Bilkent University

Department of Computer Engineering

CS 319 Course Project

Group: 2C

Design Report

- Furkan Kazım AKKURT
- Abdullah Can ALPAY
- Murat ANGIN
- Ümit Yiğit BAŞARAN
- Muhammed Emre YILDIZ

Instructor: Eray TÜZÜN

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

1. Introduction

1.1 Purpose of the System

Monopoly is a strategy and luck-based board game where players try to compete with each other by buying, selling, and renting properties. The design that is thought during the implementation differs from the original game in several ways such as having an additional customizable board feature. Within this feature, the aim is to provide users with a more enjoyable and satisfactory game. Monopoly is thought to be a fun game by implementing these additional features and involving players in a different environment than the standard version of the game. The goal in the game is to make other players declare bankruptcy. The design goal is to provide a fast, responsive, and good looking user interface with a backend which works fast and uses low memory and power consumption.

1.2 Design Goals

Obviously as a game, the main goal is to entertain players and provide them with fun moments. In order to make sure that it is happening, it is required to have some kind of optimizations and modifications such as using design patterns while implementing the game. This subsection is about the design goals that are identified during the project and the trade-offs that are faced in order to maintain those goals.

1.2.1 End User Criteria

Usability: In order for a new user to understand the user interface and navigate between different menus, the user interface must be understood easily by the user. To make the interface more usable and understandable, it is important to consider that the user profile may be ranging from children to elderly people, such as in Candy Crush, which is a mobile

game played by both children and adults [1]. Thus, the design will be in such a way that all the users will not have to deal with complex interfaces and complex game controls, and users will be offered to go over a tutorial game so as to understand the game logic and controls.

Performance: In order to make our game more enjoyable, it is designed online and we used PIXI renderer in order to make the game faster. Also, some libraries are used to make monopoly faster and efficient.

1.2.2 Maintenance Criteria

Extendibility: All features of the game are designed as functions so with respect to user feedback our game can be updated easily. For example, the skills of the characters can be changed with respect to end-users feedback to make the game more balanced. Also by these feedbacks, some default maps can be added.

Portability: The game will be online, therefore our game can be played everywhere. A computer and internet connection is enough. That is why our game is portable.

Reusability: Considering some subsystems regarding the GUI part of the game, some of them can be reused in some different projects easily just by making appropriate changes. For instance, the GUI subsystem including the main menu, options, and etc. can easily be used in different games without needing to modify the subsystem. To do so, it is aimed to design the subsystems independent from the system.

1.2.3 Performance Criteria

Response Time: We want to create our game using the electron framework in order to minimize the response time and process user requests as fast and accurate as possible.

1.2.4 Trade-Offs

Understandability vs Functionality:

Monopoly is one of the most played board games on the planet. Everyone knows the general rules of the Monopoly which are simple buy-sell or build operations. That's why no one will have the difficulty to understand the basics of the game. However, since Monopoly is a simple game, we decided to add some additional rules and features. Our additional rules make the game more complex and present different win opportunities. For instance, we add additional chance cards, such as Natural Disaster Cards, Bluff Cards etc. to increase the game functionality and possibilities. Similarly, we add additional buildings to increase profits of the lands. We add different characters with different special skills to make the game more complex and more fun. Rather than just buy-sell, in our game players can build up different strategies to bankrupt other players or earn more profit. That's why our game has lots of different functionalities and various extra rules, thus, it is hard to keep up with the additional functionalities on the first two games. After that, like any other game, our game can be understood completely.

Memory vs Maintainability:

During the design of the game, we have two different implemented parts in the game. In the first part, we have the well-known Monopoly game in which players buy/sell to collect the same groups to increase their profit and build different buildings. In this part, we can minimize the memory usage and while minimizing, we can maximize the maintainability. We do not require additional memory space rather than the memory required for players (which holds the assets for a single player), lands and buildings. When we come to the second part, the additional features that we plan to implement, we require additional memory space. Each player will have different characters which have different skills and will have different items, buildings (new buildings) and special cards; and different functions which dynamically

allocate extra memory space when called (such as calling penalty function which returns the penalty fee considering game status).

Since we have two different parts that needed to be implemented and we needed to work faster. We believe that using Javascript will increase the speed of our game. We use Node.js and React.js to increase the efficiency and the speed of the code. Hence, our program will be fast, our design goal will move to less memory usage and increase maintainability. To accomplish that, we focused on using the same object via it's id. To mean that, for instance, when a player draws a special card, like Bluff Card, this card was already created when players started the game. When the card is drawn and becomes an asset of the player, player class only knows the id of the card. When the player uses that card, the id is deleted. Thus, rather than using the object itself, using integers (id) will decrease the memory usage. To maintainability, we will design our program with utilities. Therefore, our server only focuses on the relation between the classes (utilities, such as cards, buildings, players) and with the client interface. With that it will be easy to understand the code, the classes and the functions. Along with that, the changes or updates can be done in one class which will be beneficial during coding.

Development time vs User Experience:

Since we use Javascript, our main coding environment will consist of Node.js and React.js. Additionally, we need a database to store assets of the players and cards, thus, we choose to use the Heroku cloud application platform. Since we use Javascript, our game will be played on the browser. Therefore, the load speed will depend on the player's internet speed. Since it will be displayed on the browser, our game will give a simple yet fast user experience thanks to React.js. On the server side, we use Node.js also supports the faster user experience. Thus, our game offers simple yet faster user experience.

Moreover, even though we stated the game will be a simple experience; we also planned to implement additional animations. These animations can be in game animations,

such as rolling dice, or moving the pawn; or it can be outside animations such as decorative animations, according to the board template. We have more than enough libraries and documentation for implementing the front end in React.js. Hence we believe that once we learn the basics, we believe that implementing front end will be easy. The same is true for Node.js too.

2. System Architecture

2.1 Subsystem Decomposition

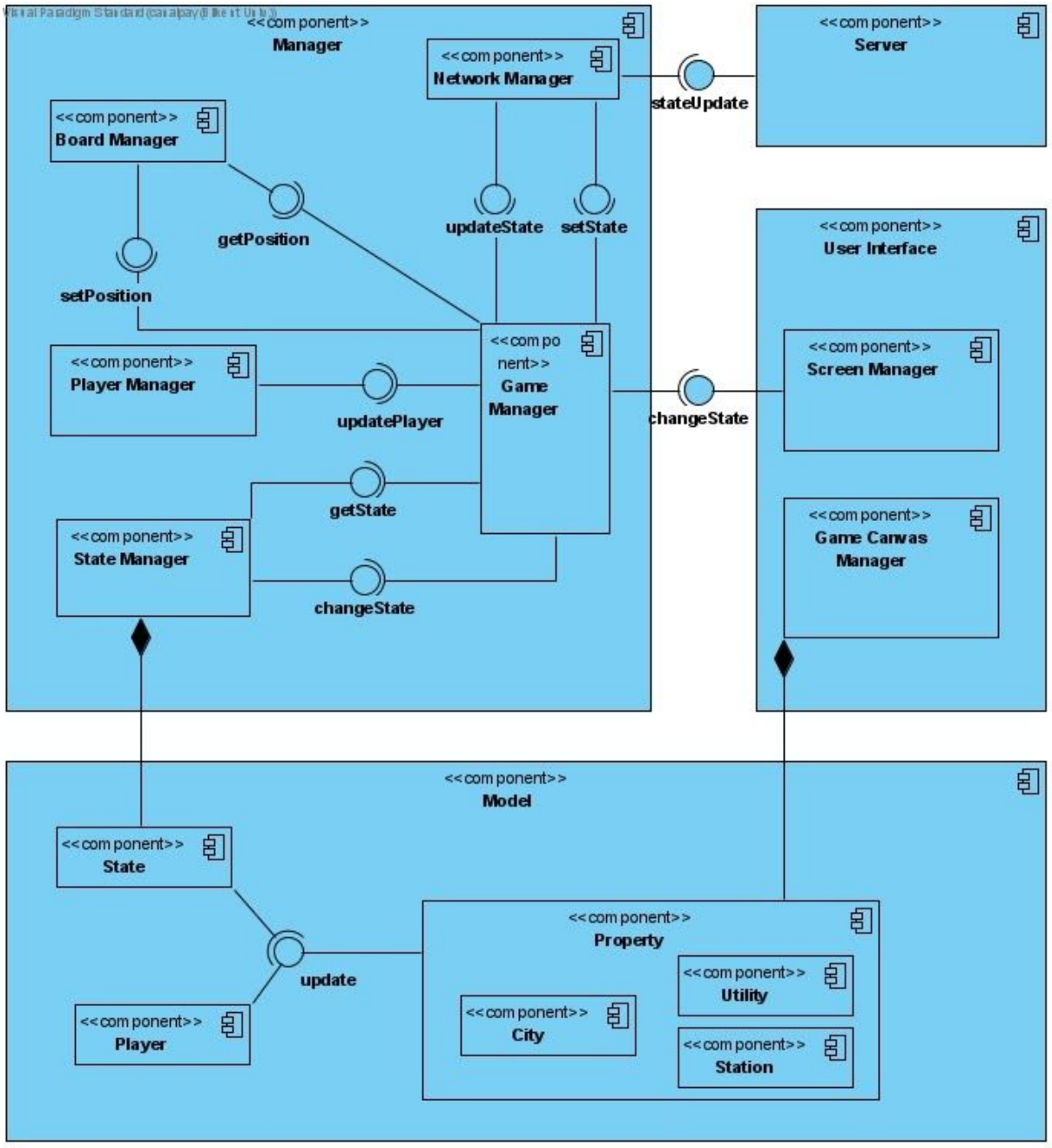

Figure 1 - Subsystem Decomposition

To analyze the program more effectively, the system is divided into three subsystems by choosing the principle of Model, Control, Viewer (MCV). The reason why MCV is the best

choice for decomposing the program is that the program consists of 4 subcomponents. 3 subcomponents are used for Modelling the program, Viewing the program and Controlling the program state. Final subsystem is used for only updating the game state according to the other players, since the game is an online game. Thus, when we consider the program without the online side we can see the MVC subsystem modelling. MVC architecture provides us high cohesion and low coupling.

For Model Architecture: The Model component is used. The player's properties, all the city, utility and station components are located into the Model component.

For View Architecture: The User Interface component is used as View architecture. The game screen is displayed, such as the board, cities, through the User Interface component. The player's interactions with the program are sent to the Controller component.

For Controller Architecture: The Manager component is used as Controller architecture. The game is decided -such as rules and game turns- and played in the Manager component and updates the Model subsystem. The player's inputs are sent to the Controller subsystem by User Interface subsystem and updated the Model subsystem.

2.2 Hardware/Software Mapping

Monopoly will be implemented in Javascript programming language with different frameworks such as ReactJS, ElectronJS and SocketIO. ElectronJS creates a bundle with all the dependencies in it therefore there will be no additional installation or library required.

Monopoly will require a keyboard and mouse as a hardware requirement. They will be used in the Input management system, User can control the menu or general UI and interact with the game via these external tools.

2.3 Persistent Data Management

Monopoly doesn't require a complex database system, we will store custom boards, user preferences and user credentials if needed in custom text files. Therefore we can say that there is no database system in Monopoly, we will implement a custom basic text based data storage system with the help of NodeJS.

2.4 Access Control and Security

Monopoly will use a socket communication system in order to handle multiplayer systems and chat systems. We will code different channels in our NodeJS server in order to listen to packages coming from clients, and most of the data sended will be only the state of the current game. There will be no critical data flow happening here. Therefore the default security rules and systems for websocket implementation and NodeJS runtime is enough for our case.

2.5 Boundary Conditions

Monopoly will not require an installation process, it will have an executable for different platforms specifically, an exe file for Windows, an Application file for Mac and an executable file for Linux systems.

Users can easily terminate the game by clicking the "Exit" button in the main menu. Also it is possible to kill the process directly externally but we do not recommend it because it negatively affects multiplayer experience for other players.

3. Subsystem Services

User Interface Subsystem

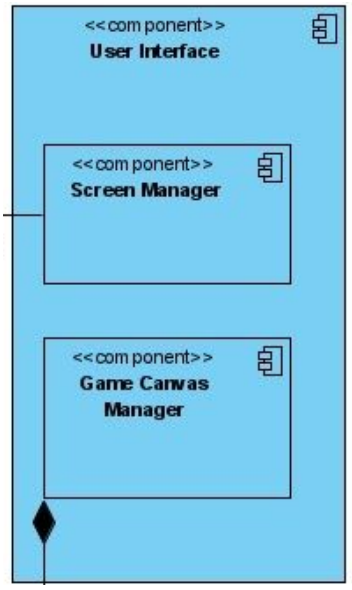

Figure 2 - Detailed User Interface Subsystem

Screen Manager

Screen Manager which will be written using React (“a JavaScript library for building user interfaces” [2]) consists of 24 different classes.

