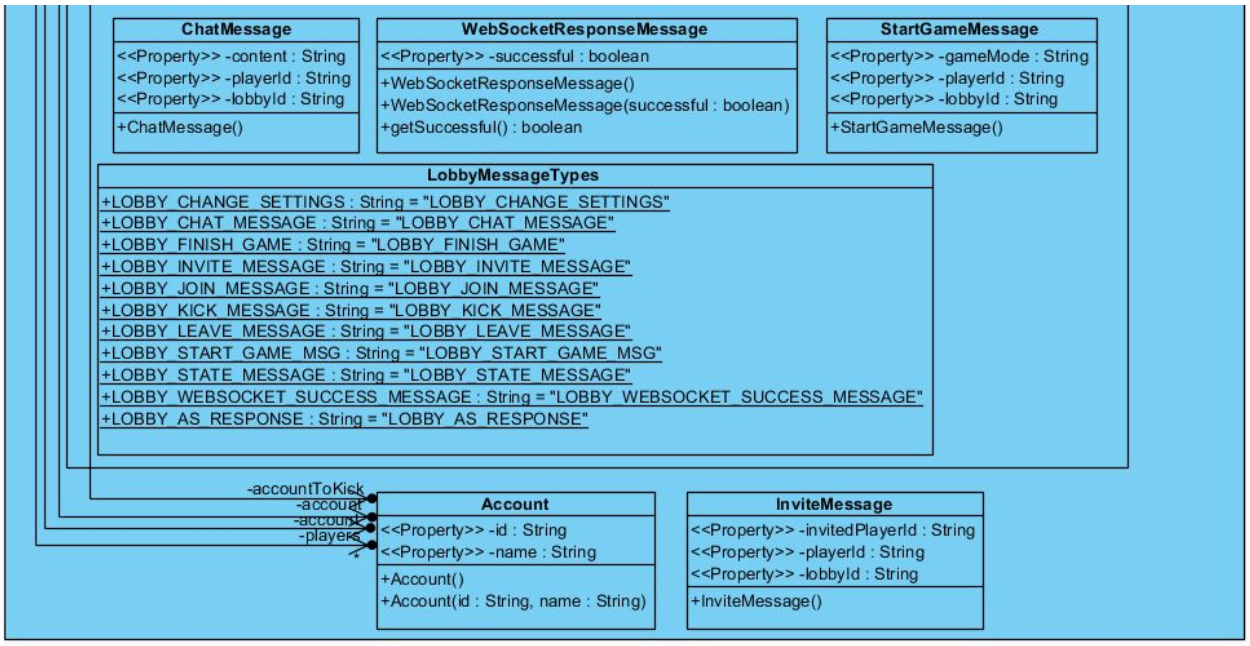



Figure 11

Websocket package communicates with client for notify client. For instance, in the lobby players can chat each other. When a message in chat is sent by a player, server notifies other players in the lobby and sends message. Websocket package has also storage package which is for parsing messages since they are sent by as strings and they should be parsed to objects.

# 4 Improvements & Summary

We have redesigned the graphical user interface package hierarchy. We used to manage the screen flow with CardLayout Java API using content pane. We have build ourselves a screen manager system for satisfying our specific needs which uses stack data structure internally. We have added lifecycle system for GUI elements.

On the other hand, besides the RESTful service, we have added WebSocket integration to improve the communication between the client and server since WebSocket provides two way communication.

Hence, we are mostly consistent with our previous design. The above two major changes

are done to satisfy the needs of the system including non-functional ones. For instance, the WebSocket system provides smooth flow of messages in the lobby and also in the invite system of lobby. Additionally, the new screen system made the code easily extendable and manageable.

# 5 References


https://www.azul.com/4-reasons-java-still-1/

http://springtutorials.com/twelve-reasons-to-use-spring-framework/

https://maven.apache.org/what-is-maven.html

https://www.johndcook.com/blog/2013/05/10/efficiency

-vs-robustness/