@material-ui/core, @material-ui/icons, and formik packages will be used to create the Screen Manager as dynamically as possible. @material-ui/core and @material-ui/icons packages are predefined component libraries [8, 9]. formik package will be used in forms to reduce the boilerplate of forms in general [10].

Screen Manager consists of one main render class called “ReactRoot” and seven different pages which are rendered in control of ReactRoot class.

Game Canvas Manager

Game Canvas Manager is used for drawing game features that users will see. There are 3 main classes in this system: Card, Board and Tile. Card class is for all cards in the monopoly which are city cards and usable cards such as “exit from the jail.”. With the Game Canvas Manager, these cards are drawn with the properties which are coming from the Model subsystem, and players can see and move his/her cards. In board class, the board is drawn by using tile classes that also contain a model class with respect to their type. All different tiles are given as a class even if it has no mission such as ParkTile. However, the board must include all tiles and some card position which is for community chest cards and chance cards. Moreover, when a player buys a tile, they can see every feature of the city on the card because of the changes reflected to view coming from models and move their city card with their cursor. In addition, usable cards are created off position because players should not see them. When a player comes to chance tile and takes a chance card, a usable card opens and its description becomes visible.

Manager Subsystem

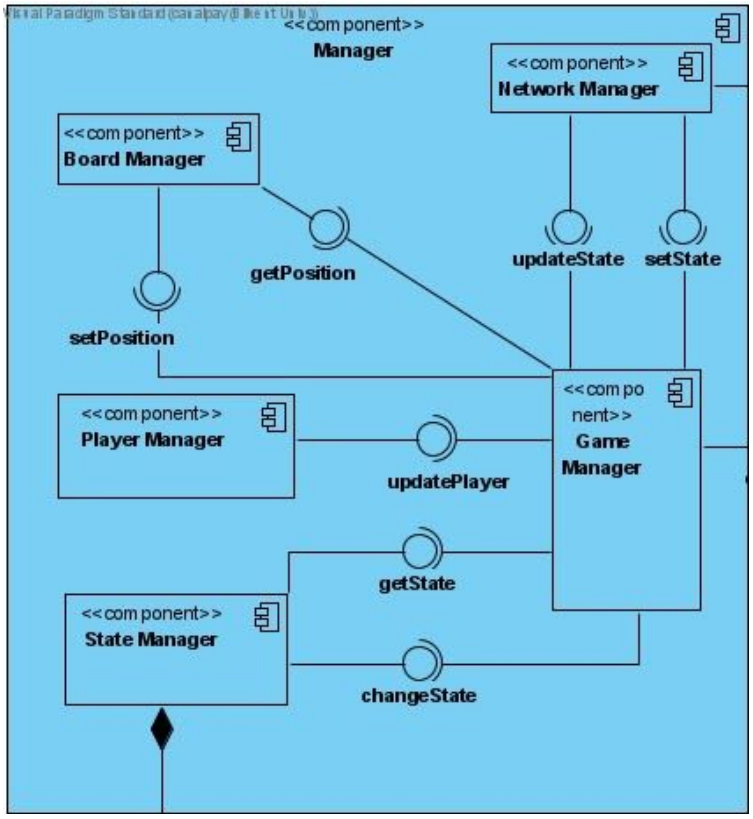

Figure 3 - Manager Subsystem

Manager Subsystem consists of all the managers of Monopoly Game. This subsystem is the starting point of the game screen and contains all the flow starting from the logic until the view render part. The main manager is GameManager in this subsystem. It can communicate with other managers in order to maintain a successful game flow until the end of the game. It uses different interfaces in order to communicate with both other managers and the View subsystem. The data flow to the models happens with the help of the State Manager. However, the changes on State Manager controlled by Game Manager, only the connection with Model subsystem handled by State Manager. Also the same architecture between State Manager and Game Manager happens in Network Manager too, the only difference is the listener outside of the system. In Network Manager this listener is Server Subsystem.

Model Subsystem

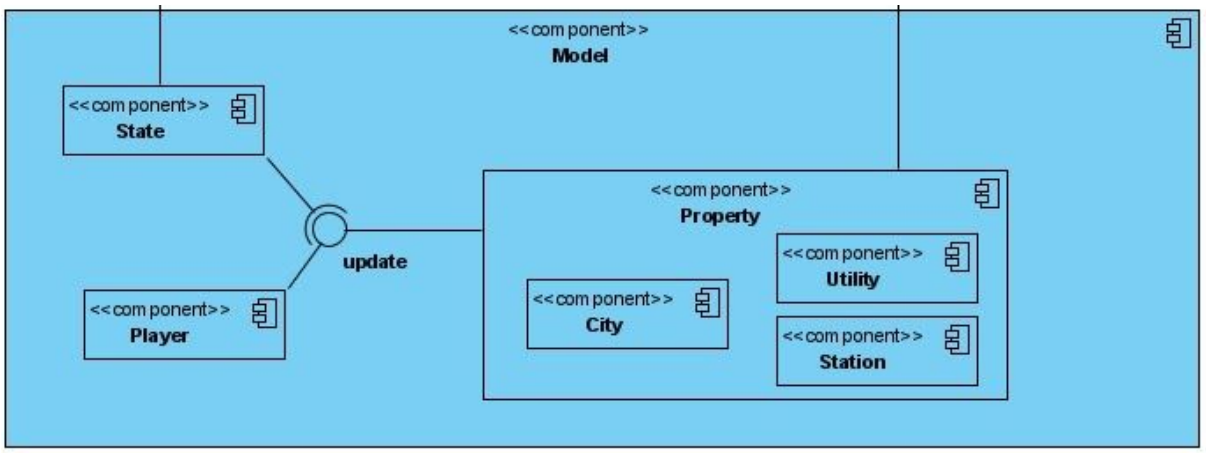

Figure 4 - Model Subsystem

Game Object Subsystem is the subsystem where all the classes that are included in the Model part of the Model-View-Controller (MVC) architecture. There are nine classes in the subsystem in which three of them are associated with the Pixi UI Subsystem. There are four main classes which are City, Player, Property, and Building. Cities come together and create CityGroups according to their colors. Cities and properties are owned by players. Buildings are constructed on top of cities, which are owned by players. There are also Utility, UsableCard, and Station classes which are used for specifying the Tile types. Since Tile class and its children are represented in UI Pixi Subsystem, they are associated with that subsystem.

Server Subsystem

Figure 5 - Server Subsystem

The game is an online game, therefore, the Server subsystem is the fundamental communication system in the game. During the game users need to communicate each. They need chat functionality and state synchronization to accomplish all player interactions with the program and each other. To accomplish that, we use socket.io library which is an implementation of websockets. Server subsystem handles the state changes between clients in order to provide a healthy communication and gameplay. This subsystem will be used by Network Manager in Manager subsystem in order to communicate with other systems such as View Subsystem and Model Subsystem. The needed state objects for synchronizations will be factored in the Manager subsystem(State Manager). Server subsystems have different roles besides in game state management, it also offers us a room system with a chat subsystem. With the help of Room and chat system users can easily play multiplayer versions of Monopoly games. The functionality of these systems is also provided by the Server subsystem.

4. Low-level Design

4.1 Design Patterns

Our implementation of the Monopoly game uses several Design Patterns in order to reduce the issues related to software development during the implementation process and make objects communicate with each other more efficiently and easily. Design patterns allowed us to use objects more efficiently and prevented “code smell.”

4.1.1 Facade Design Pattern

Facade is a Structural Design Pattern meaning that its job is to define ways to compose objects and obtain new functionalities. It provides a unified interface to a set of interfaces in a subsystem, which is making the subsystem easier to use. To exemplify, in our case the GameManager class holds instances to all other Manager classes, such as CardManager,

EventManager, or NetworkManager. All the job that is needed is done using a unified interface, which is GameManager in this case, to a set of interfaces, which are other Manager classes that are managed by GameManager class.

4.1.2 Singleton Design Pattern

Singleton Design Pattern is one of the Creational Design Patterns, which are used to manage class instantiation mechanisms. Singleton is used to ensure that a class has only one instance during the runtime. It is used if a working application needs only one instance of an object. This pattern is also used in Manager objects. Since each Manager class requires one and only one instance during the runtime, Singleton Design Pattern is used. Also since Facade objects are often Singletons, because only one Facade object is required, it is expected to have Singleton Design Pattern at the same time.

4.1.3 Observer Design Pattern

Observer is one of the Behavioral Design Patterns. It is used to change objects' behaviors in case behavior of an object changes that is associated with the other objects. This means that, for instance, when the state of the game is changed in State class, all classes that are connected to the State class will have to be updated. In addition to that, Observer Design Pattern is also used in Model-View-Controller architecture. This pattern is the View part of MVC, meaning that a change in View part has to be notified to Model and Controller parts. This is handled using Observer Design Pattern.

4.2 Final Object Design

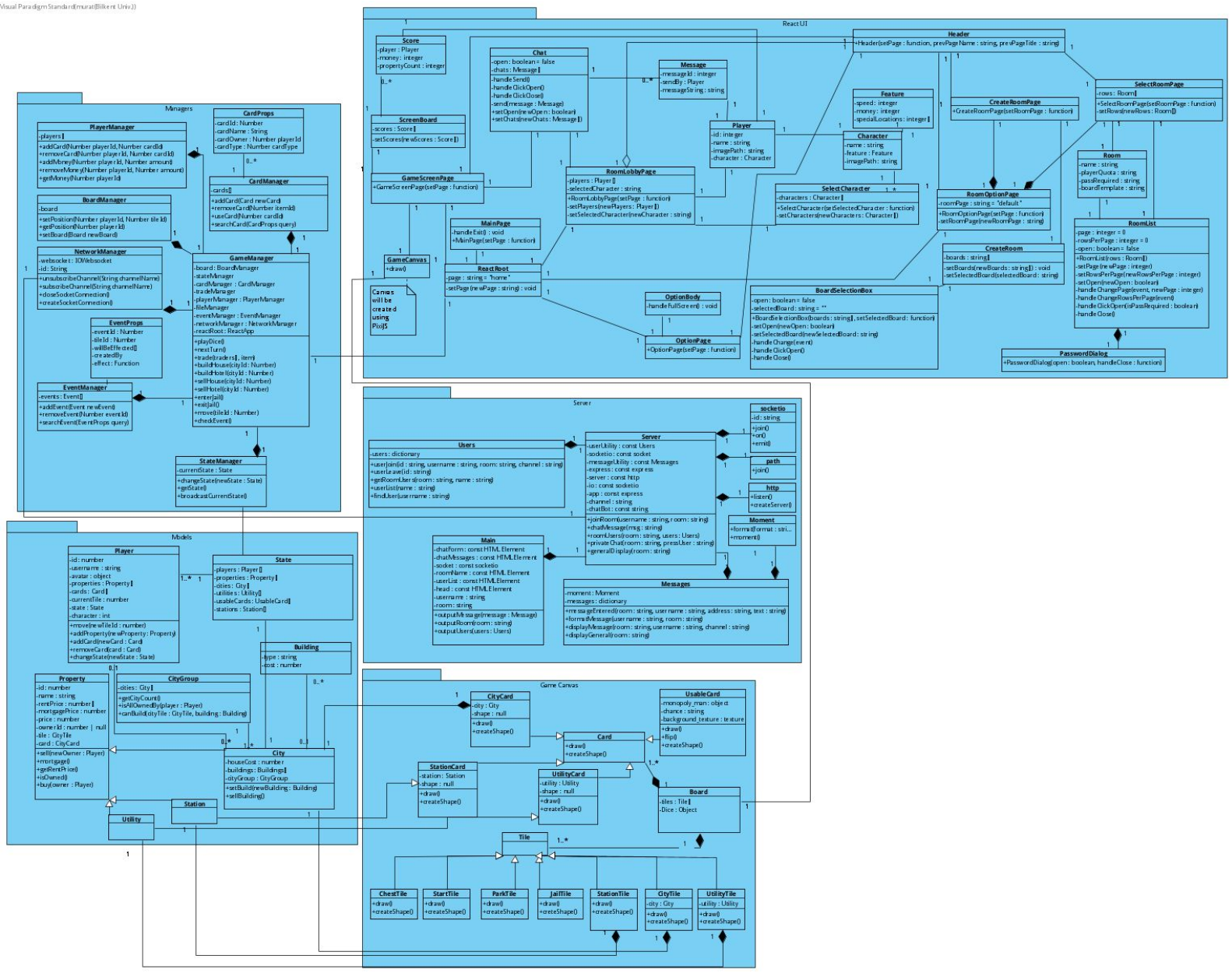

Figure 6 - Final Object Design

You can see the whole final object class diagram, in order to analyze it closely you can refer to next images which divided different parts of the object class diagram in to seperate images.

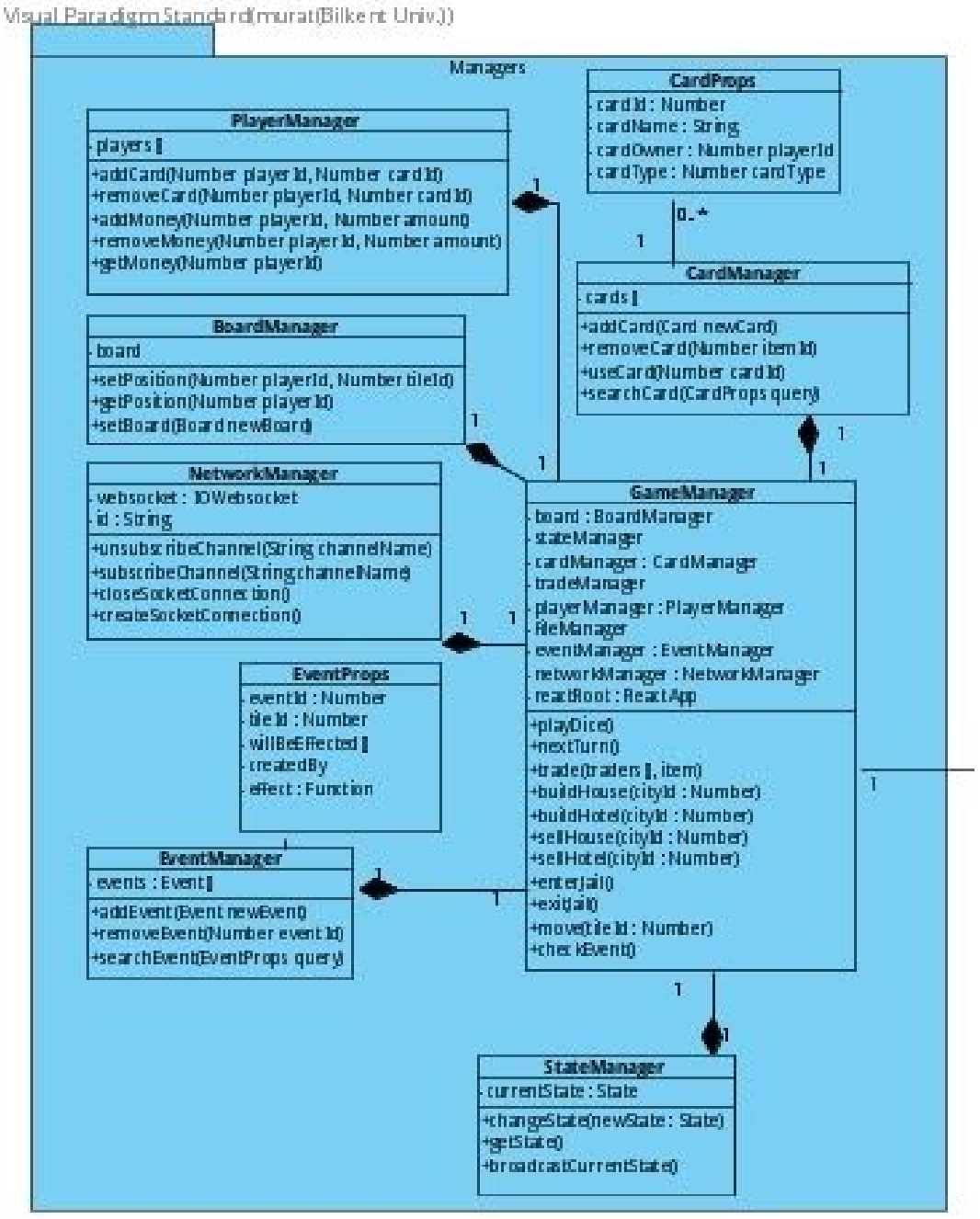

Figure 7 - Managers Package

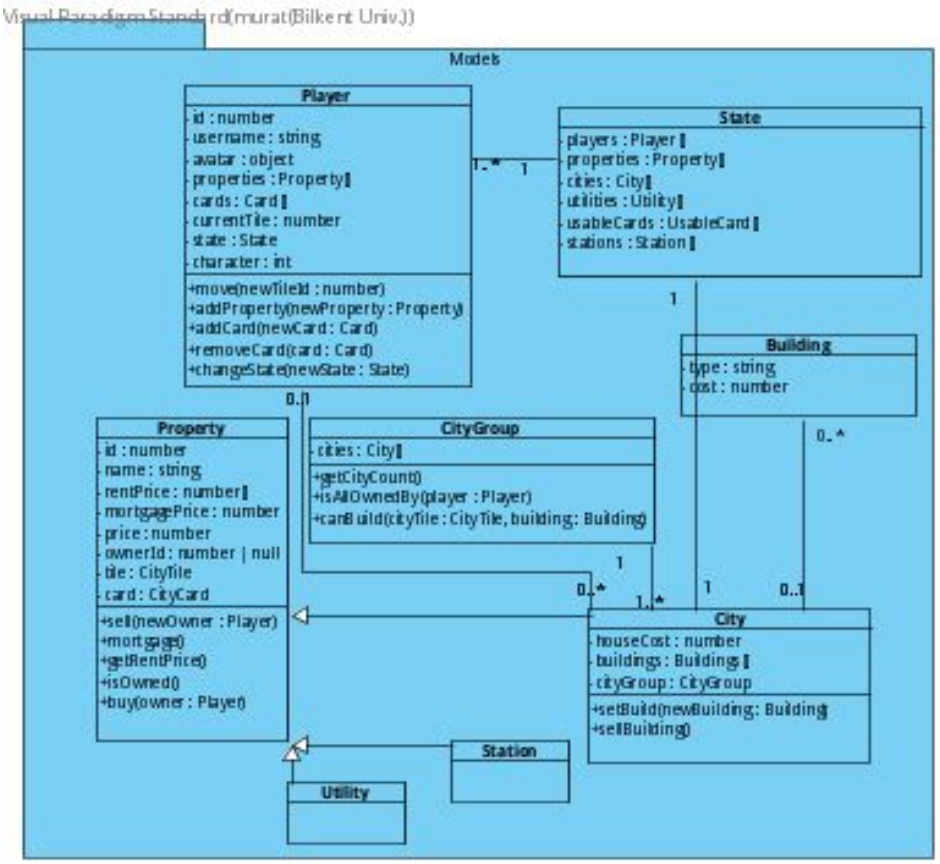

Figure 8 - Models Package

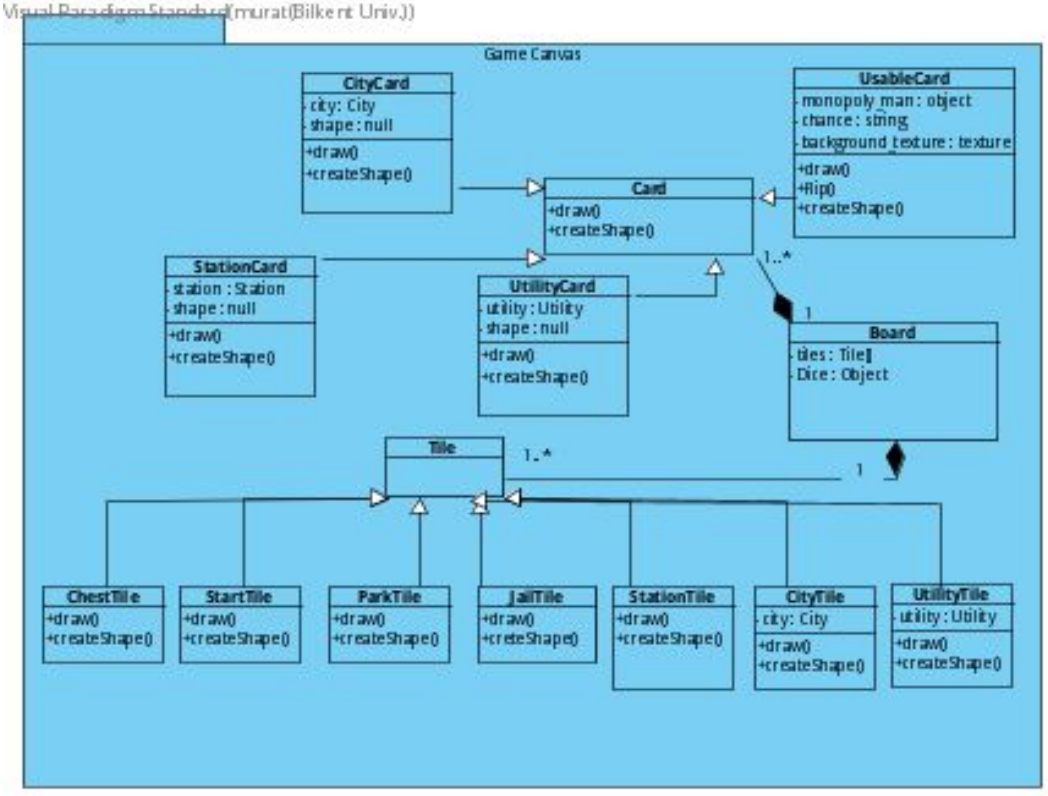

Figure 9 - Game Canvas Package

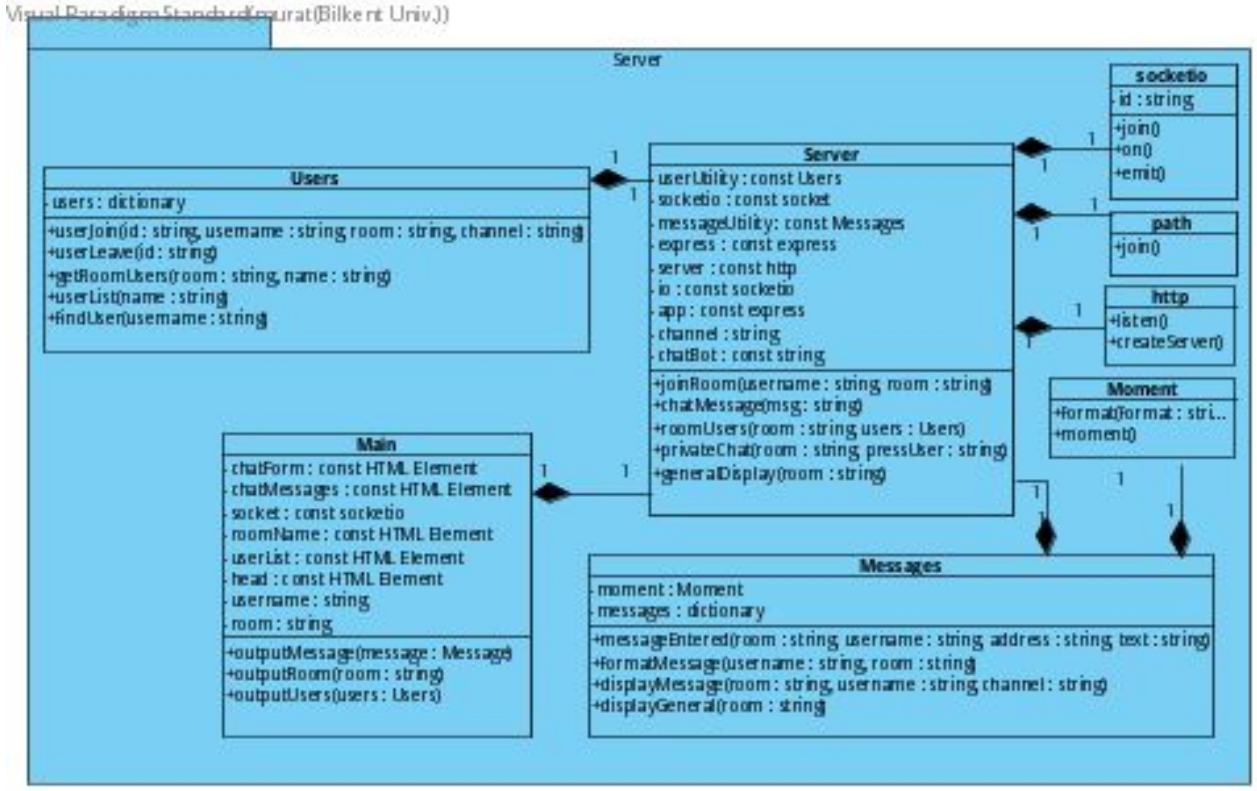

Figure 10 - Server Package

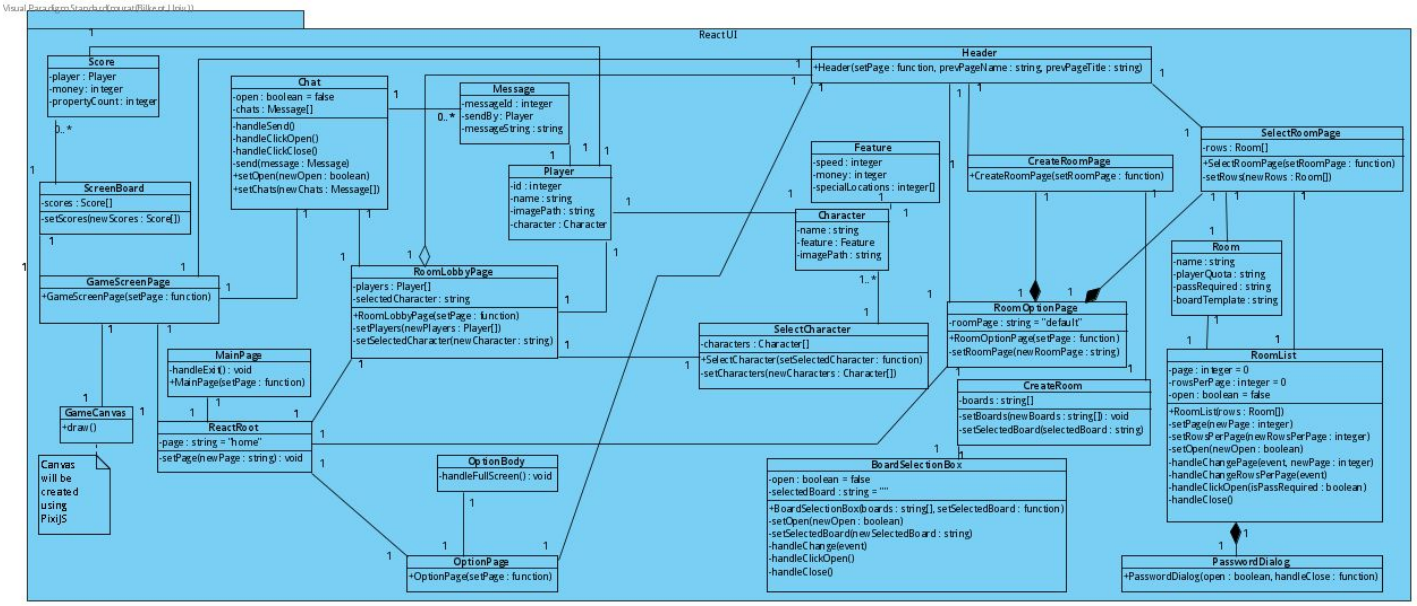

Figure 11 - General React Package

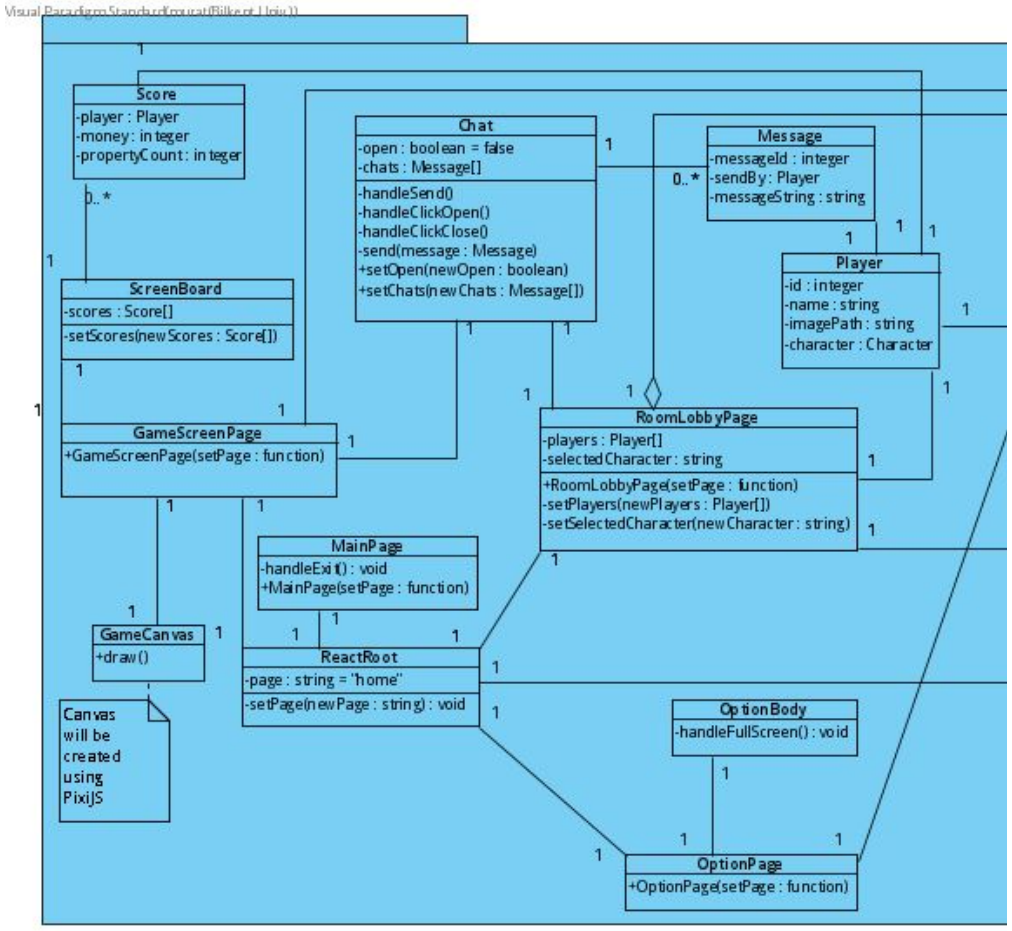

Figure 12 - React Package 1st Part

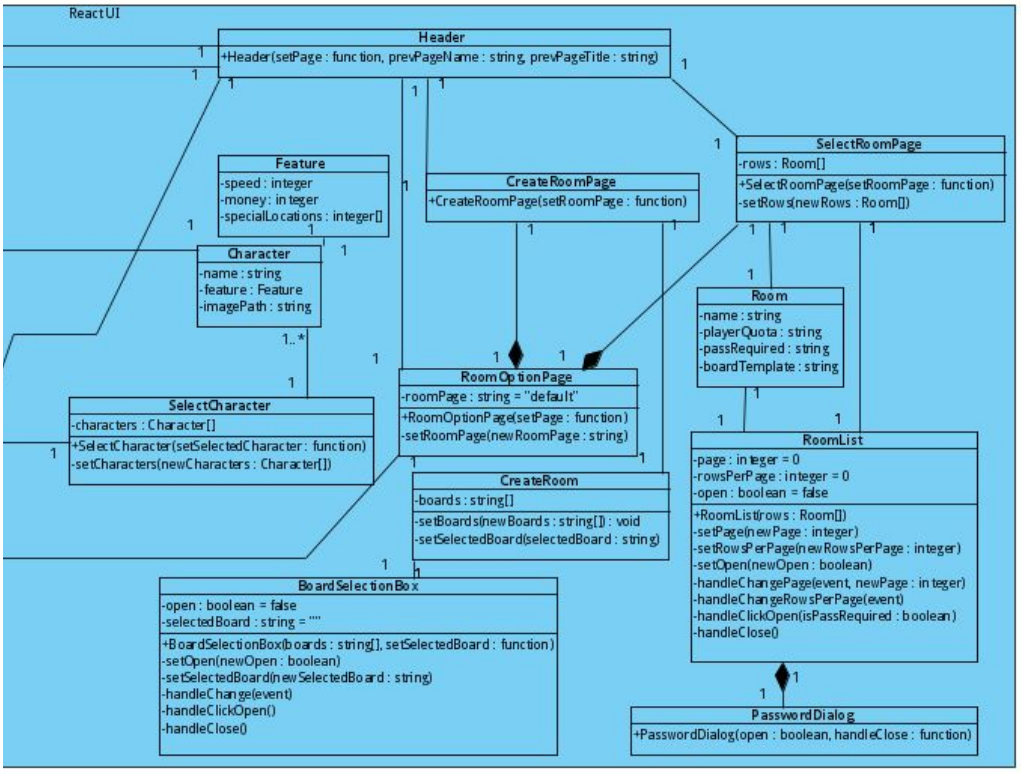

Figure 13 - React Part 2nd Part

4.3 Class Descriptions

ReactRoot Class

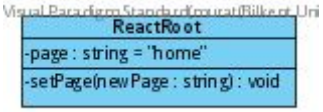

Figure 14 - ReactRoot Class

Attribute:

- **private page: string = “home”:** This attribute is used for selecting the page that will be rendered.

Method:

- **private setPage(newPage : string) : void:** This method sets the page state to the parameter “newPage”.

MainPage Class

Figure 15 - MainPage Class

This class provides users the ability to create a new game, navigate to the options menu, or exit game.

Constructor:

- **public MainPage(setPage: function):** This is a constructor for the MainPage class, it takes the setPage function of the App class to set the page to the desired string when buttons are pressed. When users want to start the game it navigates the page to “roomOptionPage”. When users want to adjust something about the game then thi class navigates the page to “optionPage”.

Method:

- **private handleExit() : void:** This method closes the application window. It is called when the user wants to exit.

OptionPage Class and Its Dependent Classes

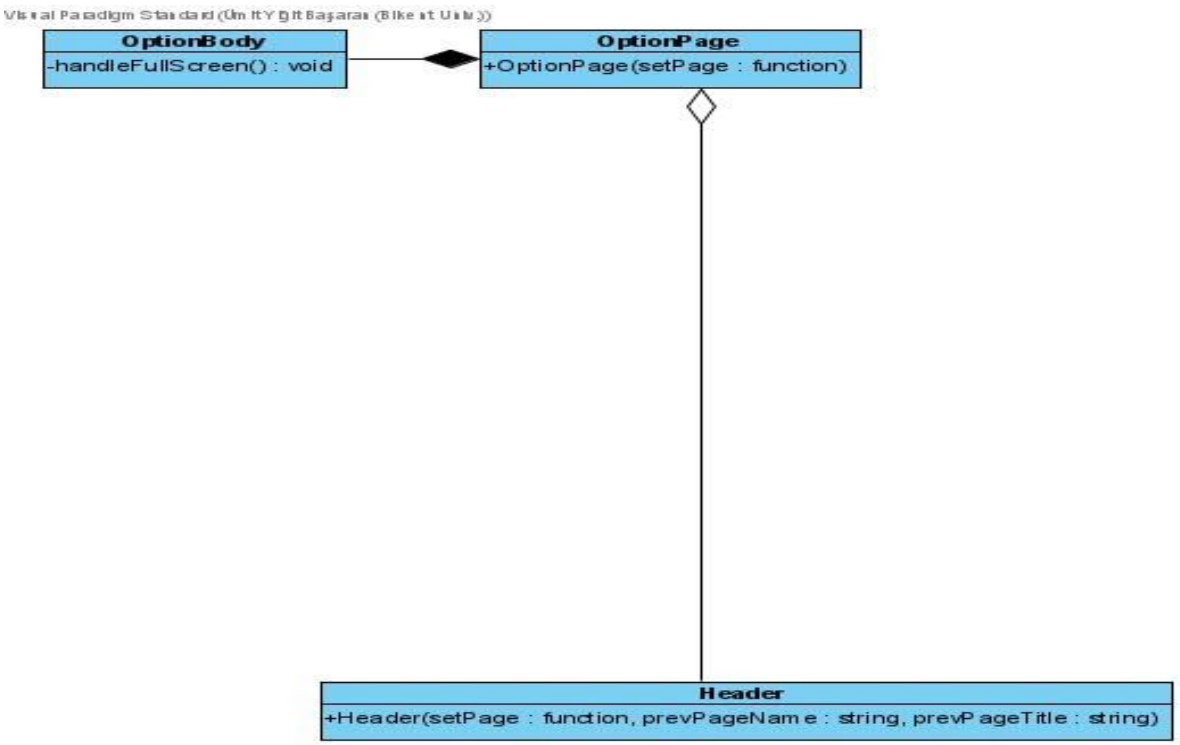

Figure 16 - OptionPage Class and Its Dependent Classes

OptionPage Class

OptionPage class consists of the Header and the OptionBody classes.

Constructor:

- **public OptionPage(setPage : function):** This is a constructor for the OptionPage class. It takes the setPage function as a parameter from App class in order to change the rendered page.

OptionBody Class

With OptionBody class, users can adjust screen to Fullscreen, adjust volume and select the music that can be played at the background of the game.

Method:

- **private handleFullscreen():** This method is used to maximize the application window. If the screen is not in fullscreen mode then when this method called screen will be in fullscreen mode, otherwise, it will exit fullscreen mode.

Header Class

This class is used in every page except the MainPage class. It is used for navigation inside the application. It provides users to navigate the current page to the previous page.

Constructor:

- **public Header(setPage : function, prevPageName : string, prevPageTitle : string):** This is a constructor for the Header class. It takes the setPage function as the parameter which comes from the App class. The prevPageName parameter is the value which is set as the page value by using the setPage function and the prevPageTitle parameter is the title of the previous page of the current page. setPage function is activated only when the navigate option is activated.

RoomOptionPage Class

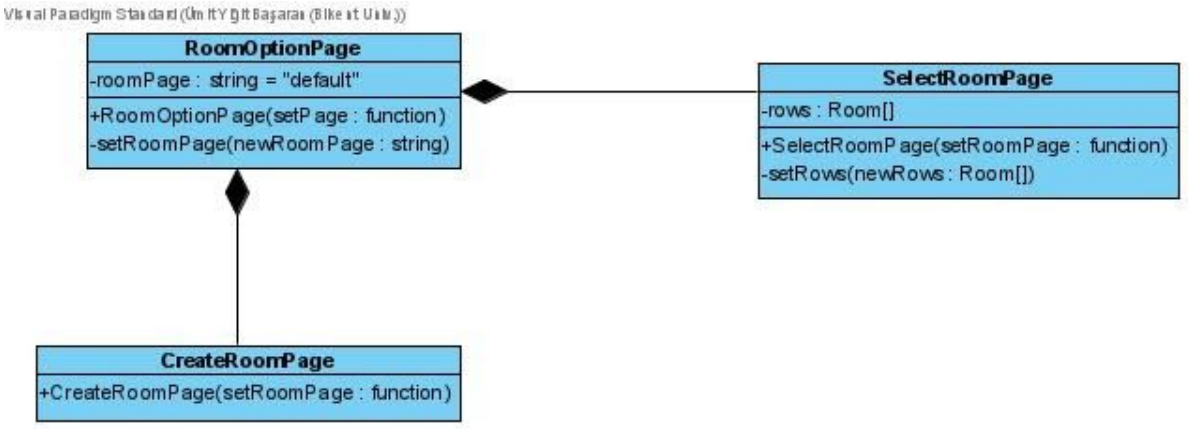

Figure 17 - RoomOptionPage Class

This class provides users the ability to go create a room page or select a room page.

Attribute:

- **private roomPage : string = “default”:** This attribute is used for selecting the operations related to rooms. In default mode it renders a monopoly logo and two options.

Constructor:

- **public RoomOptionPage(setPage : function):** This is a constructor for RoomOptionPage class. The setPage parameter is taken from App class to set the currently rendered page to the desired page.

Method:

- **private setRoomPage(newRoomPage : string):** This method is used for adjusting the default room option page to either “createRoomPage” or “selectRoomPage”.

CreateRoomPage Class and Its Dependent Classes

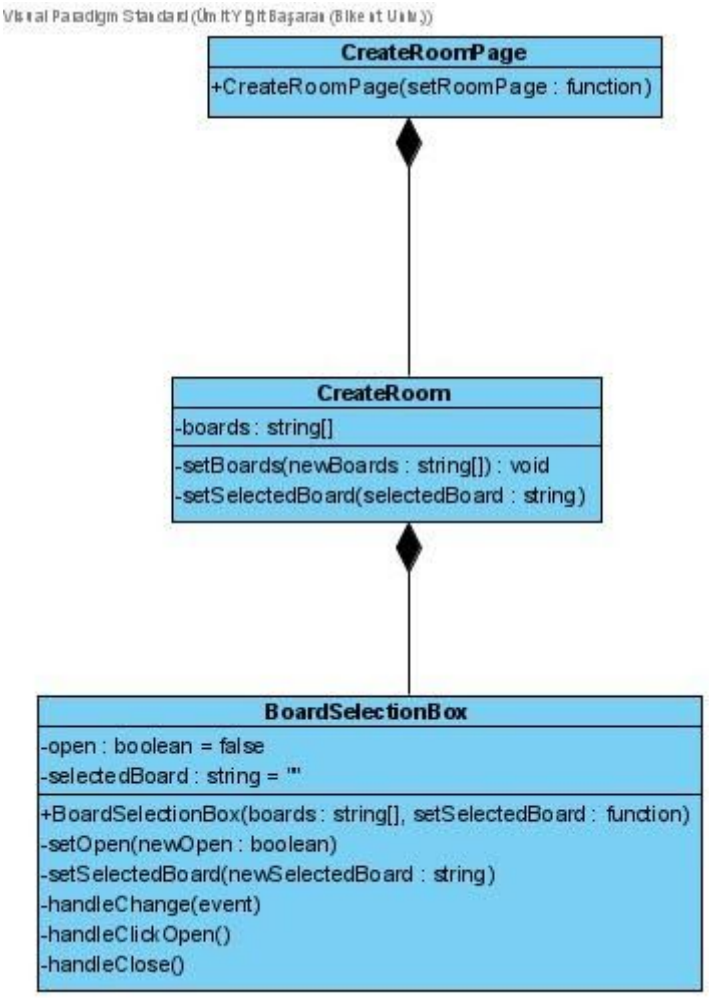

Figure 18 - CreateRoomPage Class and Its Dependent Classes

CreateRoomPage Class

This class consists of the Header class which is described in early sections of this report and the CreateRoom class which has BoardSelectionBox class.

Constructor:

- **public CreateRoomPage(setRoomPage : function):** This is a constructor for the CreateRoomPage class. It takes the setRoomPage function as a parameter from the RoomOptionPage class to set the room page.

CreateRoom Class

This class provides users the ability to create a room with room name and room password (this option is not required). Also the CreateRoom class has a BoardSelectionBox class.

Attribute:

- **private boards : string[]:** This attribute is a string array which contains the possible names for the board types.

Methods:

- **private setBoards(newBoards : string[]) : void:** This method takes a new string array which contains the possible names for the board types and replaces it with the old one.
- **private setSelectedBoard(selectedBoard : string):** This method is used to set the returned Room object which holds room name, password and selected board type. Because board type is taken from the BoardSelectionBox class this function is passed to the BoardSelectionBox class as a parameter.

BoardSelectionBox Class

This class lists all possible board templates.

Attributes:

- **private open : boolean = false:** This attribute is used to determine the state of the BoardDialogBox. When open true the dialog box opens otherwise it closes.
- **private selectedBoard : string = “”:** When a user selects a board template from the selection box this attribute is set to the selected value of the board template.

Constructor:

- **public BoardSelectionBox(boards : string[], setSelectedBoard : function):** This is a constructor for BoardSelectionBox. It takes a string array called boards as a parameter from CreateRoom class and uses it to fill selection box options. Also, it takes the setSelectedBoard function as a parameter to set the selectedBoard state of CreateRoom class.

Methods:

- **private setOpen(newOpen : boolean):** This method changes the open state to the newOpen state.
- **private setSelectedBoard(newSelectedBoard : string):** This method takes a new string which represents the board template and sets it as selectedBoard state.
- **private handleChange(event):** This method is used for taking selected board template values from the selection box.
- **private handleClickOpen():** This method is used when the button which opens the dialog box is pressed.
- **private handleClose():** When the activity of the user is done, this method is called and it closes the dialog box.

SelectRoom Class and Its Dependent Classes

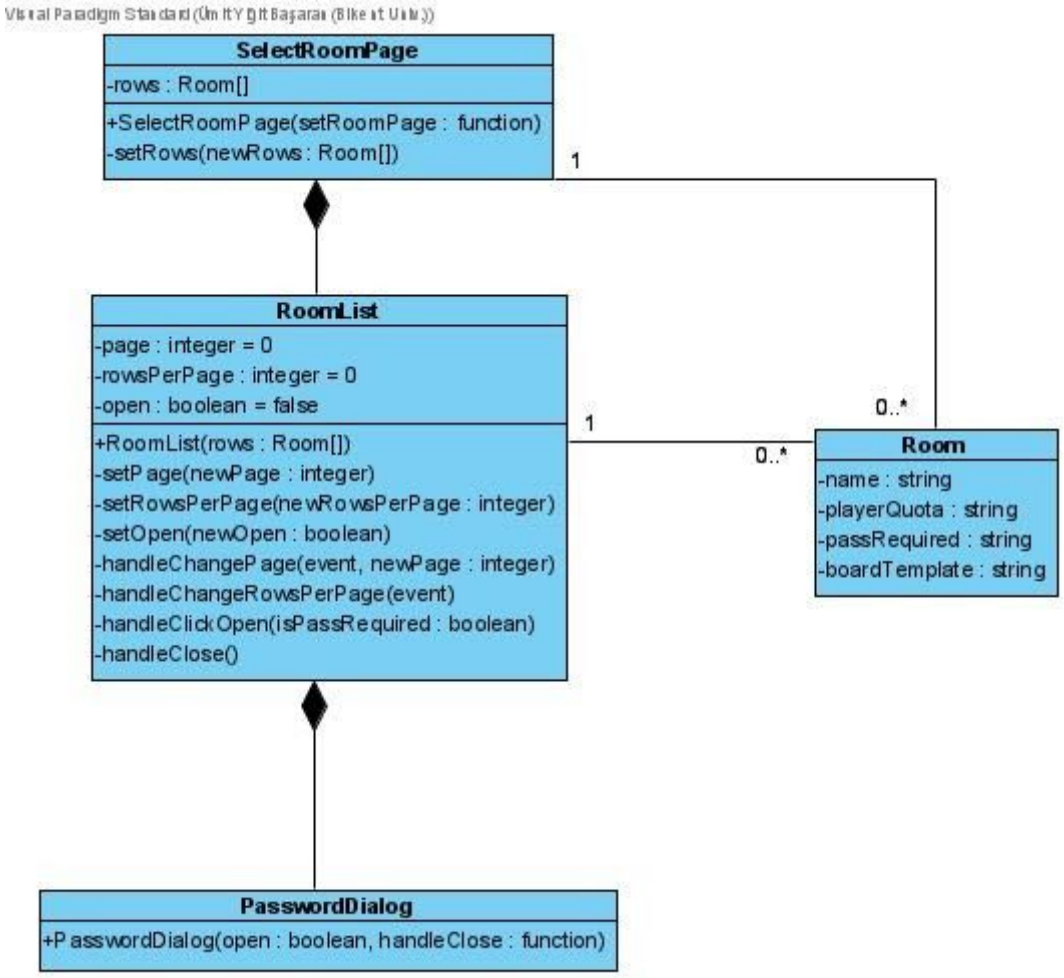

Figure 19 - SelectRoom and Its Dependent Classes

SelectRoomPage Class

This class consists of the Header class, the RoomList class, the PasswordDialog class and the Room class.

Attribute:

- **private rows: Room[]:** This attribute is used for taking Room data from a server as Room objects and holding them into an array.

Constructor:

- **public SelectRoomPage(setRoomPage : function):** This is a constructor for SelectRoomPage class. It takes the setRoomPage function as a parameter and uses it for setting the rendered page in RoomOptionPage class.

Method:

- **private setRows(newRows : Room[]):** When this method is called, rows array is set to newRows array.

RoomList Class

This class lists all available rooms and their information. Also for each room there is a join option to this room.

Attributes:

- **private page : integer = 0:** This attribute is used for pagination in the table (Page number).
- **private rowsPerPage : integer = 0:** This attribute is used for pagination in the table (Number of items per page).
- **private open : boolean = false:** This attribute is used to adjust the state of the password dialog.

Constructor:

- **public RoomList(rows : Room[]):** This is a constructor for RoomList class. It takes an array as a parameter and shows it in the table.

Methods:

- **private setPage(newPage : integer):** This method is used to set the displayed page number of the table to newPage parameter.
- **private setRowsPerPage(newRowsPerPage : integer):** This method is used to set the number of items that is displayed in one page of the table to newRowsPerPage parameter.

- **private setOpen(newOpen : boolean):** This method sets the open state to newOpen state.
- **private handleChangePage(event, newPage : integer):** This method listens to the page state and when it changes it sets the page state.
- **private handleChangeRowsPerPage(event):** This method listens to the rowsPerPage state and when it changes it sets the rowsPerPage state
- **private handleClickOpen(isPassRequired : boolean):** This method is used to adjust the state of the password dialog. If the password required for this room and join option is activated then it activates the PasswordDialog class otherwise it doesn't.
- **private handleClose():** This method is used to close the PasswordDialog class window.

Room Class

This class has JSON object notation. It will be used for transferring the room data between user interface and the server.

Attributes:

- **name : string:** This attribute represents the name of the Room.
- **playerQuota: string:** This attribute represents the state of the room. It shows the availability to the room.
- **passRequired: string:** This attribute represents the state of whether a password is required for this room or not ("Yes" or "No").
- **boardTemplate: string:** This attribute represents the name of the template of the board which is specified for a specific room.

PasswordDialog Class

This class is used for creating a dialog box that takes a password which the user needs to enter in order to enter the room.

Constructor:

- **public PasswordDialog(open : boolean, handleClose : function):** This is a constructor for PasswordDialog class. It takes a boolean to either open it or close it. Also, it takes the handleClose function as a parameter from RoomList class and uses it to close the dialog box.

RoomLobbyPage Class and Its Dependent Classes

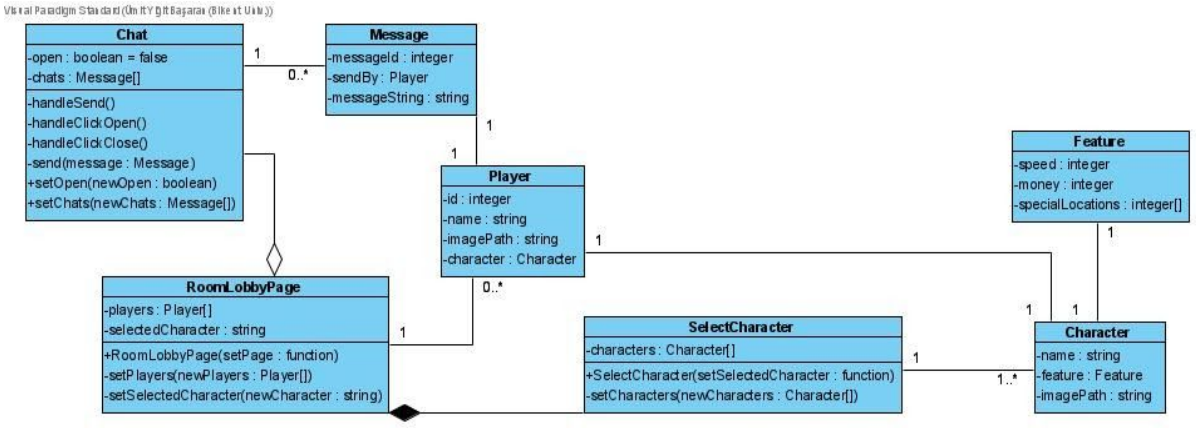

Figure 20 - RoomLobbyPage Class and Its Dependent Classes

RoomLobbyPage Class

This class consists of the Header class, the Chat class, the SelectCharacter class. Also, this class lists all the players for a room.

Attributes:

- **private players : Player[]:** This attribute is used to list and show all player objects in a card layout.

- **private selectedCharacter: string:** This attribute represents the selected character for the user.

Constructor:

- **public RoomLobbyPage(setPage : function):** This is a constructor for RoomLobbyPage class. It takes the setPage function as a parameter and uses it to set the rendered page.

Methods:

- **private setPlayers(newPlayers : Player[]):** This method is used to set players state to newPlayers state.
- **private setSelectedCharacter(newCharacter: string):** This method is used to set selectedCharacter state to newCharacter state.

Player Class

This class has JSON object notation. It will be used for transferring the player data between user interface and the server.

Attributes:

- **id : integer:** This attribute represents the id of the player.
- **name : string:** This attribute represents the name of the player.
- **imagePath : string:** This attribute is used for getting the image of the player.
- **character : Character:** This attribute represents the player's selected character.

Chat Class

This class is a user interface for chatting. In this window, users can chat with each other while either in the room lobby or in the game.

Attributes:

- **private open : boolean = false:** This attribute represents the state of the chat window as either it is open or not.
- **private chats : Message[]:** This attribute holds all the messages which come from the server side.

Methods:

- **private handleSend():** This method listens to the send activity and when the activity is activated it calls send method.
- **private handleClickOpen():** This method listens to the open chat activity and when the activity is activated it opens the chat window.
- **private handleClickClose():** This method listens to the close chat activity and when the activity is activated it closes the chat window.
- **private send(message : Message):** This method sends a Message object to the server to display the message to all the players.
- **private setOpen(newOpen : boolean):** This method sets the open state to newOpen parameter.
- **private setChats(newChats : Message[]):** This method sets the chats state to newChats parameter.

Message Class

This class has JSON object notation. It will be used for transferring the message data between user interface and the server.

Attributes:

- **messageId : integer:** This attribute represents the id of the message.
- **sendBy : Player:** This attribute represents the player who sends the current message.
- **messageString : string:** This attribute represents the message string.

SelectCharacter Class

This class lists all Character options.

Attribute:

- **private characters : Character[]:** This attribute represents all possible characters.

Constructor:

- **public SelectCharacter(setSelectedCharacter : function):** This is a constructor for SelecterCharacter class. It takes the setSelectedCharacter function as a parameter from RoomLobbyPage class and uses it to set the selectedCharacter for a player.

Method:

- **private setCharacters(newCharacters : Character[]):** This method sets the characters state to newCharacters parameter.

Character Class

This class has JSON object notation. It will be used for transferring the character data between user interface and the server.

Attributes:

- **name : string:** This attribute represents the name of the character.
- **feature : Feature:** This attribute represents the features of this character.
- **imagePath : string:** This attribute is used to display a character's image.

Feature Class

This class has JSON object notation. It will be used for transferring the feature data between user interface and the server.

Attributes:

- **speed : integer:** This attribute represents the speed of the character. In this game with speed property a character can pass multiple tiles at once.

- **money : integer:** This attribute represents the money to be taken from the bank when every time a player completes a tour at the board.
- **specialLocations : integer[]:** This attribute represents locations in the table which are special for a special character.

GameScreenPage Class and Its Dependent Classes

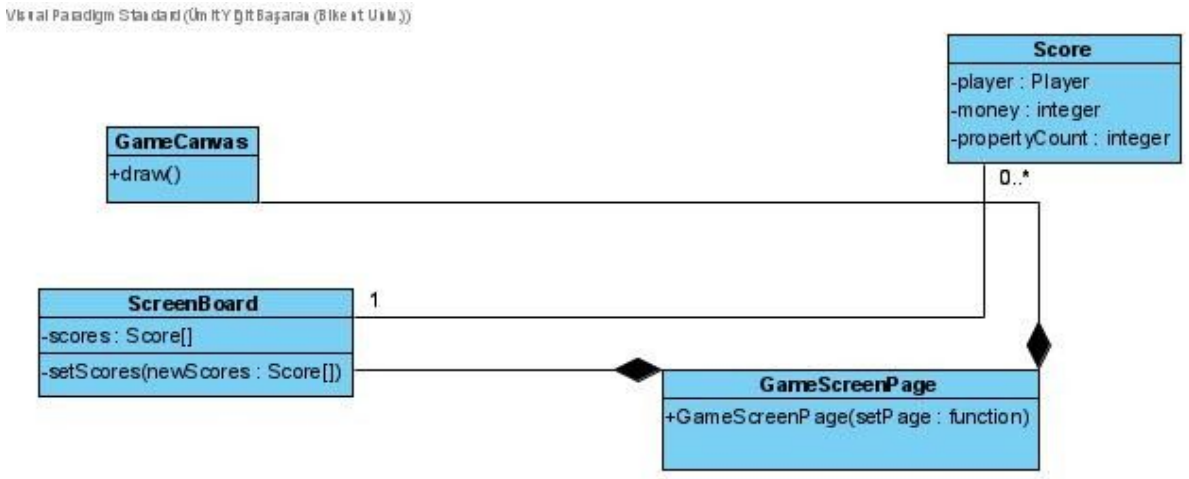

Figure 21 - GameScreenPage Class and Its Dependent Classes

GameScreenPage Class

This class is the user interface of the gameplay. The game will be played on a canvas which will be implemented by using Pixi.js. Therefore, this class contains the ScreenBoard class, the Chat class and the GameCanvas class.

Constructor:

- **public GameScreenPage(setPage : function):** This is a constructor for the GameScreenPage class. It takes the setPage function as a parameter from App class to set the rendered page.

ScreenBoard Class

This class is the user interface that displays a table that contains the Score object information.

Attribute:

- **private scores : Score[]:** This attribute holds the all possible Score object data and is used for displaying these data.

Method:

- **private setScores(newScores : Score[]):** This method sets the scores state to newScores parameter.

Score Class

This class has JSON object notation. It will be used for transferring the score data between user interface and the server.

Attributes:

- **player : Player:** This attribute represents the player who owns this score.
- **money : integer:** This attribute represents the amount of money that player has.
- **propertyCount : integer:** This attribute represents the amount of property that player has.

GameCanvas Class

This class is used for connecting the user interface that will be implemented with React and the user interface that will be implemented with Pixi.js.

Method:

- **public draw():** This method provides connection between the canvas and user interface.

Card Class

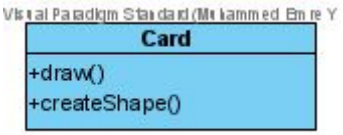

Figure 22 - Card Class

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** createShape function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

CityCard Class

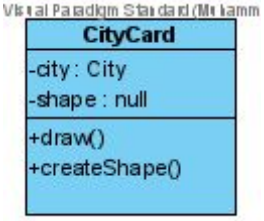

Figure 23 - CityCard Class

Attributes:

- **city : City:** It holds the city that it belongs to.
- **shape : null :** It is default null and it holds the shape of city card.

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.

- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

Board Class

Figure 24 - Board Class

Attributes:

- **tiles: Tile[] :** It holds all tiles which are drawing and included on board.
- **Dice : Object:** It is the dice and its drawing.

UsableCard Class

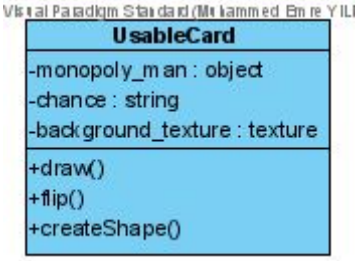

Figure 25 - UsableCard Class

Attributes:

- **monopoly_man : object :** It holds the png file of the monopoly man in order to use the background of the usable cards.

- **Chance : string :** It holds the description of the usable cards such as given tasks

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** createShape function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.
- **filp():** flip function purpose is these cards firstly should be off. When a user comes to a chance tile, the card becomes on using the flip function.

Tile Class

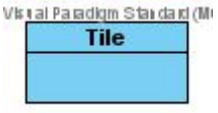

Figure 26 - Tile Class

This class is the father of every type of tile.

StartTile Class

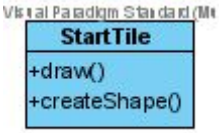

Figure 27 - StartTile Class

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

ParkTile Class

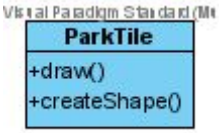

Figure 28 - ParkTile Class

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

ChestTile Class

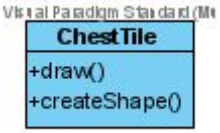

Figure 29 - ChestTile Class

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

JailTile Class

Figure 30 - Jail Tile Class

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

StationTile Class

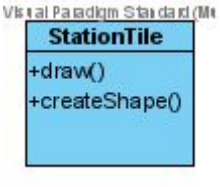

Figure 31 - StationTile Class

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

CityTile Class

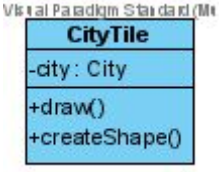

Figure 32 - CityTile Class

Attributes:

- **city : City:** It holds the city that it belongs to.

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

UtilityTile Class

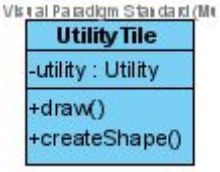

Figure 33 - UtilityTile Class

Attributes:

- **utility: Utility:** It holds the utility that it belongs to like city tile.

Methods:

- **draw():** Draw function purpose is checking if the card is drawn before or not in order to avoid multiple drawing.
- **createShape():** This function purpose is drawing the card using Pixi's rectangle method. Using the container architecture, cards become nearly the same with monopoly's hard cards.

GameManager class

Figure 34 - GameManager Class

Game manager class will handle all the managers and establish the connection between all of them. The data flow between different managers will be handled in this class. We can consider GameManager as the “Main controller” class.

When the game starts, we will construct a GameManager object and it will initialize other controllers such as StateManager, CardManager, EventManager, NetworkManager, TradeManager, BoardManager and PlayerManager.

We can consider all of these managers different services, works and manage a subsystem of Monopoly games.

Attributes:

- **-board: BoardManager:** Contains the BoardManager object for Monopoly Game.
- **-stateManager : StateManager:** Contains the StateManager object for Monopoly Game.
- **-tradeManager : TradeManager:** Contains the TradeManager object for Monopoly Game.

- **-playerManager : PlayerManager:** Contains the PlayerManager object for Monopoly Game.
- **-cardManager : CardManager:** Contains the CardManager object for Monopoly Game.
- **-eventManager : EventManager:** Contains the EventManager object for Monopoly Game.
- **-networkManager : NetworkManager:** Contains the NetworkManager object for Monopoly Game.
- **reactRoot: ReactApp:** Contains the root of react.

Methods:

- **playDice():** This method rolls two dice and returns the result. It will be used in the starting of turns and different events such as escape from jail and minigames.
- **nextTurn():** This method will give a signal to StateManager in order to end the current turn and start a new one.
- **trade(traders: Player[], item: Property):** This method creates a trade via TradeManager. It will be used in cases where the user doesn't want to buy a tile in his turn.
- **buildHouse(cityId : Number):** This method will build a house in the given city if it satisfies requirements.
- **buildHotel(cityId : Number):** This method will build a hotel in the given city if it satisfies requirements.
- **sellHouse(cityId : Number):** This method will sell a house in the given city if it satisfies requirements.
- **sellHotel(cityId : Number):** This method will sell a hotel in the given city if it satisfies requirements.
- **enterJail():** When this method is called, currentPlayer will go to jail. It will call needed functions inside StateManager, PlayerManager, EventManager and BoardManager.

- **exitJail():** When this method is called, currentPlayer will exit from jail. It will call needed functions inside StateManager, PlayerManager, EventManager and BoardManager.
- **move(tileId: Number):** When this method is called, currentPlayer will go to the desired tile. It will call needed functions inside StateManager, PlayerManager, EventManager and BoardManager.
- **checkEvent():** This function will search is there any event must occur at the current turn via EventManager.

PlayerManager

Figure 35 - PlayerManager Class

PlayerManager class will handle the dataflow about a player. In Monopoly, a user can obtain cards, use cards, gain or lose money. It will hold all the player data in a game.

Attributes:

- **-players: Player[]:** An array contains all the players as a Player object.

Methods:

- **addCard(playerId: Number, cardId: Number):** Add a card to a player. A card can be a CityCard, UtilityCard, ChanceCard etc.
- **removeCard(playerId: Number, cardId: Number):** Remove a card from a player. A card can be a CityCard, UtilityCard, ChanceCard etc.

- **addMoney(playerId: Number, amount: Number):** Add money to a player. This function will be called from GameManager in situations like selling a property or passing over the starting tile.
- **removeMoney(playerId: Number, amount: Number):** Remove money from a player. This function will be called from GameManager in situations like buying a property or as a result of an event.
- **getMoney(playerId: Number):** This method will return the amount of money a player has got.

BoardManager

Figure 36 - BoardManager Class

BoardManager class handles movements in the game board. Also according to the selected board in the room creation, it will create a special board.

Attributes:

- **-board: Board:** Board object which will be created at the creation of GameManager and BoardManager.

Methods:

- **setBoard(Board newBoard):** This method will change the current board property to newly given one. It will be called at the initialization of GameManager.
- **setPosition(playerId: Number, tileId: Number):** Set the player position found by the playerId to tile found by tileId. This method will use PlayerManager in order to change player position.

- **getPosition(playerId: Number):** Get the player position found by the playerId and return the tileId associated with the tile which player was on top of. This method will use PlayerManager in order to get the position of the player.

TradeManager

Figure 37 - TradeManager Class

TradeManager class handles the trade system which starts when a user doesn't want to buy a property. It handles the loop and logic about trading and auction systems.

Attributes:

- **players: Player[]:** Player array contains the players enter the auction loop for current trade.
- **bids: Number[]:** Array of bids given from players enter the auction.

Methods:

- **createTrade(players: Player[]):** This method assigns players array to the given array and starts the loop of auction until it is finished.
- **setBid(playerId: Number, amount: Number):** Set the bid comes from a user to the bids array.
- **closeTrade():** Finish a trade and return a result object containing the changes on players.

StateManager

Figure 38 - StateManager Class

This object contains the current state of the game. GameManager will use StateManager to change and get the current state. State objects will be used in order to synchronize the current game between different users via localhost or a server. It is the main component of syncing an online game.

Attributes:

- **currentState: State:** Current state object which contains all the data about current board and players.

Methods:

- **changeState(newState: State):** This method is called when there is a change in the current state of the game. It takes the new State object.
- **getState():** Return the current state to GameManage

NetworkManager

Figure 39 - NetworkManager Class

The NetworkManager class will handle the online connection via websocket connection. We will use socket.io for implementation. Besides implementation, NetworkManager gives GameManager an interface for listen different channels on socket connection.

Attributes:

- **websocket: IOWebSocket:** Websocket object which will be used for all the connections.
- **id:** Id of current client.

Methods:

- **subscribeChannel(channelName: String):** Subscribe to the channel given as a parameter. In other words, it creates a listener for a given channel.
- **unsubscribeChannel(channelName: String):** Unsubscribe to the channel given as a parameter.In other words, it removes a listener for a given channel.
- **createSocketConnection():** Creates the websocket object at the starting phase of the game.
- **closeSocketConnection():** Closes the websocket object at the ending phase of the game directly before exit from the game.

CardManager

Figure 40 - CardManager Class

CardManager class will load a card set for a game and will be used for searching a card in the card set.

Attributes:

- **cards: Card[]:** Cards array contains all the cards in the current game.

Methods:

- **addCard(newCard: Card):** Add a new card to cards array. It will be used when the game starts by GameManager.
- **removeCard(itemId: Number):** It will remove a card from the cards array. It will be used in special quests in order to remove a card from the current game.
- **useCard(cardId: Number):** It will execute the action of the given card if it is a usable card.For example we will use this function for playing chance cards.
- **searchCard(CardProps query):** This function takes a query object and according to the query it will return a card from the cards array if it exists in the array.

CardProps class

Figure 41 - CardProps Class

This class is a wrapper for a query object which will be used for searching a card from cards array. It contains different parts of a card for searching.

Attributes:

- **cardId: Number:** The card id assigned at the creation of cards array process.
- **cardName: String:** The card name for a special card. For example “Chance Card” or “Ankara”
- **cardOwner: Number:** The id of the player which is the owner of a card.

- **cardType: Number:** The type of class. Numbers represent some constants such as usable cards, chance cards, city cards etc.

EventManager

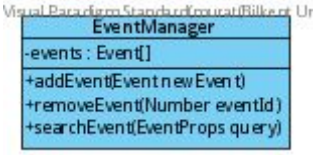

Figure 42 - EventManager Class

EventManager contains different events and controls the flow of events among the whole game. Actually we can assign a lot of different events therefore EventManager will be used a lot. We can give “pay to a player for passing over a city” or “passing over the starting point” as different examples for events.

Attributes:

- **events: Event[]:** An array contains all the events as an array.

Methods:

- **addEvent(newEvent: Event):** Add a new event to the events array. It will be used when a user buys a new city, we will add an event that affects other players in order to transfer money when they pass over from this city.
- **removeEvent(eventId: Number):** Remove a new event to the events array. It will be used when a user sells a city, we will remove the event that affects other players in order to transfer money when they pass over from this city.
- **searchEvent(query EventProps):** This function takes a query object and according to the query it will return an event from the events array if it exists in the array.

EventProps class

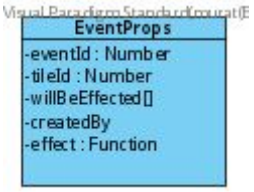

Figure 43 - EventProps Class

This class is a wrapper for a query object which will be used for searching an event from an events array. It contains different parts of an event for searching.

Attributes:

- **eventId: Number:** The event id assigned at the creation of events array process.
- **tileId: Number:** Tile id which event will effect.
- **willBeEffected: Player[]:** An array of players which will be affected from the event.
- **createdBy: Player:** Player which is the creator of the event. For general events such as pass from the starting point, it will be null.
- **effect: Function:** Function which changes willBeEffected array and createdBy according to the mission of the event. For example, when a player B passes from a city which was bought by player A,this function calls removeMoney(200) for player B and addMoney(200) for player A.

City class

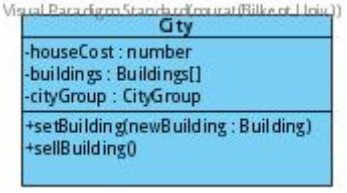

Figure 44 - City Class

Attributes:

- **houseCost : number:** Defines the cost of building a house in that city after completing the requirement that a player should own all the tiles in the same color in order to build a city and a hotel to that tile.
- **buildings : Building[]:** A list that holds all the buildings that are constructed in the city by a player.
- **cityGroup : CityGroup:** Holds the information about the city group that the specific city belongs to.

Methods:

- **setBuilding(newBuilding : Building):** Builds a new building in that city, whether a hotel or a house.
- **sellBuilding():** This method is used for selling buildings in a city that is owned by a player.

Player class

Figure 45 - Player Class

Attributes:

- **id : number:** Holds the id number of a player. For each of the players in a game, each player is identified by a unique id number.
- **username : string:** Holds the username of a player.
- **avatar : object:** Avatar is like a profile picture, and the avatar attribute holds the object that is responsible for holding the avatar of a player.
- **properties : Property[]:** A list that holds all the properties that are owned by a player.
- **cards : Card[]:** A list that holds all the cards that are owned by a player. Those cards can be UsableCard or PropertyCard.
- **currentTile : number:** This attribute is used for holding the information regarding the current tile of a user.

Methods:

- **move(newTileId : number):** This method is used to move players to a new tile. It takes a tile id as a parameter that corresponds to the tile that a player needs to be moved.
- **addProperty(newProperty : Property):** Adds a property to a player. It takes a parameter specifying the property type.
- **addCard(newCard : Card):** Adds a card to a player. The card type is specified using the parameter newCard.
- **removeCard(card : Card):** Removes the card from the card deck that is owned by a player.

Property class

Figure 46 - Property Class

Attributes:

- **id : number:** Like the id attribute of Player class, a property is uniquely identified by its id number.
- **name : string:** All properties have a name. This attribute holds the name of properties.
- **rentPrice : number[]:** Rent price that players except the owner need to pay depends on the building number and type of that city. This array holds the rent prices for all combinations, such as rent price for a property having one house or two houses.
- **mortgagePrice : number:** Holds the information regarding the mortgage price of a property.
- **price : number:** Specifies the information about the price of a property that a player needs to pay in order to buy the property.
- **ownerId : number | null:** This attribute specifies the owner of a property. It is null if none of the players own the city, the id of the owner else.
- **tile : CityTile:** Used for creating the association between Tile class.

- **card : CityCard:** Used for creating the association between Card class.

Methods:

- **sell(newOwner : Player):** Sells the property to a player specified with the parameter given as newOwner.
- **mortgage():** Mortgages the given property.
- **isOwned():** Returns true if the property is owned by any of the players, false otherwise.
- **buy(owner : Player):** Player is able to buy the property within this method. After the execution of the method, given property is assigned to the given player in the parameter.

Building class

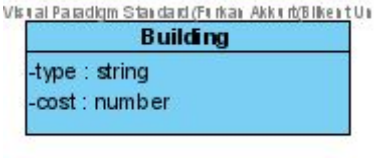

Figure 47 - Building Class

Attributes:

- **type : string:** Holds the information about the type of a building. It is whether a hotel or a house.
- **cost : number:** Cost of building the specified type.

CityGroup class

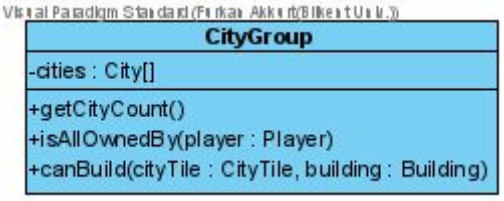

Figure 48 - CityGroup Class

Attributes:

- **cities : City[]:** Holds all the cities belonging to a specific city group.

Methods:

- **getCityCount():** Returns the number of cities belonging to that city.
- **isAllOwnedBy(player : Player):** Returns true if a player given in the parameter owns all the cities in the specific city group, false otherwise.
- **canBuild(cityTile : CityTile, building : Building):** Returns true if a building can be built to a city tile, false otherwise. Note that a building can be built to a city if and only if all cities belonging to the same city group are owned by the same player.

State class

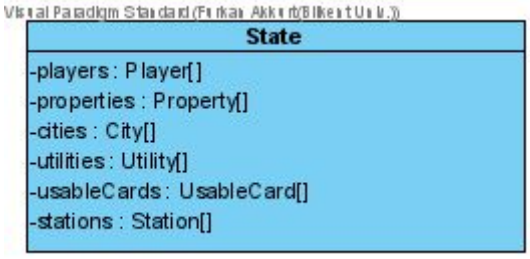

Figure 49 - State Class

State class is the class which holds the whole information about the current state of the game. For instance, when a game is paused, in order to continue from the current state of the game, State class is needed. In addition to that, if the game was not only multiplayer based and was also supporting single player game mode, when a player would click on save and exit, the state would hold the information, transfer it to a database, and the player would continue from where s/he left off.

Attributes:

- **players : Player[]:** Holds the information about the current players in the game.
- **properties : Property[]:** Holds the information about the state of the properties of the game. An example of a state can be given as whether a property is owned by a player or not.
- **cities : City[]:** Such as the state of the properties, the state of the cities are also kept in an attribute in State class.
- **utilities : Utility[]:** Current state of the utilities (Utility Tiles) are held in the given attribute.
- **usableCards : UsableCard[]:** Current state of the cards is held in the given attribute.
- **stations : Station[]:** Current state of the stations is held by the given attribute.

Utility class

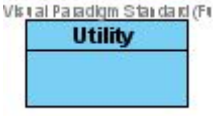

Figure 50 - Utility Class

A model class that holds the information about the utilities in the game.

UsableCard class

Figure 51 - UsableCard Class

A model class that holds the information about the usable cards (Community Chest Cards or Chance Cards) in the game.

Station class

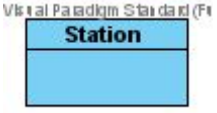

Figure 52 - Station Class

This class is to represent the data part of the stations. It has nothing to do with the UI part, it just holds the data attributes.

Users Class

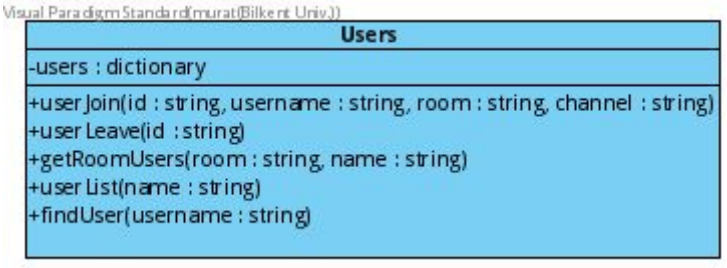

Figure 53 - Users Class

In users class, users are stored with their username, specific id that is assigned from socket.io, their room name and their current channel.

Attributes:

- **users: Dictionary:** In dictionary users stores as lists. Each user has [id, username, room, channel].

Methods:

- **userJoin(id:String, username:String, room:String, channel:String):** In userJoin method, users are added to the users dictionary, if the users does not exist. Users are checked via their username and id. If the username or id exist, the user is not added.
- **userLeave(id:String):** When a user exits the game, user is deleted from the users dictionary and chat screen.
- **getRoomUsers(room:String, name:String):** Returns the other users except the current user that wants to see the users.
- **userList(name:String):** Returns the users except the user that username is name.
- **findUser(username:String):** Returns the user that given username.

Server Class

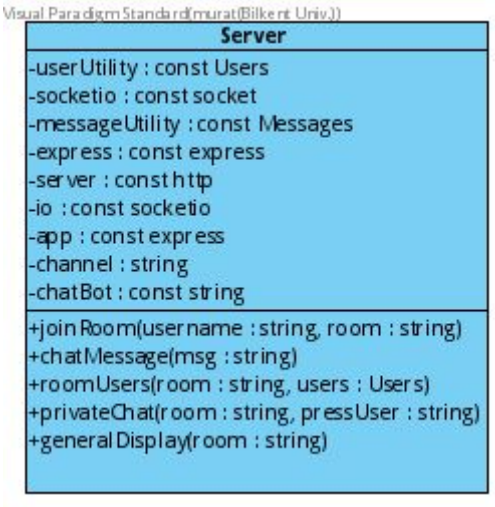

Figure 54 - Server Class

Server class is where the backend is handled.

Attributes:

- **userUtility: const Users:** Creates a users class that can access the users and users class's methods.
- **socketio: const socket:** Creates a class variable of socket.io to access it's attributes and methods.
- **messageUtility: const Messages:** Creates a messages class that can access the messages and messages class's methods.
- **express: const express:** Necessary for online server creation.
- **server: const http:** Necessary for online server creation.
- **io: const socketio:** Creates a class variable of socket.io to access it's attributes and methods.
- **app: const express:** Necessary for online server creation.
- **channel: String:** Holds the current user's channel (to whom with messaging).
- **chatBot: const String:** System messages sent via the name of chatBot.

Methods:

- **joinRoom(username:String, room:String):** When a new user selects a room, this method is called.
- **chatMessage(msg:String):** When a user sends a message this method is called. This method is called from the frontend. The call is made over socket.
- **roomUsers(room:String, users:Users):** This method sends a request to the frontend over socket, to print out the users in the room.
- **privateChat(room:String, pressUser:String):** When a user wants to chat with another user, this method is called. Method is called from the frontend over socket. This method returns the messages between the user and the other user.

- **generalDisplay(room:String):** When a user wants to see the general lobby, this method is called from the frontend over socket. This message returns the general lobby messages.

Messages Class

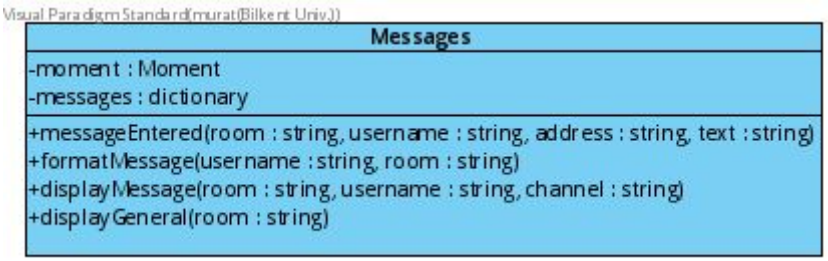

Figure 55 - Messages Class

Attributes:

- **moment: Moment:**Creates a moment object to learn the time.
- **messages: dictionary:** Holds the every message in every room by their room id, username, address (to whom), text and the time that the message is sent.

Methods:

- **messageEntered(room:String, username:String, address:String, text:String):** When a new message is entered, this method stores the message in the messages dictionary.
- **formatMessage(username:String, text:String):** When a message is sent, this method returns the username, the text and the time of the message sent.
- **displayMessage(room:String, username:String, channel:String):** Returns the message list that the username user's and channel user's messages in the current room.
- **displayGeneral(room:String):** Returns the general lobby's messages in the given room.

Main Class

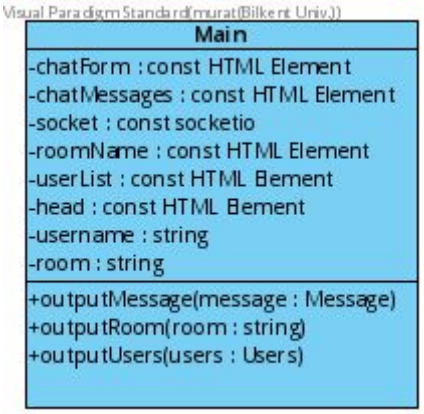

Figure 56 - Main Class

Main class is where the frontend is handled.

Attributes:

- **chatForm: const HTML Element:** Is used for holding the chat display's HTML.
- **chatMessaages: const HTML Element:** Is used for passing the messages into the HTML.
- **socket: const socketio:** Is used for responding socket handlers or sending socket handlers.
- **roomName: const HTML Element:** Is used for passing the room name to the HTML.
- **userList: const HTML Element:** Is used for passing the user names in HTML.
- **head: const HTML Element:** Is used for passing the chat name to the HTML.
- **username: String:** It receives the username from the URL link.
- **room : String:** It receives the room name from the URL link

Methods:

- **outputMessage(message:Message):** Parses the messages into the chatMessages HTML.
- **outputRoomName(room:String):** Parses the room name into the roomName HTML.
- **outputUsers(users:Users):** Parses the usernames into the userList HTML.

Path Class

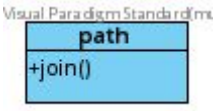

Figure 57 - path Class

Path is necessary for creating a url path.

Methods:

- **join():** Joins all arguments together and creates a normalized path argument.

HTTP Class

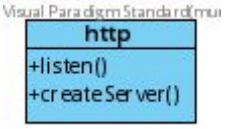

Figure 58 - http Class

Is necessary for creating a server.

Methods:

- **+listen():** Listens to the PORT logins.
- **+createServer():** Used for creating the server.

Moment Class

Figure 59 - Moment Class

Moment class is used to retrieve the time that the message is sent.

Methods:

- **format(format:String):** Returns the time in desired format.
- **moment():** Returns the exact moment.

4.4 Packages & Frameworks

Pixi.JS

Pixi is a rendering framework which uses HTML Canvas and WebGL according to browser support. It gives us a uniform interface for rendering in these two different platforms. It gives us a Scene object in order to hierarchically store them and access them later. It also distinguishes listeners among these objects. Therefore we can easily catch an object and assign a private listener just for this item.

Socket.IO

Socket io is one of the easiest real-time libraries that Javascript has. It is used for communication between the user(client) and the server. It uses Node.js for the server side, hence they have a similar API. Socket.io is an event driven language, therefore, client side and server side can simultaneously interact with each other [4], [5].

Node.JS

Node.JS is an asynchronous event-driven Javascript runtime environment. It is mainly used in backend/server and used in to build network applications which are scalable. In Node.JS connecting to the server can be handled simultaneously which is essential for the web. Also with the help of Node, we can publish our game for different operating systems because all the operating systems support javascript via V8 engine which was developed by Chromium project. [6]

react Package

“react” package is an entry point to access React.js library. Every component and class should import the react package as “import React from ‘react’” in order to be a React component or class [7].

@material-ui/core Package

Material UI is a predefined React component library. This library is used in order to ease the design process of the components with predefined good looking components. @material-ui/core is the core package for Material UI library. With this package, every component such as Button component, Grid component, Table Component etc. which are defined in Material UI can be used in React projects [8].

@material-ui/icons

@material-ui/icons package is a icon package for @material-ui/core components. By using this package, icons can be added to any component [9].

formik

formik is used to reduce the boilerplate in the React forms. It automatically handles all form values when values are set in the formik hook (useFormik()) [10].

REFERENCES

[1] King.com. (n.d.). Retrieved November 29, 2020, from https://www.king.com/tr/game/candycrush

[2] “React – A JavaScript library for building user interfaces,” – *A JavaScript library for building user interfaces*. [Online]. Available: https://reactjs.org/. [Accessed: 29-Nov-2020].

[3] “Introducing Hooks,” *React*. [Online]. Available: https://reactjs.org/docs/hooks-intro.html. [Accessed: 29-Nov-2020].

[4] “Introduction | Socket.IO.” https://socket.io/docs/ (accessed: Nov. 29, 2020).

[5] “bradtraversy/chatcord: Realtime chat app with rooms - GitHub.” https://github.com/bradtraversy/chatcord (accessed: Nov. 29, 2020).

[6] “About | Node.js.” https://nodejs.org/en/about/ (accessed: Nov. 29, 2020).

[7] “React Top-Level API,” *React*. [Online]. Available: https://reactjs.org/docs/react-api.html. [Accessed: 29-Nov-2020].

[8] “@material-ui/core,” *npm*. [Online]. Available: https://www.npmjs.com/package/@material-ui/core. [Accessed: 29-Nov-2020].

[9] “@material-ui/icons,” *npm*. [Online]. Available: https://www.npmjs.com/package/@material-ui/icons. [Accessed: 29-Nov-2020].

[10] “formik,” *npm*. [Online]. Available: https://www.npmjs.com/package/formik. [Accessed: 29-Nov-2020].